Onlinevarsity
your e-way to learning

Learn Java By Example Learner's Guide

Are you □□□□□□□□□□ with Onlinevarsity.com?	R G E T R E E S I D
Did you □□□□□□□□ this ebook?	D O O W L N A D
Do you have a □□□□□ copy of this ebook?	L G A E L
Are you a victim of □□□□□□?	P C I A R Y

Answers: REGISTERED, DOWNLOAD, LEGAL, PIRACY

Download a **LEGAL** copy of this ebook which you can say is **mine** because **PIRACY** is a **crime**

Learn Java By Example
Learner's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2013

Dear Learner,

We congratulate you on your decision to pursue an Aptech Worldwide course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

  Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

  The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

  The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

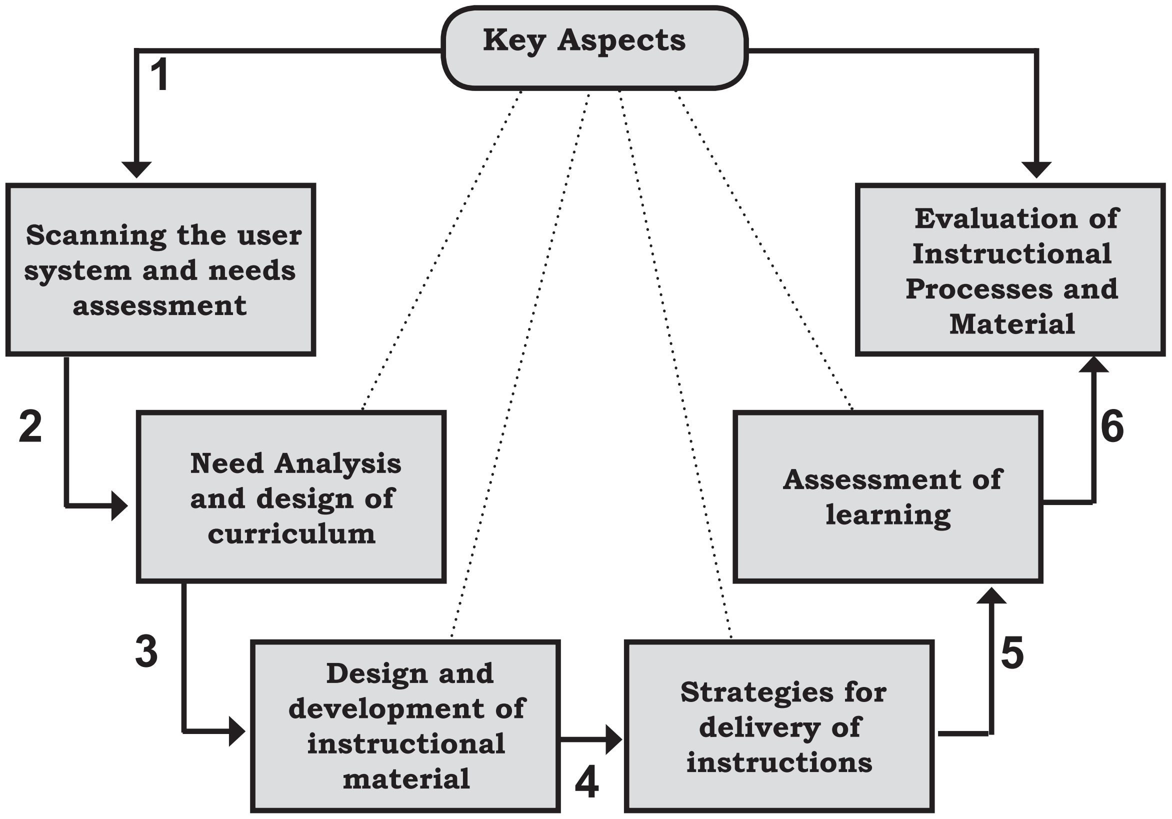
- Evaluation of instructional process and instructional materials

  The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model


Key Aspects
1
Scanning the user system and needs assessment
2
Need Analysis and design of curriculum
3
Design and development of instructional material
4
Strategies for delivery of instructions
5
Assessment of learning
6
Evaluation of Instructional Processes and Material

Preface

The book, Learn Java By Example, aims to teach the basics of Java programming language. The Java programming language was created by Sun Microsystems Inc. which was merged in the year 2010 with Oracle USA Inc. The merged company is now formally known as Oracle America Inc.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions. Please send us your feedback, addressed to the Design Centre at Aptech's corporate office.

Design Team

Table of Contents

Module

Module 1

Introduction to Java

Module Overview

Welcome to the module, **Introduction to Java**. This module introduces object-oriented concepts and the Java programming language. The module begins with a brief explanation of object-oriented concepts. It introduces Java as a platform and as a programming language. It further explains the Java Development Kit (JDK), Java Virtual Machine (JVM), and demonstrates how to write and execute a Java application program. Finally, it provides an overview on writing comments in Java programs.

In this module, you will learn about:

- Classes and Objects
- Getting Started with Java
- Introduction to JDK
- Java Virtual Machine
- Writing a Java program
- NetBeans IDE Overview
- Using Comments in Java

1.1 Classes and Objects

In this first lesson, **Classes and Objects**, you will learn to:

- Describe real-world entities as objects.
- Describe a software object.
- Define a class and describe its structure.
- Compare classes and objects.

1.1.1 Introduction to Classes and Objects

Objects and classes are the bricks used to build Java applications. A class is a template for multiple objects with similar features. Classes represent all the features of a particular set of objects.

1.1.2 Object

An object is the software representation of a real-world entity. For example, objects in a railway reservation system may include a train, a journey destination, an icon on a screen, or a full screen with which a railway booking agent interacts. Any tangible entity in the real-world can be described as an object. Examples of real-world entities exist around everyone, such as a car, bike, flower, or person.

Every entity has some characteristics and is capable of performing certain actions. Characteristics are attributes or features describing the entity, while actions are activities or operations involving the object.

For instance, consider a real-world entity, such as a Toyota or a Ford car. Some of the characteristics associated with a Toyota or Ford car are as follows:

- Color
- Make
- Model

Some of the actions associated with a Toyota or Ford car are as follows:

- Drive
- Accelerate
- Decelerate
- Apply brake

1.1.3 Software Object

The concept of objects in the real-world can be extended to the programming world, where software 'objects' can be defined.

Like its real-world counterpart, a software object has state and behavior. The 'state' of the software object refers to its characteristics or attributes, whereas the 'behavior' of the software object comprises its actions.

For example, a real-world object, such as a Car will have the state as color, make, or model and the behavior as driving, changing gear, increasing speed, and applying brakes.

The software object stores its state in fields (also called variables in programming languages) and exposes its behavior through methods (called functions in programming languages).

Figure 1.1 depicts a 'Car' object.

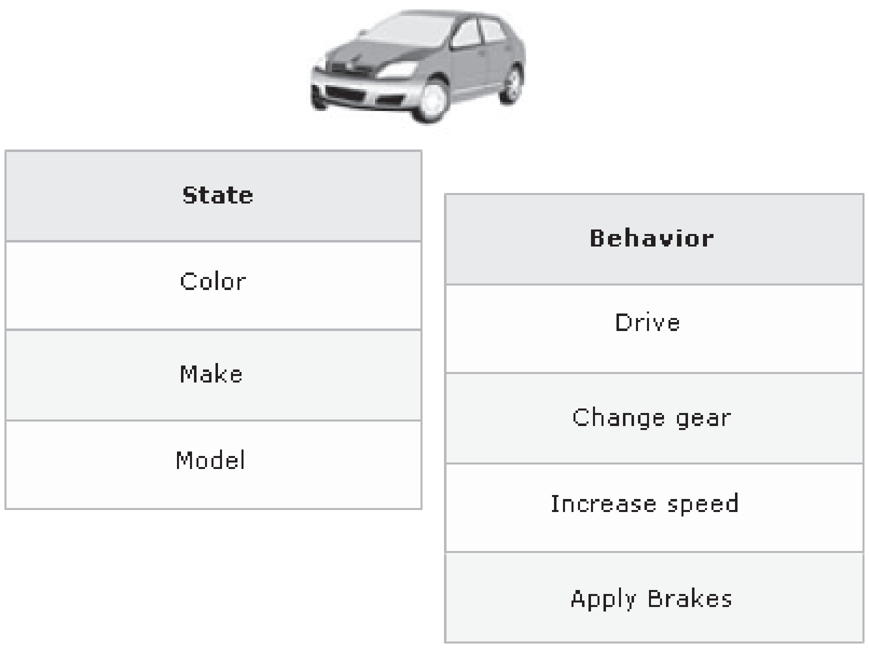


Figure 1.1: A 'Car' Software Object

The advantages of using objects are as follows:

- They help to understand the real world.
- They map attributes and actions of real world entities into state and behavior of software objects.

1.1.4 Defining a Class

In the real-world, several objects having common state and behavior can be grouped under a single class. For example, a car (any car in general) is a class and a Toyota car (a specific car) is an object or instance of the class.

A class is a template or blueprint which defines the state and behavior for all objects belonging to that class.

Typically, in an object-oriented language, a class comprises fields and methods, collectively called as members, which depict the state and behavior respectively.

Figure 1.2 defines a class that represents a conceptual model of a 'Car' entity.

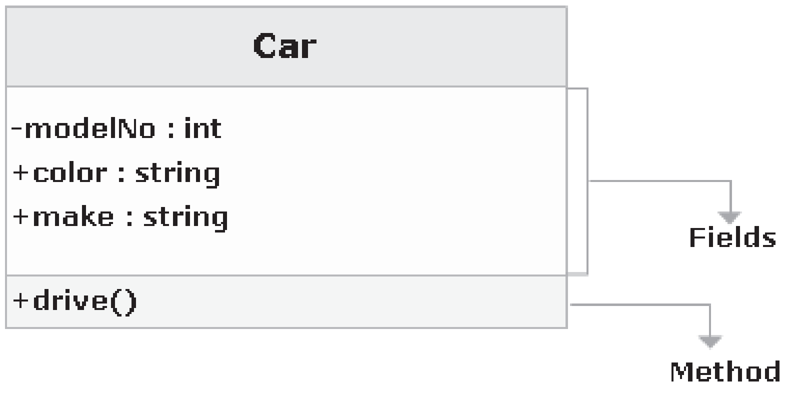


Figure 1.2: Defining Class

As shown in figure 1.2, the conceptual model of a class is universal and is not specific to a particular object. An actual instance of the entity will come into existence when the object is created.

1.1.5 Comparison Between Class and Object

Table 1.1 shows the difference between a class and an object.

Class	Object
Class is a conceptual model	Object is a real thing
Class describes an entity	Object is the actual entity
Class consists of fields (data members) and functions	Object is an instance of a class

Table 1.1: Difference Between a Class and an Object

Figure 1.3 depicts the comparison of a class and an object.

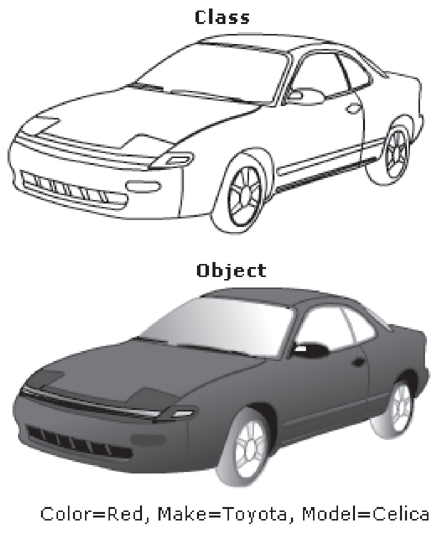


Figure 1.3: Comparison of a Class and an Object

As shown in figure 1.3, each object is said to be an instance of its class.

Knowledge Check 1

1. Can you match the terms against their corresponding description?

Description		Term	
(A)	This represents the behavior of an object	**(1)**	Object
(B)	This is an instance of a class	**(2)**	Class
(C)	This represents the state of an object	**(3)**	Method
(D)	Template or blueprint which defines the state and behavior for all objects	**(4)**	Field
(E)	This describes an entity		

1.2 Getting Started with Java

In this second lesson, **Getting Started with Java**, you will learn to:

- Identify the evolution stages of Java.
- State the components of the Java platform.
- List the features of Java as a programming language.

1.2.1 Introduction to Java

In the year 1995, Sun Microsystems introduced a new programming language to the world - Java. Until then, the word 'Java' could only mean an island in Indonesia or a particular blend of coffee.

Java is a language and a platform that helps professional programmers to develop wide range of applications on various platforms. Java is built upon C and C++. It derives its syntax from C and its features from C++. Java was initially developed to cater to small-scale problems, but was also found capable of addressing large-scale problems across the Internet. Very soon, Java was hugely successful and adopted by millions of programmers all across the world.

1.2.2 Evolution

Java has, over the years, undergone a series of changes and evolved into the robust language that it is today. The different stages of evolution of Java are:

- **Java Origins: Embedded Systems (1991-1994)**

  In 1991, a team of engineers from Sun Microsystems wanted to design a language for electronic devices, such as television, ovens, washing machines, and so on. Thus, the basic objective behind developing the language was to create software that could be embedded in consumer electronic devices. In C and C++ languages the compilers were targeted for a particular CPU.

  Compilers are expensive and a lot of time is required to create these. Hence, it was not possible to have compilers for every CPU. An easy and cost efficient way was needed for creating compilers. In addition, the software had to be small, fast, efficient and platform independent, that is, a code that could run on different CPUs under different environments. Efforts were taken to produce a portable, platform independent language, and the result of this led to the creation of Java. James Gosling and a team of other programmers were the pioneers behind this development. It was initially called 'Oak', but was later renamed to Java.

- **Java: A Client-side Wonder (1995-1997)**

  In 1995, Web pages did not have dynamic capability. Java provided this capability. Later, Java gained popularity and served as ideal software for network computers.

- **Java: Moved into the Middle-tier (1997 to Present)**

  In the late 1990's, Sun revised middle-tier capabilities for Java to ensure that it runs on Web/Application Servers. In 1997, Sun defined Servlets for Java to generate dynamic HTML Web pages. Sun also defined Enterprise JavaBeans (EJB) so that business logic can be developed in Java. In 1999, Sun offered a middle-tier solution for Java called Java 2 Enterprise Edition (J2EE).

- **Future**

  In future, Java may gain more success on the client-side. Improved Java performance and the introduction of Java 2 Micro Edition have opened the door for increased use of Java in embedded devices.

1.2.3 Platform

A platform is a hardware or software environment in which a program runs. Some of the commonly used platforms are: Microsoft Windows, Linux, and Solaris. A number of these platforms, such as Linux and Solaris are a combination of operating system and underlying hardware components.

The Java platform can be considered as an execution engine referred to as virtual engine and not a specific operating system or hardware. The Java platform comprises two essential components:

- **The Java Virtual Machine (JVM)**

  The Java Virtual Machine (JVM) is the Java runtime environment and is available on different operating systems. It serves as the intermediary between a Java program and a host computer. JVM executes compiled Java programs (bytecodes). It forms a layer of abstraction for:

  - Underlying hardware platform
  - Operating system
  - Compiled code

  Different versions of JVMs are available for different operating systems.

➢ **The Java Application Programming Interface (API)**

Java APIs contain vast libraries of classes and other software components, such as interfaces. These are included as a part of the Java SDK. Newer releases of Java APIs provide enhanced features with introduction of new class libraries and packages.

1.2.4 Programming Language

In addition to providing a platform-independent environment, Java is a high-level programming language. It can be used to create applications and applets (a Java program designed to run within Web browsers).

Java as a programming language has the following features:

➢ **Object-oriented**

Java is purely object-oriented language. There are no stand-alone constants, variables, or functions in Java. All of these are part of objects. The constants, variables, or functions in Java are accessed through classes and objects.

Other hybrid object-oriented languages, such as C++, have features of structured languages as well as object extensions. For example, C++ is an object-oriented language, but structured programming constructs, such as `main()` method external to any classes and objects still have to be written, which dilutes the effectiveness of the object-oriented extensions. Java does not allow this way of declaration. In Java the `main()` method is to be declared inside a class.

➢ **Platform-independent**

Platform-independence is the ability of a program to run on any machine regardless of the underlying platform. Java is platform independent at the source level, and the binary level, that is the bytecode level.

In short, platform independence at the source level allows moving the source code from one system to another, compile the code, and run it cleanly on a system.

Using bytecode, Java solves the problem of platform independence. Unlike the C compiler, Java compiler produces a special format known as bytecode, which is same on every platform.

To run Java on a new computer or an operating system, only the interpreter and a few native libraries need to be ported. The work of the interpreter is to read the bytecode and translate it into the native language of the host machine.

- **Robust**

  Java is strongly typed language. It is designed for writing highly reliable or robust software. Hence, it requires explicit method declaration. Java checks the code for syntax error at the time of compilation, and also at the time of interpretation. Thus, it eliminates certain programming errors.

  Java does not have pointers and pointer arithmetic. It checks all access to arrays and strings at runtime, to ensure that they are in bounds. It also checks the casting of objects from one type to another at runtime.

  In traditional programming environments, the programmer had to manually allocate memory. By the end of the program, the programmer had to also explicitly free this memory. Problems arose when the programmer forgot to deallocate the memory. In Java, the programmer does not need to bother about memory deallocation. It is done automatically, as Java provides garbage collection for unused objects.

  Java's exception handling feature simplifies the task of error handling mechanism.

- **Secure**

  Computer viruses are a great cause of worry in the world of computers. Prior to the advent of Java, programmers had to first scan files before downloading and executing them. Often, even this precaution was no guarantee against viruses. Also, there are many malicious programs that can search the contents of your local file system and retrieve sensitive data.

  Java provides a controlled environment for the execution of the program. It never assumes that the code is safe for execution. Since, Java is more than a programming language, it provides several layers of security control.

  In the first layer, the data and methods are accessed only through the interface that the class provides. Java does not allow any pointer arithmetic. Hence, it does not allow direct access to memory. It disallows array overflow, and provides garbage collection. All these features help minimise safety and portability problems.

  In the second layer, the compiler ensures that the code is safe, and follows the protocols set by Java before compiling the code.

  The third layer is the safety provided by the interpreter. The verifier provided by interpreter thoroughly screens the bytecodes to ensure that they obey the rules before executing them.

  The fourth layer takes care of loading the classes. The class loader ensures that the class does not violate the access restrictions, before loading it into the system.

- **Distributed**

  Java is used to develop applications that are portable across multiple platforms, operating systems, and graphical user interfaces. It supports network applications. Hence, Java is widely used as a development tool in an environment like the Internet, which has different platforms.

- **Multi-threaded**

  Java provides support for multi-threading to perform many tasks simultaneously. For example, an application has to carry out a task while waiting for user input. Similarly, in a GUI-based network application, such as a Web browser, there are usually multiple things going on at the same time.

- **Dynamic**

  Java is a dynamic language designed to adapt to an evolving environment. Java source code is divided into `.java` files. The compiler compiles these into .class files containing bytecode. Each `.java` file generally produces exactly one `.class` file.

  The compiler first checks the path containing current directory and other directories specified in the `CLASSPATH` environment variable. This is required to find other classes explicitly referenced by name in each source code file. For example, if the file compiled depends on other non-compiled files, the compiler will try to find them and compile them as well. A compiler can handle circular dependency as well as methods that are used before they are declared. It also can determine whether a source code file has changed since the last time it was compiled. Thus, the compiler is quite smart.

- **Architecture-neutral**

  Java technology is designed to support applications which will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures.

  Java programs are compiled to an architecture neutral bytecode format to transport code efficiently to multiple hardware and software platforms. Hence, the binary distribution problem and the version problem are solved by the interpreted nature of Java technology.

  Any system that implements the Java Virtual Machine allows the Java application to run on that system. This is useful not only for the networks, but also for single system software distribution.

- **Portable**

  The portability actually comes from architecture-neutrality. Java technology takes portability a stage further by being strict in its definition of the basic language. It explicitly specifies the size of its basic data types to eliminate implementation dependence, and the behavior of its arithmetic operators. The Java system itself is quite portable.

The Java compiler is written in Java, while the Java run-time system is written in ANSI C with a clean portability boundary.

➢ **High Performance**

Performance is always a consideration. Compared to those high-level, fully-interpreted scripting languages, Java is high-performance. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability. Sun claims that the performance of bytecodes converted to machine code is nearly as good as native C or C++.

Knowledge Check 2

1. Which of these statements about the components of the Java platform are true and which statements are false?

(A)	Java APIs contain libraries of classes and other software components such as interfaces
(B)	JVM serves as the intermediary between a Java program and a host computer
(C)	JVM executes compiled Java programs also called as bytecodes
(D)	The Java SDK does not include APIs
(E)	JVM is not available on different operating systems

2. Can you match the features of Java against their corresponding descriptions?

Description		**Feature**	
(A)	Provides security control	**(1)**	Platform-independent
(B)	Supports the development of applications across multiple platforms, operating systems and graphic user interfaces	**(2)**	Object-oriented
(C)	All constants, variables, or functions must exist within classes or objects	**(3)**	Distributed
(D)	Java provides support for performing many tasks simultaneously	**(4)**	Secure
(E)	Can run on any machine	**(5)**	Multithreading

1.3 Introduction to JDK

In this third lesson, **Introduction to JDK**, you will learn to:

- Explain JDK and its tools.
- Explain JVM.
- Configure JDK.

1.3.1 Java Development Kit

The Java Development Kit (JDK) is provided by Sun Microsystems and is popularly used by Java programmers worldwide. The purpose of JDK is to provide the software and tools required for compiling, debugging, and executing Java applications. The JDK software and documentation are freely available at Sun's official Web site.

Java Standard Edition, popularly called Java SE, is a technology and platform that provides support to build applications that have high functionality, speed, and reliability. Java SE Development Kit, popularly called JDK, includes the necessary development tools, runtime environment, and APIs for creating Java programs with the Java platform.

JDK includes two important tools:

- **javac (compiler)**

  The `javac` compiler is used to compile Java source code into bytecodes. The source code can be created using an Integrated Development Environment, such as NetBeans or a simple text editor, such as Notepad.

  Syntax:

  ```
  javac [option] source
  ```

  where,

  `source` is one or more file names that end with the extension `.java`.

  For example,

  ```
  javac FirstProgram.java
  ```

  This command will produce a file named '**`FirstProgram.class`**'. This class file is what runs on the Java Virtual Machine.

Table 1.2 lists some of the options that can be supplied for the `javac` compiler.

Option	Description
-classpath	Specifies the location for import classes (overrides the CLASSPATH environment variable).
-d	Specifies the destination directory for class files.
-g	Prints all debugging information instead of the default line number and file name.
-verbose	Produces additional output about each class loaded and each source file compiled.
-version	Displays version information.
-sourcepath	Imports a class location.
-help	Prints a synopsis of standard options.

Table 1.2: Options for javac Compiler

Consider an example using an option `-d`.

```
javac -d c:\ FirstProgram.java
```

This command will produce and save '**`FirstProgram.class`**' file in C drive.

➢ **java (interpreter)**

The `java` interpreter is used to interpret and run Java bytecodes. It takes the name of a class file as an argument for execution.

Syntax:

```
java [option] classname [arguments]
```

where,

`classname` is the name of the class file.

`arguments` is the arguments passed to the main function.

For example,

```
java FirstProgram
```

Table 1.3 lists some of the options that can be supplied for the `java` interpreter.

Options	Description
classpath	Specifies the location for import classes (overrides the CLASSPATH environment variable).
-v or -verbose	Produces additional output about each class loaded and each source file compiled.
-version	Displays version information and exits.
-jar	Uses a JAR file name instead of a class name.
-help	Displays usage information and exits.
-X	Displays information about non-standard options and exits.

Table1.3: Options for java Interpreter

1.3.2 Java Virtual Machine

The Java Virtual Machine (JVM) is at the heart of the Java programming language. The Java environment consists of five elements:

- Java language
- Bytecode definitions
- Java/Sun libraries
- Java Virtual Machine
- Structure of .class files

The portability feature of the `.class` file has made the principle of Java '**write once, and run anywhere**' possible. This file helps the execution on any computer or chip set, with an implementation of the JVM.

The virtual machine is a software concept based on the idea of an imaginary computer. It has a logical set of instructions. These instructions define the operations of this imaginary computer. It can be thought as a mini operating system. It forms a layer of abstraction for the underlying hardware platform, the operating system, and the compiled code.

The compiler converts the source code into a code based on the imaginary computer's instruction set that is not targeted to a particular processor. An interpreter is an application that understands these streams of instructions. It converts these instructions for the underlying hardware to which the interpreter is targeted.

The JVM internally creates a runtime system that helps in execution of code through the following steps:

- **Loading the .class files**

  Class loaders are one of the fundamentals of the JVM architecture. They enable the JVM to load classes without knowing anything about the file system semantics, and they allow applications to dynamically load Java classes as extension modules.

- **Managing the memory**

  The JVM manages memory in the following way:

  - When a JVM is invoked to run an application, it requests the operating system for enough memory for the JVM itself for running and free memory for the application to create new objects.
  - When a new object is created, the JVM allocates memory for that object from the free memory area.
  - When the free memory area is reduced after several object creations, the JVM asks the operating system for more.
  - When an object is no longer used, it will be destroyed. The memory occupied by it will be freed up and it will be merged back to the free memory area.
  - When the free memory area is occupied, and no more additional memory is available from the operating system, the JVM will halt the application that created the situation, and will raise 'Out of memory error'.

- **Performing the garbage collection**

  The garbage collector releases the memory occupied by an object once it determines that the object is no longer accessible. This automatic process makes it safe to throw away unnecessary object references because the garbage collector does not collect the object if it is still needed elsewhere. Therefore, in Java the act of releasing unnecessary references never runs the risk of deallocating memory prematurely.

The other popular concept in Java is the use of the 'Just-In-Time' (JIT) compiler. Browsers, such as Netscape Navigator and Internet Explorer include JIT compilers that increase the speed of Java code execution. JIT's main purpose is to convert the bytecode instruction set to machine code instructions targeted for a particular microprocessor. These instructions are stored and used whenever a call is made to that particular method.

Figure 1.4 shows the relationship between the Java compiler and JIT compiler.

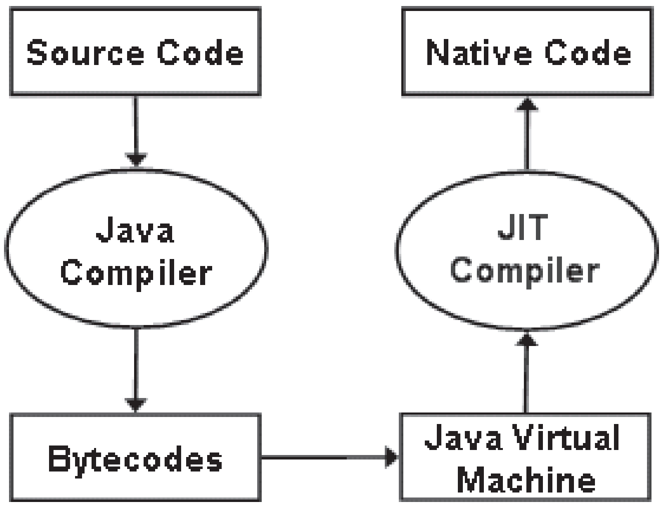


Figure 1.4: Relationship between Java Compiler and JIT Compiler

1.3.3 Configuring JDK

To work with the JDK and Java programs, certain settings need to be made to the environment variables. Environment variables are pointers pointing to programs or other resources. The variables to be configured are as follows:

➢ **PATH**

The `PATH` variable is set to point to the location of Java executables (`javac.exe`, `java.exe`). This is necessary so that the `javac` and `java` command can be executed from any directory without having to type the full path.

Setting path in DOS:

```
C:\>set path=<drivename>:\<installation_folder>\bin
```

Setting path in Windows 2000/XP:

Right-click the **My computer** icon present on the desktop. Click **properties**. Click **Advanced tab**. Click **Environment variables**. In the System variable area, select the **PATH** variable. Click **Edit** and type the path of the `bin` folder of jdk1.5.

- **CLASSPATH**

  `CLASSPATH` is an environment variable that specifies the location of the class files and libraries needed for the Java compiler (`javac`) to compile applications.

  Setting CLASSPATH in DOS:

```
C:\>set CLASSPATH=<drivename>:\<installation_folder>
```

  Setting CLASSPATH in Windows 2000/XP:

  Right-click **My Computer** icon present on the desktop. Click **properties**. Click **Advanced tab**. Click **Environment variables**, and then in System variables area click **New**. Type `CLASSPATH` in **Variable Name** and then type `C:\jdk1.5.0` as value. Click **OK**.

1.4 Writing a Java Program

In this fourth lesson, **Writing a Java program**, you will learn to:

- Identify the steps in writing a Java program.
- Describe how to compile and execute a Java program.

1.4.1 Steps

A step by step procedure to create a simple Java program is now discussed here. The program will display a message, 'Welcome to the world of Java'.

- **Step 1: Declare a class**

```
class <class_name> {
}
```

  where,

  `class` is a keyword and `<classname>` is the class name.

  Opening and closing curly braces tell the compiler where the class will begin and end. The area between the braces is known as the class body and contains the code for that class.

For instance,

```
class HelloWorld {
}
```

`HelloWorld` is the identifier depicting the name of the class.

- **Step 2: Write the main method**

The `main()` method is the entry point for an application.

Syntax:

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

where,

`public` keyword enables the JVM to access the main method.

`static` keyword allows a method to be called from outside a class without creating an instance of the class.

`void` keyword tells the compiler that the method will not return any value.

`args` is an array of type String and stores command line arguments.

- **Step 3: Write desired functionality**

In this step, actionable statements are written within the methods. The current example illustrates a string can be displayed as the output.

```
System.out.println("Welcome to the world of Java");
```

`System.out.println()` statement displays the string that is passed as an argument.

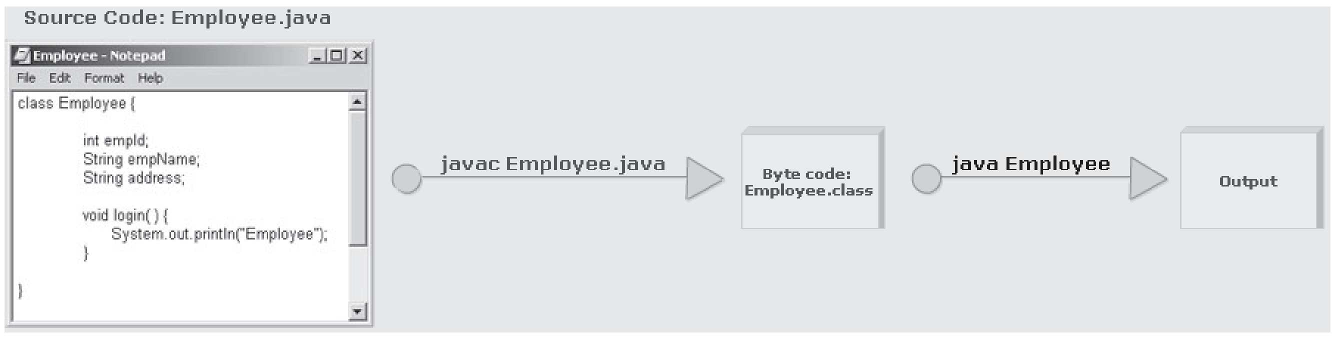
- **Step 4: Save the program**

Save the file with a `.java` extension. The name of the file plays a very important role in Java. A `.java` extension is a must for a Java program. In Java, the code must reside in a class, and hence the class name and the file name would be the same most of the times.

1.4.2 Compile and Execute

The Java compiler, `javac.exe`, compiles a Java program by checking the syntax, grammar, and the data types used in it. Compilation leads to creation of a file with a `.class` extension. The file contains the bytecode that is interpreted and converted into machine code before execution.

To actually run the program, the Java interpreter, `java.exe`, is required, which will interpret and run the code. Figure 1.5 demonstrates the compilation and execution steps of the Java program.



```
class Employee {

        int empId;
        String empName;
        String address;

        void login() {
            System.out.println("Employee");
        }

}
```

Figure 1.5: Compilation and Execution

Knowledge Check 4

1. Which of these statements about compiling and executing a Java program are true and which statements are false?

(A)	The `javac` tool invokes the Java compiler
(B)	The `java` tool invokes the Java interpreter
(C)	The Java interpreter checks for the syntax, grammar, and data types of the program
(D)	The `.class` files contain bytecodes
(E)	The Java interpreter compiles the code

1.5 NetBeans IDE Overview

In this fifth lesson, **NetBeans IDE Overview**, you will learn to:

- Identify the benefits of NetBeans Integrated Development Environment (IDE).
- Identify the elements of NetBeans IDE.
- List the steps to create a Java program using NetBeans IDE.

1.5.1 NetBeans IDE

NetBeans is an open-source integrated development environment written purely in Java. It is a free and robust IDE that helps developers to create cross-platform desktop, Web, and mobile applications using Java.

Benefits of using NetBeans IDE as compared to Notepad are as follows:

- The coding is completed faster with features such as code completions, code template, and fix import.
- Debugging can be done in the source editor.

Benefits of NetBeans IDE:

- Builds IDE plug-in modules and supports rich client applications on the NetBeans platform.
- Provides graphical user interface for building, compiling, debugging, and packaging of applications.
- Provides simple and user-friendly IDE configuration.

1.5.2 Elements

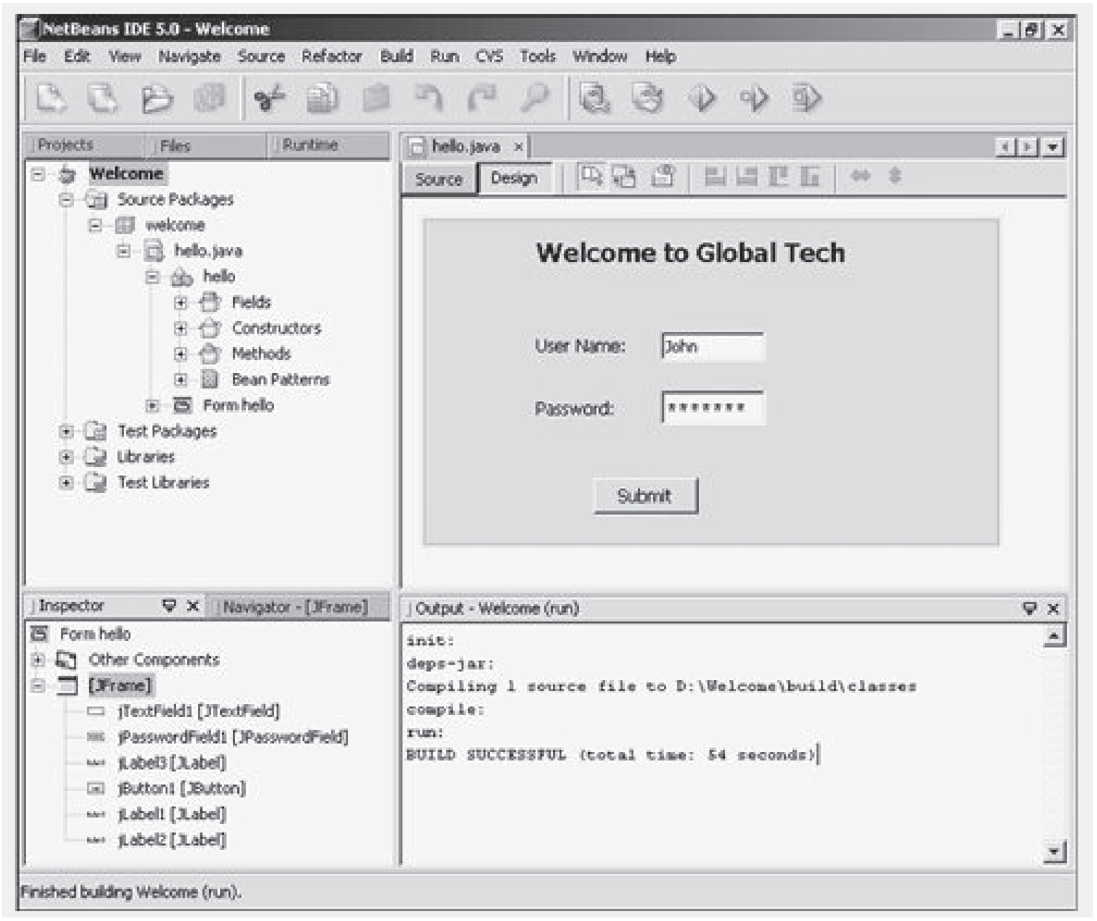
The NetBeans IDE has the following elements and views:

- Menu Bar
- Folders View
- Components View

➢ Coding and Design View

➢ Output View

Figure 1.6 shows the various elements in the NetBeans IDE.



User Name: John

Password: *******

```
init:
deps-jar:
Compiling 1 source file to D:\Welcome\build\classes
compile:
run:
BUILD SUCCESSFUL (total time: 54 seconds)
```

Finished building Welcome (run).

Figure 1.6: Elements in NetBeans IDE

Each of these elements will be explained in detail in the following pages.

Concepts

1.5.3 Menu Bar

The image shows the main menu of NetBeans IDE. Each menu item has several sub-menu items, each providing a unique functionality. Figure 1.7 shows the menu bar in the NetBeans IDE.

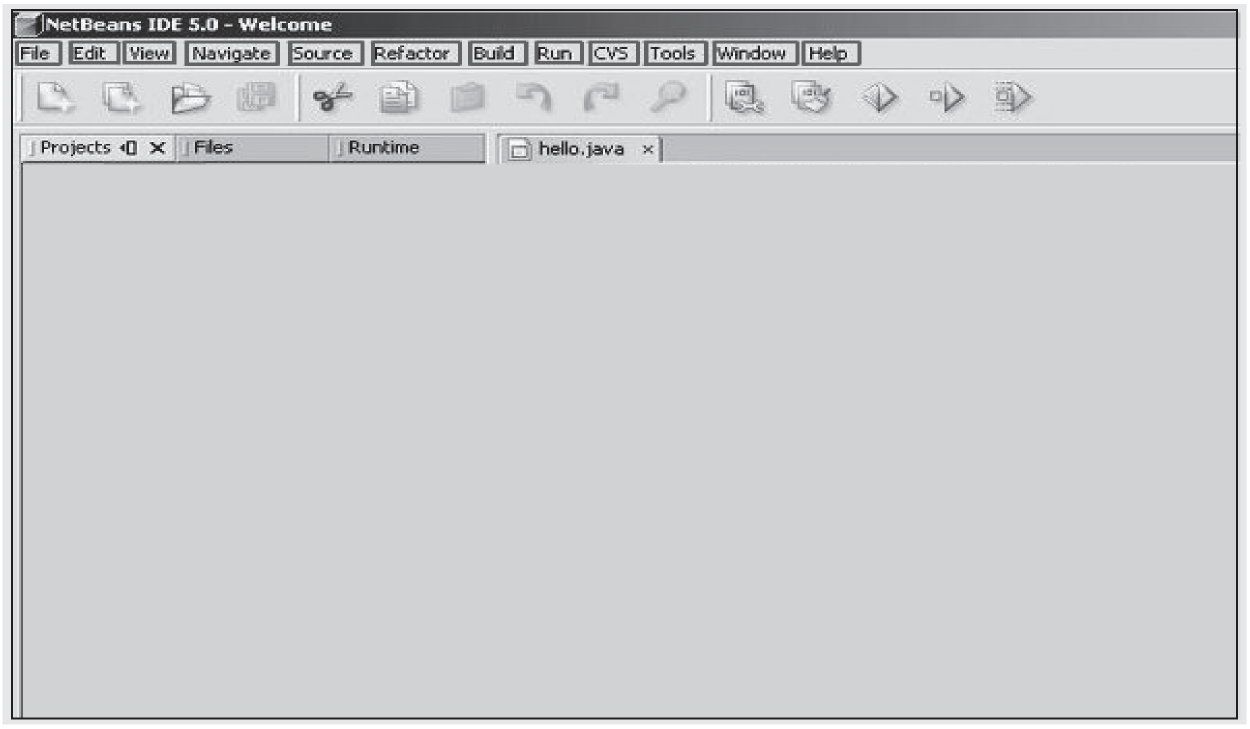


Figure 1.7: Menu Bar in NetBeans IDE

Explanation for highlighted areas in the image:

File

The File menu displays commands for using the project and the NetBeans IDE.

Edit

The Edit menu provides standard commands for editing.

View

The View menu provides options to display the document in different views.

Navigate

The Navigate menu provides options for navigation in the project.

Source

The Source menu provides options for overriding methods, fixing imports, and inserting try-catch blocks.

Refactor

The Refactor menu allows you to change the structure of the code and reflect the changes made in the code.

Build

The Build menu builds, cleans, and compiles the applications. It also provides an option for generating `Javadoc`.

Run

The Run menu executes the application, inserts breakpoints, watches, and attaches the debugger.

CVS

The Concurrent Version System (CVS) menu enables version controlling of the files.

Tools

The Tools menu provides utilities to search through the `Javadoc` index and also to insert internationalized string. It also provides different tool managers.

Window

The Window menu provides options to select or close any window. It also provides debugging options.

Help

The Help menu provides guidance on how to use NetBeans IDE effectively.

1.5.4 Folders View

Figure 1.8 shows a pane in the NetBeans IDE featuring:

➢ Project window

➢ Files window

➢ Runtime window

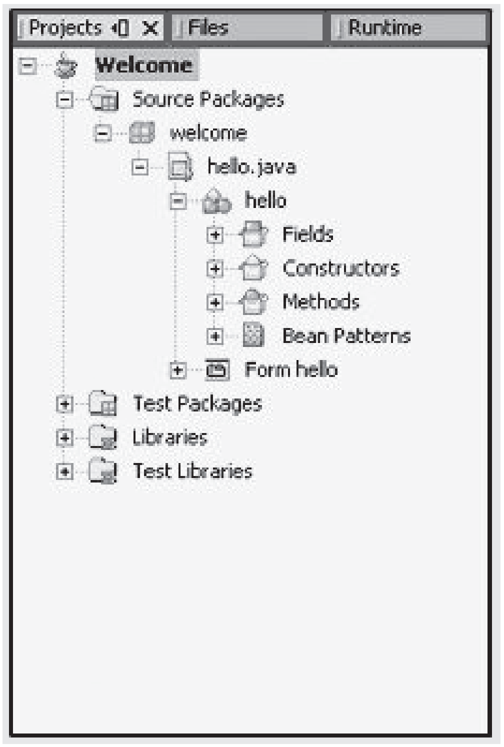


Figure 1.8: Folders View

Explanation for highlighted areas in the image:

➢ **Projects Window**

The Project window displays the project content, such as Source Packages, Test Packages, Libraries, and Test Libraries.

- Source Packages folder contains the Java source code for the project.
- Test Packages is a default folder provided to store the source code for testing the application.

- Libraries comprise resources required by the project, such as source files and Javadoc files.
- Test Libraries configure the test classpath.

➢ **Files Window**

The Files window shows the directory structure of all files and folders present in the project. You can open, edit, compile, build, and run the files from the Files window. Also, fields, constructors, methods, and bean patterns can be added using the Files window.

➢ **Runtime Window**

The Runtime window displays runtime information on currently running processes, registered servers, Web services, xml catalog, and database drivers added to the IDE and debugging sessions.

1.5.5 Components View

Figure 1.9 shows the Inspector window and Navigator window used for viewing components in the NetBeans IDE.

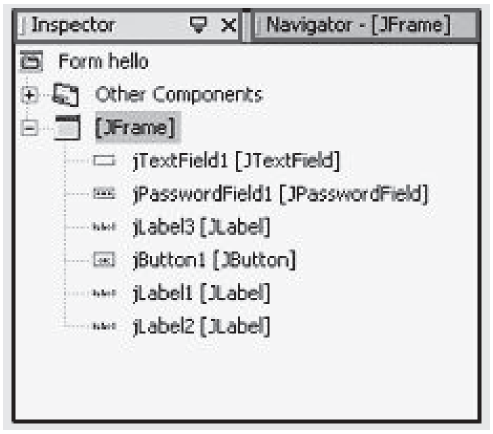


Figure 1.9: Components View

Explanation for highlighted areas in the image:

➢ **Inspector**

Displays a tree hierarchy of all components contained in the currently opened form.

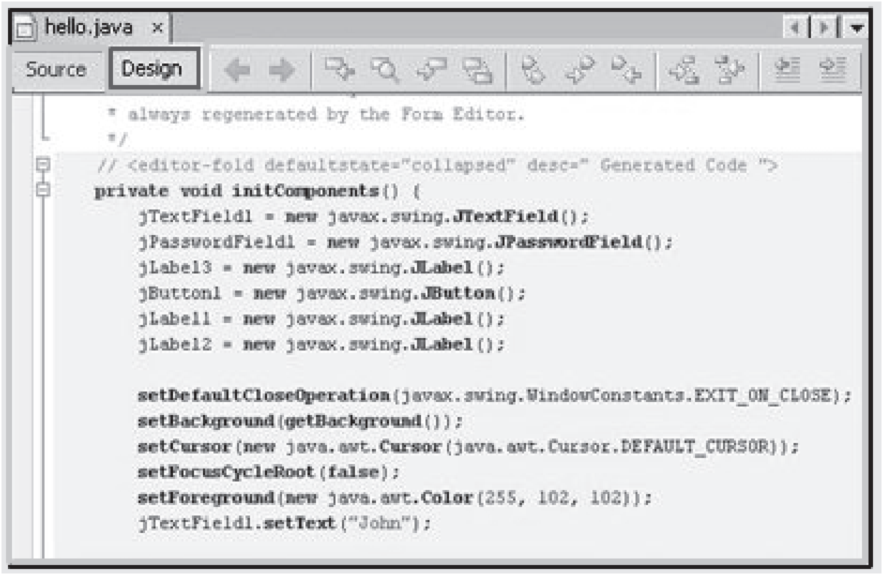
- **Navigator Window**

  The Navigator window serves as a tool that displays details of the source files of the currently opened project. The elements used in the source are displayed here in the form of a list or an inheritance tree.

1.5.6 Coding and Design View

Two elements of the coding and designing area in the NetBeans IDE are namely, Source Editor and Design Form.

Figure 1.10 shows the Source Editor used for viewing the code.



```
      * always regenerated by the Form Editor.
      */
    // <editor-fold defaultstate="collapsed" desc=" Generated Code ">
    private void initComponents() {
        jTextField1 = new javax.swing.JTextField();
        jPasswordField1 = new javax.swing.JPasswordField();
        jLabel3 = new javax.swing.JLabel();
        jButton1 = new javax.swing.JButton();
        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setBackground(getBackground());
        setCursor(new java.awt.Cursor(java.awt.Cursor.DEFAULT_CURSOR));
        setFocusCycleRoot(false);
        setForeground(new java.awt.Color(255, 102, 102));
        jTextField1.setText("John");
```

Figure 1.10: Source Editor

Figure 1.11 shows the Design Form with user interface controls.

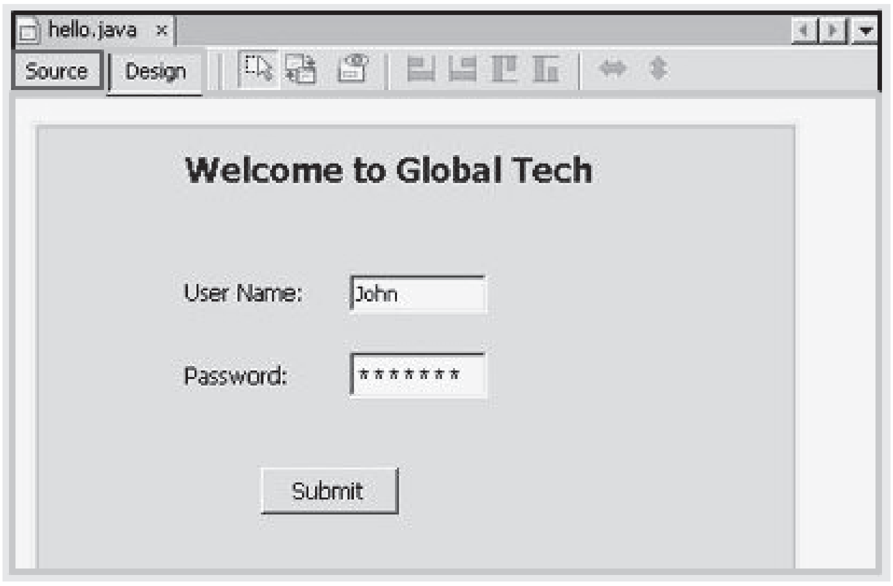


Figure 1.11: Design Form

Explanation for highlighted areas in the image:

- **Source Editor**

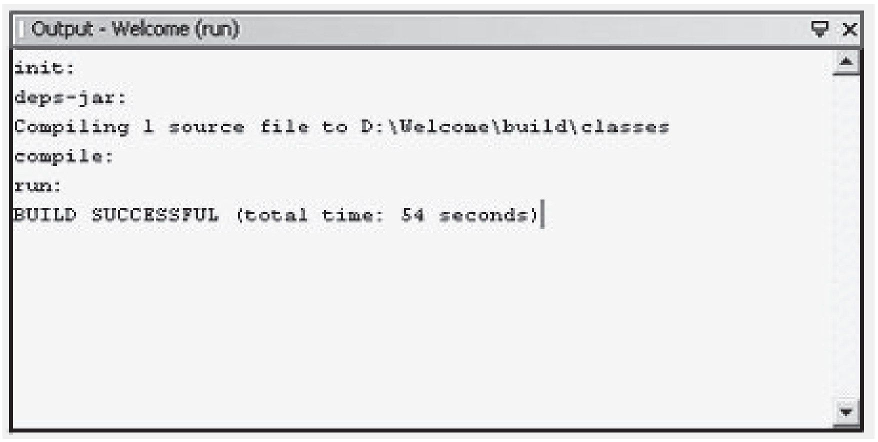
  The Source Editor is a text editor of NetBeans IDE. With Source Editor, the user can create, edit, and view the source code. It also directs the user in adding the missing code.

- **Design**

  Helps in designing the form and dragging controls on to the form, so as to have a view of the interface.

1.5.7 Output View

The last element of the NetBeans IDE under study is the output view, which is exemplified through the image. Figure 1.12 shows the output view which displays the status of compilation and execution of the application.



```
init:
deps-jar:
Compiling 1 source file to D:\Welcome\build\classes
compile:
run:
BUILD SUCCESSFUL (total time: 54 seconds)
```

Figure 1.12: Output View

➢ **Output window**

The Output window displays messages from the NetBeans IDE. It shows compilation errors, debugging messages, and `Javadoc` comments. The output of the program is also displayed in the Output window.

1.5.8 Creating a Java Program Using NetBeans IDE

A project in NetBeans sets up a framework for the user to write, compile, test, and debug applications. It has folders containing Source Packages, Test Packages, Libraries, and Test Libraries.

➢ **Steps to create a Java Program using NetBeans IDE**

1. **Create a new project**

   Select New Project from the File menu to create a new project.

2. **Specify Name and Location for the project**

   In the New Project window, select General as the category and Java Application as the Project. Click **Next** and specify the Name and Location of the application.

3. **Select Set as Main Project and Create Main Class**

   After specifying the Name and Location, select Set as Main Project and Create Main Class check boxes.

4. **Write the code**

   Write the appropriate code.

5. **Save the project**

   Select Save from the File menu to save the project.

➢ **Building and executing Java Program in NetBeans**

6. **Build Main Project**

   Select Build Main Project from the Build menu to build the project.

7. **Run Main Project**

   Select Run Main Project from the Run menu to run the project.

The output is displayed in the output window.

Knowledge Check 5

1. Can you match the elements of NetBeans IDE against their corresponding descriptions?

	Description		Element
(A)	Shows compilation errors, debugging messages, and `Javadoc` comments	(1)	Runtime window
(B)	Directs the user in adding the missing code	(2)	Properties window
(C)	Displays information on currently running processes and registered servers	(3)	Source Editor
(D)	Consists of Source packages, Test Packages, Libraries, and Test Libraries	(4)	Output window
(E)	Lists the properties of the control selected	(5)	Project window

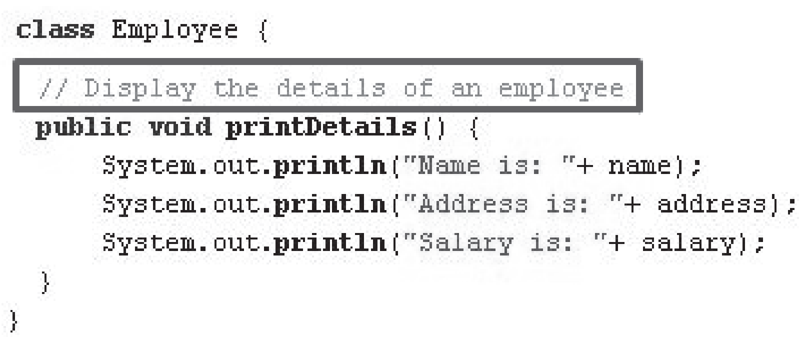
1.6 Using Comments in Java

In this last lesson, **Using Comments in Java**, you will learn to:

- State the syntax of single-line comments.
- State the syntax of multi-line comments.
- State the syntax of `Javadoc` comments.

1.6.1 Single-line Comments

Comments help easy understanding of code. A single-line comment is used to document the functionality of a single line of code. Figure 1.13 shows the Java program documented with single-line comments.



```
class Employee {
  // Display the details of an employee
  public void printDetails() {
       System.out.println("Name is: "+ name);
       System.out.println("Address is: "+ address);
       System.out.println("Salary is: "+ salary);
  }
}
```

Figure 1.13: Single-line Comments

There are two ways of using single-line comments:

- **Beginning-of-line comment**

  This type of comment can be placed before the code (on a different line).

- **End-of-line comment**

  This type of comment is placed at the end of the code (on the same line).

Conventions for using single-line comments are as follows:

- Provide a space after the forward slashes.
- Capitalize the first letter of the first word.

The following is the syntax for applying the comments.

Syntax:

```
// Comment text
```

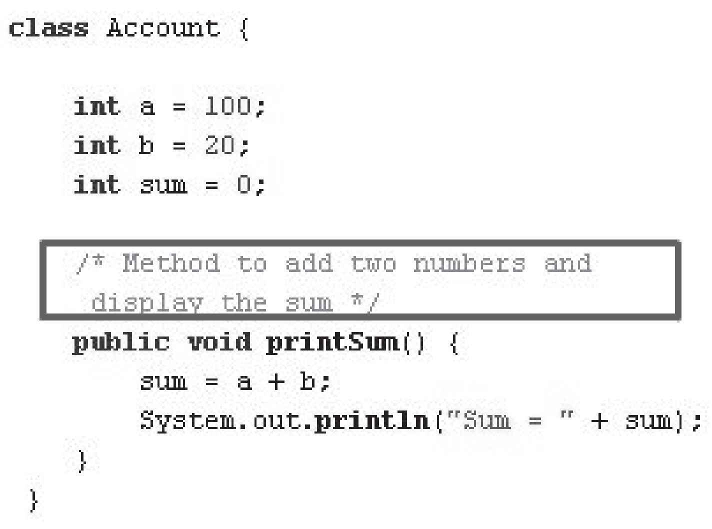
The following code shows the different ways of using single-line comments in the Java program.

Code Snippet:

```
// Declaring a variable
int a = 32;
int b = 64; // Declaring a variable
```

1.6.2 Multi-line Comments

A multi-line comment is a comment that spans multiple lines. A multi-line comment starts with a forward slash and an asterisk (`/*`). It ends with an asterisk and a forward slash (`*/`). Anything that appears between these delimiters is considered to be a comment. Figure 1.14 shows the Java program documented with multi-line comments.



```
class Account {

    int a = 100;
    int b = 20;
    int sum = 0;

    /* Method to add two numbers and
     display the sum */
    public void printSum() {
        sum = a + b;
        System.out.println("Sum = " + sum);
    }
}
```

Figure 1.14: Multi-line Comments

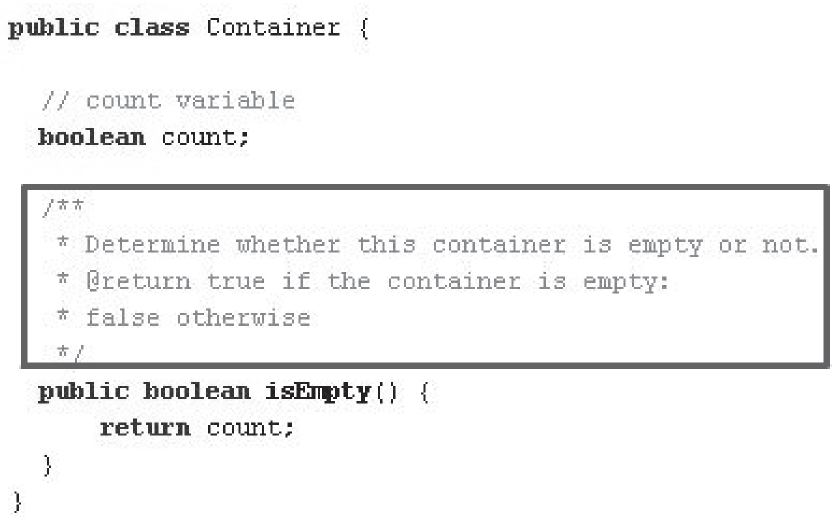
The following code shows a Java program using multi-line comments.

Code Snippet:

```
/*
    This code performs mathematical
    operation of adding two numbers.
*/
int a = 20;
int b = 30;
int c;
c = a + b;
```

1.6.3 Javadoc Comments

A Javadoc comment is used to document public or protected classes, attributes, and methods. It starts `/**` and ends with `*/`. Everything between the delimiters is a comment, even if it spans multiple lines. The `javadoc` command can be used for generating Javadoc comments. Figure 1.15 shows the Java program with Javadoc comment.



```
public class Container {

  // count variable
  boolean count;

  /**
   * Determine whether this container is empty or not.
   * @return true if the container is empty:
   * false otherwise
   */
  public boolean isEmpty() {
      return count;
  }
}
```

Figure 1.15: Javadoc Comments

The following code shows the Javadoc comment specified on the method in the Java program.

Code Snippet:

```
/**
 * This program prints a welcome message
 * using the println statement.
 */
System.out.println("Welcome to Java Programming");
```

Knowledge Check 6

1. Which of these statements about comments are true and which statements are false?

(A)	A multi-line comment starts with a forward slash and a hash (/#)
(B)	A single-line comment is used to document the functionality of a single line of code
(C)	A Javadoc comment ends with double asterisk and a forward slash (**/)
(D)	A multi-line comment can span several lines
(E)	Comment appears within delimiters

Module Summary

In this module, **Introduction to Java**, you learnt about:

➢ **Classes and Objects**

Any tangible entity in the real-world can be represented as an object. Like real-world objects, a software object also has state and behavior. A class is a template or blueprint which defines the state and behavior for all objects belonging to that class. The class is a conceptual model, whereas the object is a real entity.

➢ **Introduction to Java**

Improved Java performance and the introduction of Java 2 Micro Edition have opened the door for increased use of Java in embedded devices. The Java platform comprises two essential components: the JVM and Java API. Basic features of the Java language are that, it is object-oriented and portable.

➢ **Introduction to JDK**

JDK aims to provide the software and tools required for compiling, debugging, and executing Java applets and applications. To work with the JDK and Java programs, the environment variables to be configured are `PATH` and `CLASSPATH`.

➢ **Writing a Java program**

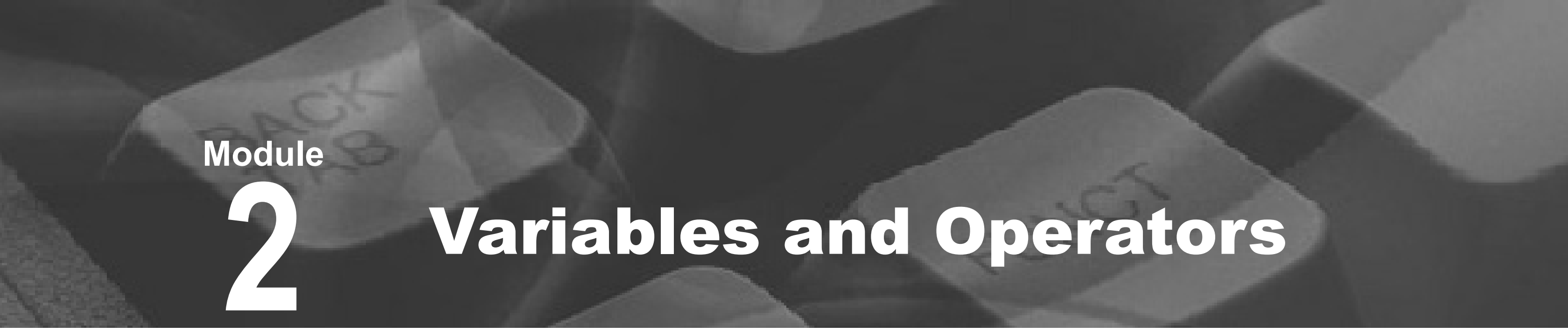
A Java program (source code) can be created using Notepad. It can be compiled using the Java compiler, `javac.exe`, and executed by using the Java interpreter, `java.exe`.

➢ **NetBeans IDE**

NetBeans is an open-source integrated development environment written purely in Java. The main elements of NetBeans IDE are: Project window, Source Editor, and the Compiler. Using NetBeans IDE, a user can write, compile, test, and debug Java applications.

➢ **Using Comments in Java**

The different types of comments in Java, such as single-line and multi-line comments help in easy understanding of the code. The Javadoc comments also allow you to document public or protected classes, attributes, and methods.



Module

2 Variables and Operators

Module Overview

Welcome to the module, **Variables and Operators**. This module focuses on the usage of variables, data types, and operators in Java programs. This module aims at providing a clear understanding of the different data types available in Java. It also, discusses the different operators and their implementation in Java programs. The implicit and explicit data conversion techniques are also covered in this module.

In this module, you will learn about:

- Introduction to Variables
- Introduction to Data Types
- Formatted Output and Input
- Operators
- Type Casting

2.1 Introduction to Variables

In this first lesson, **Introduction to Variables**, you will learn to:

- Explain variables and their purpose.
- Explain the rules and conventions for naming variables.
- Explain declaring literals.

2.1.1 Variables

A variable is a location in the computer's memory where a value is stored and from which the value can be retrieved later.

Variables are used in a Java program to store data that changes during the execution of the program. They are the basic units of storage in a Java program. Variables can be declared to store values, such as names, addresses, and salary details. Variables must be declared, before they can be used.

Figure 2.1 shows the storage of variable in memory.

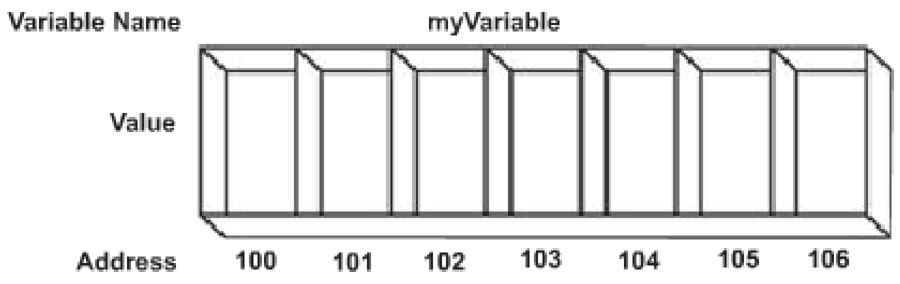


Figure 2.1: Variable Storage in Memory

Syntax:

```
datatype variableName;
```

where,

`datatype` is a valid data type in Java.

`variableName` is a valid variable name.

The following code shows the declaration of variables.

Code Snippet:

```
int rollNumber;
char gender;
```

The statements declare an integer variable called **`rollNumber`**, and a character variable called **`gender`**. These statements instruct the operating system to allocate the required amount of memory to each variable in order to hold the type of data specified for each. Additionally, each statement also provides a name that can be used within the program to access the values stored in each of the variables.

Values can be assigned to variables by using the assignment operator (=) as follows:

```
rollNumber = 101;
```

```
gender = ‘M’;
```

101 and ‘M’ are called literals. You can also assign a value to a variable upon creation, as shown:

```
int rollNumber = 101;
```

The following code shows the different ways for declaring and initializing variables.

Code Snippet:

```
Line 0: int x, y, z; // declares three integer variables x, y and z.
Line 1: int a = 5, b, c = 10; // declares three integer variables, initializes
a and c.
Line 2: byte num = 20; // declares a byte variable num and initializes its
value to 20.
Line 3: char c = ‘c’; // declares the character variable c with value ‘c’.
Line 4: int num1 = num2 = 10; // value 10 is stored in num1 and num2.
```

As shown in the code, `Line 0` and `Line 1` are examples of comma separated list of variables. `Line 4` displays an example of same value assigned to more than one variable at the time of declaration.

2.1.2 Rules and Conventions

Java programming language has its own set of rules and conventions that need to be followed for naming variables. For a start, variable names should be short and meaningful.

Not adhering to the rules will result in syntax errors. Naming conventions are to be followed as they ensure good programming practices.

➢ **Naming Conventions**

The rules and conventions for naming variables are as follows:

- Variable names may consist of Unicode letters and digits, underscore (_), and dollar sign ($).
- A variable’s name must begin with a letter, the dollar sign ($), or the underscore character (_). The convention, however, is to always begin your variable names with a letter, not “$” or “_”.
- Variable names must not be a keyword or reserved word in Java.
- Variable names in Java are case-sensitive (for example, the variable names **number** and **Number** refer to two different variables).

- If a variable name comprises a single word, the name should be in lowercase (for example, velocity or ratio).

- If the variable name consists of more than one word, the first letter of each subsequent word should be capitalized (for example, employeeNumber and accountBalance).

2.1.3 Variable Names

The variable names in Java should start with a letter (A-Z or a-z), dollar sign ($), or underscore (_).

Table 2.1 lists some examples of valid and invalid Java variable names.

Variable name	Valid / Invalid
rollNumber	Valid
a2x5_w7t3	Valid
$yearly_salary	Valid
_2010_tax	Valid
$$_	Valid
amount#Balance	Invalid and contains the illegal character #
double	Invalid and is a keyword
4short	Invalid and the first character is a digit

Table 2.1: Valid and Invalid Java Variable Names

2.1.4 Declaring Literals

A literal signifies a fixed value and is represented directly in the code without requiring computation. For example, the following code shows assigning of literals in the Java program.

Code Snippet:

```
Line 1: int val = 50;
Line 2: float num = 35.7F;
Line 3: char x = 'x';
```

A literal is used wherever a value of its type is allowed. However, there are several different types of literals. Some of them are as follows:

➢ **Integer Literals**

Integer literals are used to represent an `int` value, which in Java is a 32-bit integer value.

In a program, integers are probably the most commonly used type. Any whole number value is an integer literal.

Integers can be expressed as:

- Decimal values, expressed in base 10
- Octal values, expressed in base 8
- Hexadecimal values, expressed in base 16

Each of these have their own literal. An integer literal can also be assigned to other integer types, such as byte or long. When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type. Integer numbers can be represented with an optional uppercase character ('L') or lowercase character ('l') at the end, which tells the computer to treat that number as a long (64-bit) integer.

➢ **Floating-point Literals**

Floating-point literals represent decimal values with a fractional component. Floating point literals have several parts.

- Whole number component, for example 0, 1, 2,....., 9.
- Decimal point, for example 4.90, 3.141, and so on.
- Exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. For example, e+208, 7.436E6, 23763E-05, and so on.
- Type suffix D, d, F, or f.

Floating-point literals in Java default to double precision. A float literal is represented by F or f appended to the value, and a double literal is represented by D or d.

➢ **Boolean Literals**

Boolean literals are simple and have only two logical values - **true** and **false**. These values do not convert into any numerical representation. A true boolean literal in Java is not equal to one, nor does the false literal equals to zero. They can only be assigned to variables declared as boolean, or used in expressions with Boolean operators.

➢ **Character Literals**

Character literals are enclosed in single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as ‘g’, ‘$’, and ‘z’. Single characters that cannot be directly entered are hyphen (-) and backslash (\).

➢ **Null Literals**

When an object is created, a certain amount of memory is allocated for that object. The starting address of this memory is stored in an object, that is, a reference variable. However, at times, it is not desirable for the reference variable to refer that object. In such a case, the reference variable is assigned the literal value null as shown in the following example:

```
obj = null;
```

➢ **String Literals**

String literals consist of sequence of characters enclosed in double quotes. The characters can be printable or non-printable. However, backslash, double quote and non-printable characters are represented using escape sequences. Following is an example of String literal:

```
"Welcome to Java"
```

Knowledge Check 1

1. Which of these statements about variables are true and which statements are false?

(A)	Variables need not be initialized when declared
(B)	Variable names `$bank_account` and `_1account_balance` are valid
(C)	Only one variable can be declared in a statement at a time
(D)	Variables can be declared to store any type of values
(E)	Variable names must begin with a letter, the dollar sign ($), or the underscore character (_)

2.2 Introduction to Data Types

In this second lesson, **Introduction to Data Types**, you will learn to:

- Identify the different data types and explain their purpose.
- State the different primitive data types.
- State the different reference data types.

2.2.1 Data Types

When you define a variable in Java, you must tell the compiler what kind of a variable it is. That is, whether it will be expected to store an integer, a character, or some other kind of data. This information tells the compiler how much space to allocate in the memory depending on the data type of the variable. The data type of a variable, therefore, determines the type of data that can be stored in the variable and the operations that can be performed on this data.

Java is a strongly typed language. This means that every variable has a type, every expression has a type, and every type is strictly defined. It also means that all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility by the Java compiler. Any type mismatches are considered as errors and need to be corrected before compilation is over.

In Java programming language, data types are mainly divided into two categories and they are as follows:

- Primitive data types
- Reference data types

2.2.2 Primitive Data Types

The Java programming language provides primitive data types to store and represent data. A primitive data type, also called built-in data types, stores a single value, normally a literal of some sort, such as a number or a character.

Table 2.2 lists the primitive data types in Java.

Data Type	Description	Range
`byte`	8-bit signed integer	-128 to 127
`short`	16-bit signed integer	-32768 to 32767

Data Type	Description	Range
`long`	64-bit signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`int`	32-bit signed integer	-2,147,483,648 to 2,147,483,647
`float`	32-bit floating-point variable	-3.40292347E+38 to +3.40292347E+38
`boolean`	Stores a true or false value	true or false
`char`	16-bit Unicode character	0 to 65535
`double`	64-bit floating-point variable	-1.7976931348623157OE+308 to 1.7976931348623157OE+308

Table 2.2: Primitive Data Types in Java

2.2.3 Variables of Primitive Data Types

Whenever a variable is declared, it is either assigned a value (if the variable has been declared and initialized on the same line) or it holds a default value (if the variable has only been declared and not initialized). More than one variable of the same data type can be declared in Java by using the comma (,) separator.

The following code demonstrates the declaration and initialization of variables in Java.

Code Snippet:

```
int empNumber;
float salary;
char gender = 'M';
double shareBalance = 456790.897;
boolean ownVehicle = false;

empNumber = 101;
salary = 6789.50f;

System.out.println("Employee Number: "+empNumber);
System.out.println("Salary: "+salary);
System.out.println("Gender: "+gender);
System.out.println("Share Balance: "+shareBalance);
System.out.println("Owns vehicle: "+ownVehicle);
```

The snippet demonstrates how to declare and initialize variables in Java.

Here, variables of type `int`, `float`, `char`, `double`, and `boolean` are declared. All float values need to have an `f` appended at its end. Values are assigned to each of these variables and are displayed using the `println()` method.

2.2.4 Reference Data Types

A variable of a reference data type holds the reference to the actual value or a set of values represented by the variable.

Table 2.3 displays three reference data types that are available in Java.

Data Type	Description
Array	A collection of several items of the same data type such as names of students.
Class	A collection of variables and methods. For example, class Student containing the complete details of the students and the methods that operate on the details.
Interface	A type of class in Java used to implement multiple inheritance.

Table 2.3: Reference Data Types in Java

Knowledge Check 2

1. Can you match the values which will optimally fit with the corresponding data type?

Data Type		Value	
(A)	byte	**(1)**	256
(B)	char	**(2)**	50000
(C)	int	**(3)**	48999.988
(D)	short	**(4)**	126
(E)	double	**(5)**	‘F’

2.3 Formatted Output and Input

In this third lesson, **Formatted Output and Input**, you will learn to:

- State the different format specifiers in Java.
- Identify the methods for accepting formatted input.
- List the various escape sequence characters in Java.

2.3.1 Format Specifiers

Whenever an output is to be displayed on the screen, it needs to be formatted. The formatting can be done with the help of format specifiers in Java. The `printf()` method introduced in J2SE 5.0 can be used to format the numerical output to the console.

Table 2.4 lists some of the format specifiers in Java.

Format Specifier	Description
%d	Result formatted as a decimal integer
%f	Result formatted as a real number
%o	Results formatted as an octal number
%e	Result formatted as a decimal number in scientific notation
%n	Result is displayed in a new line

Table 2.4: Format Specifiers in Java

The following code demonstrates the use of various format specifiers.

Code Snippet:

```
int i = 55/22;
// decimal integer
System.out.printf ("55/22 = %d %n", i);
// Pad with zeros
double q = 1.0/2.0;
System.out.printf ("1.0/2.0 = %09.3f %n", q);
// Scientific notation
q = 5000.0/3.0;
System.out.printf ("5000/3.0 = %7.2e %n", q);
```

```
// Negative infinity
q = -10.0/0.0;
System.out.printf ("-10.0/0.0 = %7.2e %n", q);
// Multiple arguments
//Pi value, E-base of natural logarithm
System.out.printf ("pi = %5.3f, e = %5.4f %n", Math.PI,Math.E);
```

Output:

```
55/22 = 2
21.0/55.0 = 00000.500
5000/3.0 = 1.67e+03
-10.0/0.0 = -Infinity
pi = 3.142, e = 2.7183
```

In the snippet, '`%09.3f`' indicates that there will be total 9 digits including decimal point and three places of decimals. If the number of digits is less than 9, then it will be padded with zeroes. If '0' is omitted from '`%09.3f`', then it will be padded with spaces.

2.3.2 'Scanner' Class

The `Scanner` class allows the user to read values of various types. To use the `Scanner` class, pass the `InputStream` object to the constructor.

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

Here, `input` is an object of `Scanner` class and `System.in` is an input `stream` object.

Table 2.5 lists the different methods of the `Scanner` class that can be used to accept numerical values from the user.

Method	Description
`nextByte()`	Returns the next token as a byte value
`nextInt()`	Returns the next token as an int value
`nextLong()`	Returns the next token as a long value
`nextFloat()`	Returns the next token as a float value
`nextDouble()`	Returns the next token as a double value

Table 2.5: Methods of Scanner Class

The following code demonstrates the `Scanner` class methods and how they can be used to accept values from the user and finally print the output.

Code Snippet:

```
//creates an object and passes the inputstream object
Scanner s = new Scanner(System.in);
//Accepts values from the user
byte byteValue = s.nextByte();
int intValue = s.nextInt();
float floatValue = s.nextFloat();
long longValue = s.nextLong();
double doubleValue = s.nextDouble();

System.out.println("Values entered are: ");
System.out.println(byteValue + " " + intValue + " " + floatValue + " " +
longValue + " " + doubleValue);
```

Output:

Values entered are:

```
121 2333 456.789 456 3456.876
```

Note:

Package

A package is a collection of classes.

Scanner Class

The `Scanner` class belongs to the `java.util` package. The Scanner looks for tokens in the input. A token is a series of characters that ends with delimiters. A delimiter can be a whitespace (default delimiter for tokens in `Scanner` class), a tab character, a carriage return, or the end of the file. Thus, if we read a line that has a series of numbers separated by whitespaces, the scanner will take each number as a separate token.

Constructor

Constructors are used to create an instance of a class.

InputStream

Characters are read from the keyboard by using `System.in`.

2.3.3 Escape Sequences

An escape sequence is a special sequence of characters that is used to represent characters, which cannot be entered directly into a string. For example, to include tab spaces or a new line character in a line or to include characters which otherwise have a different connotation in a Java program (such as \ or “), escape sequences are used. An escape sequence begins with a backslash character (\), which indicates that the character (s) that follows should be treated in a special way. The output displayed by Java can be formatted with the help of escape sequence characters.

Table 2.6 displays the various escape sequences in Java.

Escape Sequence	Description
\b	Backspace character
\t	Horizontal Tab character
\n	New line character
\’	Single quote marks
\\	Backslash
\r	Carriage Return character
\”	Double quote marks
\f	Form feed
\xxx	Character corresponding to the octal value xxx, where xxx is between 000 and 0377
\uxxx	Unicode corresponding with encoding xxxx, where xxxx is one of four hexadecimal digits. Unicode escapes are distinct from the other escape types

Table 2.6: Escape Sequences in Java

The following code demonstrates the use of escape sequence characters.

Code Snippet:

```
// use of tab and new line escape sequences
System.out.println(“Java \t Programming \n Language”);
// printing Tom “Dick” Harry string
System.out.println(“Tom \“Dick\” Harry”);
```

Output:

```
Java      Programming
  Language
Tom "Dick" Harry
```

To represent a Unicode character, Unicode `\u` escape sequence can be used anywhere in a Java program. A Unicode character can be represented using hexadecimal or octal sequences.

The following code demonstrates the use of `\xxx` and `\uxxx` escape sequence types.

Code Snippet:

```
// Print 'Hello' using hexadecimal escape sequence characters
System.out.println("\u0048\u0065\u006C\u006C\u006F" + "!\n");

// Print 'Blake' using octal escape sequence character for 'a'
System.out.println("Bl\141ke\"2007\" ");
```

The output of Code Snippet 6 is as follows:

```
Hello!
Blake"2007"
```

Note: The hexadecimal escape sequence starts with \u followed by 4 hexadecimal digits. The octal escape sequence comprises 3 digits after back slash. For example,

`\xyy`

where, x can be any digit from 0 to 3 and y can be any digit from 0 to 7.

Knowledge Check 3

1. You are trying to use the format specifiers to display the output, "`100 * 55.0 = 5500.000`", "`100 * 55.0 = 5500.000000`", "`100 *  55.0 = 5.50e+03`", and "`100 * 55.0 = 0`". Which of the following code will help you to achieve this?

(A)	`double result = 100 * 55;`
	`int q = 0;`
	`System.out.printf ("100 * 55.0 = %1.3f %n", result);`
	`System.out.printf ("100 * 55.0 = %5f %n", result);`
	`System.out.printf ("100 * 55.0 = %1.5f %n", result);`
	`System.out.printf ("100 * 55.0 = %d %n", q);`

(B)	`double result = 100 * 55;`
	`int q = 0;`
	`System.out.printf ("100 * 55.0 = %1.3f %n", result);`
	`System.out.printf ("100 * 55.0 = %5f %n", result);`
	`System.out.printf ("100 * 55.0 = %1.2e %n", result);`
	`System.out.printf ("100 * 55.0 = %d %n", q);`

(C)	`double result = 100 * 55;`
	`int q = 0;`
	`System.out.printf ("100 * 55.0 = %09.3f %n", result);`
	`System.out.printf ("100 * 55.0 = %5f %n", result);`
	`System.out.printf ("100 * 55.0 = %f %n", result);`
	`System.out.printf ("100 * 55.0 = %d %n", q);`

(D)	`double result = 100 * 55;`
	`int q = 0;`
	`System.out.printf ("100 * 55.0 = %3f %n", result);`
	`System.out.printf ("100 * 55.0 = %5f %n", result);`
	`System.out.printf ("100 * 55.0 = %1.2e %n", result);`
	`System.out.printf ("100 * 55.0 = %d %n", q);`

2. Which of these statements about escape sequences are true and which of these are false?

(A)	The escape sequence character `\n` represents a new line character
(B)	The `nextDouble()` method of `Scanner` class returns the next token as a byte value
(C)	The escape sequence character `\r` represents a backslash character
(D)	The escape sequence character `\\` represents a new line character
(E)	The `nextByte()` method of `Scanner` class returns the next token as a byte value

2.4 Operators

In this fourth lesson, **Operators**, you will learn to:

- Describe an operator and explain its purpose.
- Identify and explain the use of Assignment, Arithmetic, and Unary operators.
- Identify and explain the use of Equality, Relational, and Conditional operators.
- Identify and explain the use of Bitwise and Bit Shift operators.
- Explain operator precedence.
- Explain operator associativity.

2.4.1 Purpose

All programming languages provide some mechanism for performing various operations on the data stored in variables. The simplest form of operations involves arithmetic (like adding, dividing, or comparing between two or more variables). A set of symbols is used to indicate the kind of operation to be performed on data. These symbols are called operators.

Consider the expression:

```
Z = X + Y
```

The + symbol in the statement is called the operator and the operation performed is addition. This operation is performed on the two variables `X` and `Y`, which are called operands. The combination of both the operator and the operands, `Z = X + Y`, is known as an Expression.

Java provides several categories of operators and they are as follows:

- Arithmetic
- Relational
- Logical
- Assignment
- Bitwise

➢ Bit Shift

2.4.2 Assignment Operator

The basic assignment operator is a single equal to sign, '='. This operator is used to assign the value on its right to the operand on its left. Assigning values to more than one variable can be done at a time. In other words, it allows you to create a chain of assignments.

Consider the following statements:

```
int balance = 3456;
char gender = 'M';
```

In the statements, the values `3456` and `M` are assigned to the variables namely, balance and gender.

In addition to the basic assignment operator, there are combined operators that allow you to use a value in an expression, and then set its value to the result of that expression.

```
X  = 3;
X += 5;
```

The second statement stores the value 8, the meaning of the statement is that `X = X + 5`.

The following code demonstrates the use of different assignment operators.

Code Snippet:

```
x = 10;    // Assigns the value 10 to variable x
x += 5;   // Increments the value of x by 5
x -= 5;   // Decrements the value of x by 5
x *= 5;   // Multiplies the value of x by 5
x /= 2;   // Divides the value of x by 2
x %= 2;   // Divides the value of x by 2 and the remainder is returned
```

2.4.3 Arithmetic Operator

Arithmetic operators manipulate numeric data and perform common arithmetic operations on the data. Operands of the arithmetic operators must be of numeric type. Boolean operands cannot be used, but character operands are allowed. The operators mentioned here are binary in nature: that is, these operate on two operands, such as X+Y. Here, + is a binary operator operating on X and Y.

Table 2.7 lists the arithmetic operators.

Operator	Description
+	Addition - Returns the sum of the operands.
-	Subtraction - Returns the difference of two operands.
*	Multiplication - Returns the product of operands.
/	Division – Returns the result of division operation.
%	Remainder - Returns the remainder from a division operation.

Table 2.7: Arithmetic Operators

The following code demonstrates the arithmetic operators.

Code Snippet:

```
x = 2 + 3;       // Returns 5
y = 8 – 5;     // Returns 3
x = 5 * 2;     // Returns 10
x =  5/2;      // Returns 2
y = 10 % 3;    // Returns 1
```

2.4.4 Unary Operator

Unary operators require only one operand. They perform various operations, such as incrementing/ decrementing the value of a variable by 1, negating an expression, or inverting the value of a boolean.

Table 2.8 lists the unary operators.

Operator	Description
+	Unary plus - Indicates a positive value.
-	Unary minus - Negates an expression.
++	Increment operator - Increments value of a variable by 1.
--	Decrement operator - Decrements value of a variable by 1.
!	Logical complement operator - Inverts a boolean value.

Table 2.8: Unary Operators

The `++` and `--` operators can be applied before (prefix) or after (postfix) the operand. The statements `++variable` and `variable++` both result in incrementing the variable value by 1. The only difference is that the prefix version (`++variable`) will increment the value before evaluating, whereas the postfix version (`variable++`) will first evaluate and then increment the original value.

The following code demonstrates the unary operators.

Code Snippet:

```
int i=5;
int j=i++;  // i=6, j=5
int k=++i;  //i=6,k=6
i  = - i ; //now i is -6
boolean result = false; //result is false
result = !result;      //now result is true
```

2.4.5 Equality and Relational Operators

Equality and relational operators test the relationship between two operands. An expression involving relational operators always evaluates to a boolean value (that is, either true or false).

Table 2.9 lists the various relational operators.

Operator	Description
==	Equal to - Checks equality of two numbers.
!=	Not Equal to - Checks inequality of two values.
>	Greater than - Checks if value on left is greater than the value on the right.
<	Less than - Checks if the value on the left is lesser than the value on the right.
>=	Greater than or equal to - Checks if the value on the left is greater than or equal to the value on the right.
<=	Less than or equal to – Checks if the value on the left is less than or equal to the value on the left.

Table 2.9: Relational Operators

The following code demonstrates the relational operators.

Code Snippet:

```
int value1 = 10;
int value2 = 20;
value1 == value2; //returns false
value1 != value2; //returns true
value1 > value2; //returns false
value1 < value2; //returns true
```

```
value1 <= value2; //returns true
```

2.4.6 Conditional Operators

Conditional operators (&& and ||) work on two boolean expressions. These operators exhibit short-circuiting behavior, which means that the second operand is evaluated only if needed.

Table 2.10 lists the two conditional operators.

Operator	Operator
&&	Conditional AND - Returns true only if both the expressions are true.
||	Conditional OR - Returns true if either expression is true or both the expressions are true.

Table 2.10: Conditional Operators

Ternary operator (`?:`) is another type of conditional operator; it accepts three operands. The syntax for the usage of ternary operator is as follows:

```
expression1 ? expression2 : expression3
```

where,

`expression1`: is a boolean condition that returns a true or false value

`expression2`: is the value returned if expression1 evaluates true

`expression3`: is the value returned if expression1 evaluates false

The following code demonstrates the conditional operators.

Code Snippet:

```
int first = 10;
int second = 20;
(first == 30) && (second == 20 ); //returns false
(first == 30) || (second == 20 ); //returns true
(second > first) ? "second number is greater" : "first number greater or equal"
;
//returns "second number is greater"
```

2.4.7 Bitwise and Bit Shift Operators

Bitwise operators work on binary representations of data. These operators are used to change individual bits in an operand. Bitwise shift operators are used to move all the bits in the operand left or right a given number of times.

Table 2.11 lists the various bitwise operators.

Operator	Description
&	Bitwise AND - compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0.
|	Bitwise OR - compares two bits and generates a result of 1 if the bits are complementary; otherwise, it returns 0.
^	Exclusive OR - compares two bits and generates a result of 1 if either or both bits are 1; otherwise, it returns 0.
~	Complement operator - used to invert all of the bits of the operand.
>>	Shift Right operator - Moves all bits in a number to the right by one position retaining the sign of negative numbers.
<<	Shift Left operator - Moves all the bits in a number to the left by the specified position.

Table 2.11: Bitwise Operators

The following code demonstrates the bitwise operators.

Code Snippet:

```
int x = 23;
int y = 12;
//23 = 10111 , 12 = 01100
(x & y) // returns 4 , i.e,  4 = 00100
(x | y) //returns 31, i.e 31 = 11111
(x ^ y) //returns 27, i.e 31 = 11011
int a = 43;
int b = 1;
//43 = 010011 , 1 = 000001
(a >> b) // returns 21 , i.e, 21 = 0010101
(a << b) //returns 86 , i.e, 86 = 1010110
```

2.4.8 Operator Precedence

Expressions that are written generally consist of several operators. The rules of precedence decide the order in which each operator is evaluated in any given expression.

Table 2.12 shows the order of precedence of operators, from highest to lowest in which operators are evaluated in Java.

Order	Operator
1.	Parentheses like ( ).
2.	Unary Operators like +, -, ++, --, ~, !.
3.	Arithmetic and Bitwise Shift operators like *, /, %, +, -, >>, <<.
4.	Relational Operators like >, >=, <, <=, ==, !=.
5.	Conditional and Bitwise Operators like &, ^, |, &&, ||.
6.	Conditional and Assignment Operators like ?:, =, *=, /=, +=, and -=.

Table 2.12: Precedence of Operators

Parentheses are used to change the order in which an expression is evaluated. Any part of an expression enclosed in parentheses is evaluated first.

The following code demonstrates the precedence of operators in an expression.

Code Snippet:

```
2*3+4/2 >  3 && 3<5 || 10<9
```

The evaluation is as shown:

1. **(2*3+4/2)** > 3 && 3<5 || 10<9

   First the arithmetic operators are evaluated.

2. **((2*3)+(4/2))** > 3 && 3<5 || 10<9

   Division and Multiplication are evaluated before addition and subtraction.

3. **(6+2)** >3 && 3<5 || 10<9

4. **(8 >3)** && [3<5] || [10<9]

Next to be evaluated are the relational operators all of which have the same precedence. These are therefore evaluated from left to right.

5. **(True && True) ||** False

   The last to be evaluated are the logical operators. && takes precedence over ||.

6. True || False

7. True

2.4.9 Operator Associativity

When two operators with the same precedence appear in an expression, the expression is evaluated, according to its associativity. For example, in Java the - operator has left-associativity and x - y - z is interpreted as (x - y) - z, and = has right-associativity and x = y = z is interpreted as x = (y = z).

Table 2.13 shows the Java operators and their associativity.

Operator	Description	Associativity
(), ++, --	Parentheses, post increment/decrement	Left to right
++, --, +, -, !,~	Pre increment/ decrement unary plus, unary minus logical NOT, bitwise NOT	Right to left
*, /, %, +, -	Multiplicative and Additive	Left to right
<<, >>	Bitwise shift	Left to right
<, <, >=, <=, ==, !=	Relational and Equality operators	Left to right
&, ^, |	Bitwise AND, XOR, OR	Left to right
&&, ||	Conditional AND, OR	Left to right
?:	Conditional operator (Ternary)	Right to left
=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	Assignment	Right to left

Table 2.13: Java Operators and their Associativity

The following code demonstrates the associativity of operators in an expression.

Code Snippet:

Consider the following expression:

```
2+10+4-5*(7-1)
```

1. According to the rules of operator precedence, the '*' has higher precedence than any other operator in the equation. Since 7-1 is enclosed in parenthesis, it is evaluated first.

   ```
   2+10+4-5*6
   ```

2. Next, '*' is the operator with the highest precedence. Since there are no more parentheses, it is evaluated according to the rules.

   ```
   2+10+4-30
   ```

3. As '+' and '-' have the same precedence, the left associativity works out.

   ```
   12+4-30
   ```

4. Finally, the expression is evaluated from left to right.

   ```
   16 - 30
   ```

The result is `-14`.

Knowledge Check 4

1. Can you match the operators in Java against their corresponding description?

Description		Operator	
(A)	Works on boolean expressions	**(1)**	Relational
(B)	Requires only one operand	**(2)**	Arithmetic
(C)	Assigns values to identifiers	**(3)**	Conditional
(D)	Tests the relationship between two operands	**(4)**	Assignment
(E)	Requires two operands to operate	**(5)**	Unary

2. You are using various operators to solve an expression. Which of the following statements will display the output as "true"?

(A)	`int a = 5, b = 4, c = 3,d = 2, e = 1;`
	`System.out.println((a+b/5.0-c*d+1*e > (a*b)) && (b<d));`

(B)	`int a = 5, b = 4, c = 3,d = 2, e = 1;`
	`System.out.println((a*b/5.0+c*d+1*e > (a/b)) && (b>d));`

(C)	`int a = 5, b = 4, c = 3,d = 2, e = 1;`
	`System.out.println((a*b+5.0-c*d+1*e < (a*b)) && (b>d));`

(D)	`int a = 5, b = 4, c = 3,d = 2, e = 1;`
	`System.out.println((a*b/5.0-c*d*1*e > (a*b)) && (b<d));`

2.5 Type Casting

In this last lesson, **Type Casting**, you will learn to:

➢ State the two types of type casting.

➢ Describe implicit type casting.

➢ Describe explicit type casting.

2.5.1 Type Casting

In any application, there may be situations where one data type may need to be converted into another data type. The type casting feature in Java helps in such conversion. Type conversion or typecasting refers to changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies. For instance, values from a more limited set, such as integers, can be stored in a more compact format. It can be converted later to a different format enabling operations not previously possible, such as division with several decimal places worth of accuracy. In object-oriented programming languages, type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

If a conversion results in the loss of precision, as in an `int` value converted to a `short`, then the compiler issues an error message unless an explicit cast is made.

The two types of casting are as follows:

➢ Implicit casting

➢ Explicit casting

Figure 2.2 shows the two types of casting.

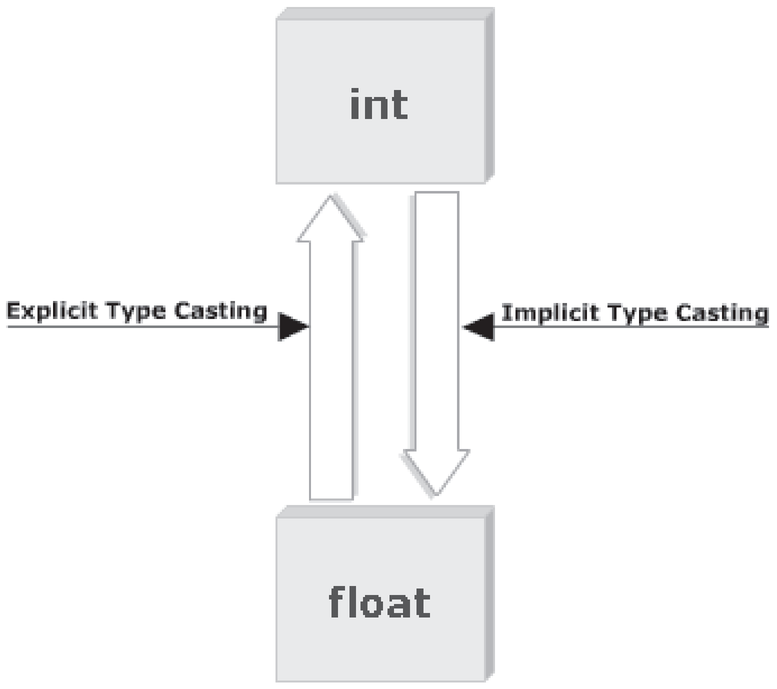


Figure 2.2: Type Casting

2.5.2 Implicit Type Casting

When one type of data is assigned to a variable of another type, then automatic type conversion takes place, also referred to as implicit type casting, provided it meets the conditions specified:

- ➢ The two types should be compatible.
- ➢ The destination type should be larger than the source.

The primitive numeric data types that can be implicitly cast are as follows:

- ➢ byte (8 bits) to short, int, long, float, double
- ➢ short(16 bits) to int, long, float, double
- ➢ int (32 bits) to long, float, double
- ➢ long(64 bits) to float, double

This is also known as the type promotion rule.

Figure 2.3 shows the implicit type casting.

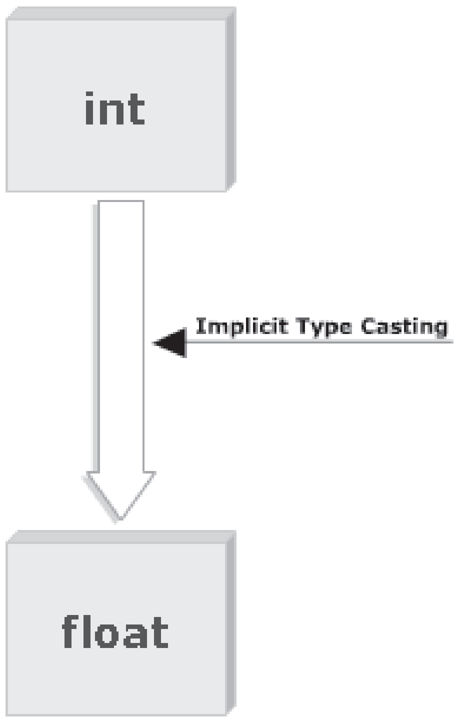


Figure 2.3: Implicit Type Casting

The following code demonstrates implicit type conversion.

Code Snippet:

```
double  dbl = 10;
long lng = 100;
int  in = 10;
dbl = in; // assigns the integer value to double variable
lng = in; // assigns the integer value to long variable
```

Note: The type promotion rules are listed as follows:

- All `byte` and `short` values are promoted to `int` type.
- If one operand is `long`, the whole expression is promoted to `long`.

- If one operand is float then the whole expression is promoted to float.
- If one operand is double then the whole expression is promoted to double.

2.5.3 Explicit Type Casting

A data type with lower precision, such as short, can be converted to a type of higher precision, such as int, without using explicit casting. However, to convert a higher precision data type to a lower precision data type, such as float to int data type, an explicit cast is required. Otherwise, the compiler will display an error message.

The following code adds a float value to an int and stores the result as an integer.

Code Snippet:

```
float a = 21.3476f;
int b = (int) a + 5;
```

The float value in a variable is converted into an integer value 21. It is then added to 5, and the resulting value, 26, is stored in b. This type of conversion is known as truncation. The fractional component is lost when a floating point is assigned to an integer type, resulting in the loss of precision.

Figure 2.4 shows the explicit type casting of data types.

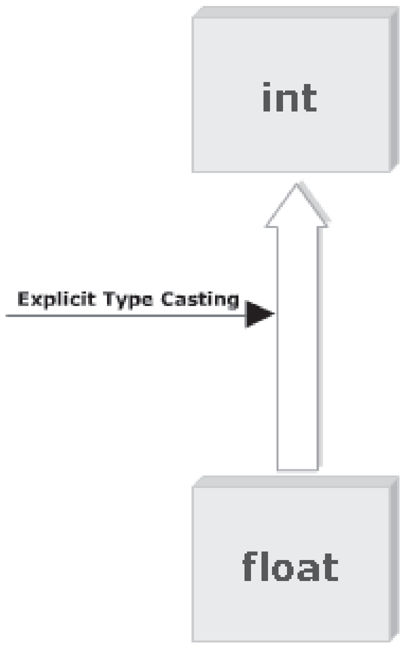


Figure 2.4: Explicit Type Casting

There are several kinds of explicit conversions. They are as follows:

- **checked**

  Before the conversion is performed, a runtime check is done to see if the destination type can hold the source value. If not, an error condition is raised.

- **unchecked**

  No check is performed. If the destination type cannot hold the source value, the result is undefined.

- **bit pattern**

  The data is not interpreted at all, and its raw bit representation is copied verbatim. This can also be achieved via aliasing.

Knowledge Check 5

1. You want the output to be displayed as, 'floatTest=1.0', 'dblTest=1.0', and 'sum=2'. Can you arrange the steps in sequence to achieve the same?

(1)	`System.out.println("dblTest=" + dblTest);`
	`int sum = (int)(dblTest + floatTest);`

(2)	`System.out.println("sum=" + sum);`

(3)	`boolean boolTest = true;`
	`float floatTest = 3.14f;`

(4)	`System.out.println("floatTest=" + floatTest);`
	`dblTest = (double)( boolTest?1:0);`

(5)	`double dblTest = 0.000000000000053;`
	`floatTest = (float)(boolTest?1:0);`

Module Summary

In this module, **Variables and Operators**, you learnt about:

➢ **Variables**

Variables store the values required in the program. Variables should be declared, before they are used. Java programming language has its own set of rules and conventions for naming variables.

➢ **Data Types**

Data types determine the type of values that can be stored in a variable and the operations that can be performed on them. Data types in Java are divided mainly into primitive types and reference types. Primitive data types contain the actual value while reference data types contain a reference to the actual value.

➢ **Formatted Output and Input**

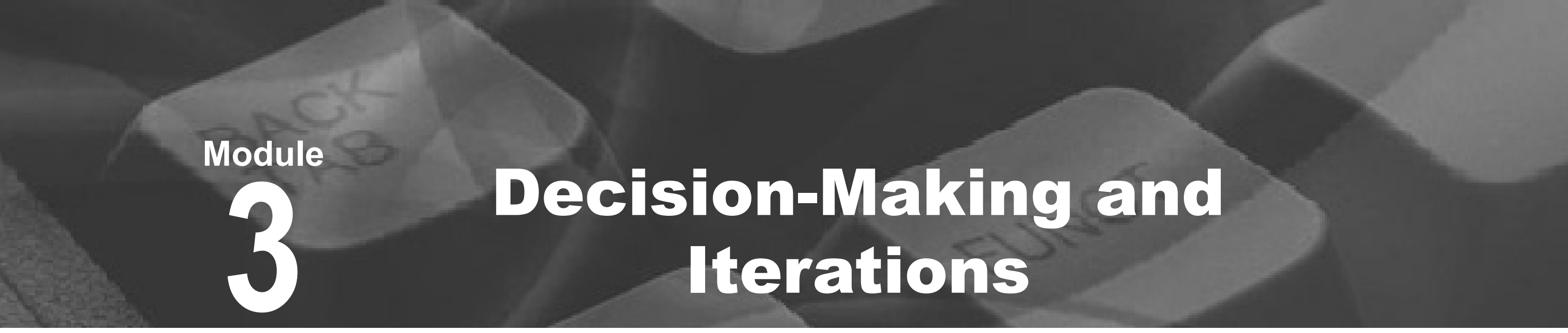
The `printf()` method along with the format specifiers can be used to format the output displayed on the screen. `Scanner` class methods can be used to format the program input. Java uses escape sequences to represent certain special character values.

➢ **Operators**

Operators are symbols that help to manipulate or perform some sort of function on data. The symbol used in the expression to perform an operation is called the operator. The variables on which the operation is performed are called operands. The combination of both the operator and the operand is known as an expression. Operators can be classified into arithmetic, assignment, relational, bitwise, unary, and conditional.

➢ **Type Casting**

The type casting feature helps in converting a certain data type to another data type. The type casting can be automatic or manual and should follow the rules for promotion.



Module 3 Decision-Making and Iterations

Module Overview

Welcome to the module, **Decision-Making and Iterations**. The ability to compare values and alter the course of a program based on the result of the comparison is what gives a computer the power to solve problems. There are two aspects to making decisions: the first is the comparison of data and the second is the sequence of execution. This module focuses more on decision-making statements and the iteration constructs. The module also covers the various jump statements that are used in Java programs.

In this module, you will learn about:

- Decision-Making Statements
- Introduction to Loops
- Jump Statements

3.1 Decision-Making Statements

In this first lesson, **Decision-Making statements**, you will learn to:

- Identify the need for decision-making statements and its types.
- Explain various forms of the `if` statement.
- Explain the `switch-case` statement.
- Compare the `if-else` and the `switch-case` construct.

3.1.1 Purpose and Types

A Java program is a set of statements, which are executed sequentially in the order in which they appear. However, in some cases, the order of execution of statements may change based on the evaluation of certain conditions.

The Java programming language possesses decision-making capabilities. The decision-making or control statements supported by Java are as follows:

- `if` statement

- `if-else-if` statement
- `switch-case` statement

Decision-making statements enable us to change the flow of the program. Based on the evaluation of a condition, a statement or a sequence of statements is executed.

3.1.2 'if' Statement

The `if` statement can be implemented in various forms based on the complexity of the conditions to be tested.

The different forms of the `if` statement are as follows:

- **Simple `if` statement**

  The `if` statement helps in decision-making based on the evaluation of a given condition to true or false. If the condition is true, then the statements in the `if` block get executed. If condition evaluates to false, the control is transferred directly outside the `if` block.

  The following is the syntax of `if` statement.

  Syntax:

```
if (condition)
{
      // one or more statements
}
```

  The following code demonstrates the use of `if` statement.

  Code Snippet:

```
int first = 400, second = 700, result;
result = first + second;
if(result > 1000)
{
      second = second + 100;
}
System.out.println("The value of second is " + second);
```

The program tests the value of result and accordingly calculates the value of `second` and prints it. If the result is greater than 1000, then the variable `second` gets increased by 100 and is printed. If not, the value of `second` is not increased and the original value is printed.

Output:

```
The value of second is 800
```

➢ **`if-else` statement**

Generally, it is not only important to specify an action to be performed when the condition is true but also to specify what action is to be performed if the condition is false. To do this, the `if-else` statement can be used.

The following is the syntax of `if-else` statement.

Syntax:

```
if (condition)
{
     // one or more statements
}
else
{
    // one or more statements
}
```

The following code demonstrates the use of `if-else` statement.

Code Snippet:

```
int number = 11, remainder;
remainder = number % 2;
if(remainder == 0)
{
    System.out.println("Number is even");
}
```

```
else
{
    System.out.println("Number is odd");
}
```

The program checks whether a number is even or odd. If the remainder is 0, the message 'Number is even' is printed. Else, the message 'Number is odd' is printed.

Output:

```
Number is odd
```

➢ **Nested-if Statement**

The if-else statement tests the result of a condition, that is, a boolean expression, and performs appropriate actions based on the result. An if statement can also be used inside another if. This is known as nested-if. Thus, a nested-if is an if statement that is the target of another if or else.

The important points to remember in nested-if statements are as follows:

➢ An else statement should always refer to the nearest if statement.

➢ It should be within the same block as the else and it should not be associated with an else.

The following is the syntax of nested-if statement.

Syntax:

```
if(condition) {
    if(condition)
        True-block statement(s);
    else
        False-block statement(s);

} else {
    False-block statement(s);
}
```

The following code demonstrates the use of `nested-if` statements.

Code Snippet:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a number: ");
num = input.nextInt();

// Number is divisible by 3
//Outer if statement
if(num % 3 == 0) {

    //Number is divisible by 5
    // Inner if statement
    if (num % 5 == 0)

       //Message is displayed if number is divisible by both 3 and 5
       System.out.println("The number is divisible by both 3 and 5.");
    else

     //Message is displayed if number is divisible by 3 but not by 5
     System.out.println("The number is divisible by 3 but not by 5.");
    } else {

     //Message is displayed if number is not divisible by 3 and 5
     System.out.println("The number is not divisible by 3 and 5.");
```

Here, code declares a variable **`num`** and stores an integer value accepted from the user. Then, using `nested-if` statements, it checks whether **`num`** is divisible by 3 and 5 or only by 3, and then prints an appropriate statement. Here, the final else is associated with **`if(num % 3 == 0)`**. The inner else refers to **`if(num % 5 == 0)`**, because it is closest to the inner `if` within the same block.

➢ **Multiple `if` statements**

The multiple `if` construct is known as the `if-else-if` ladder. The conditions are evaluated sequentially starting from the top of the ladder and moving downwards. When a true condition is found, the statement associated with the true condition is executed.

The following is the syntax of `if-else-if` construct.

Syntax:

```
if(condition)
{
    // one or more statements
}
else if (condition)
{
    // one or more statements
}
else
{
    // one or more statements
}
```

The following code demonstrates the use of `if-else-if` construct.

Code Snippet:

```
int totalMarks = 59;
if(totalMarks >= 90)
{
    System.out.println("Grade = A+");
}
else if (totalMarks >= 60)
{
    System.out.println("Grade = A");
}
else if (totalMarks >= 40)
{
    System.out.println("Grade = B+");
}
else if (totalMarks >= 30)

    System.out.println("Grade = B");
}
```

```
else
{
    System.out.println("Fail");
}
```

Output:

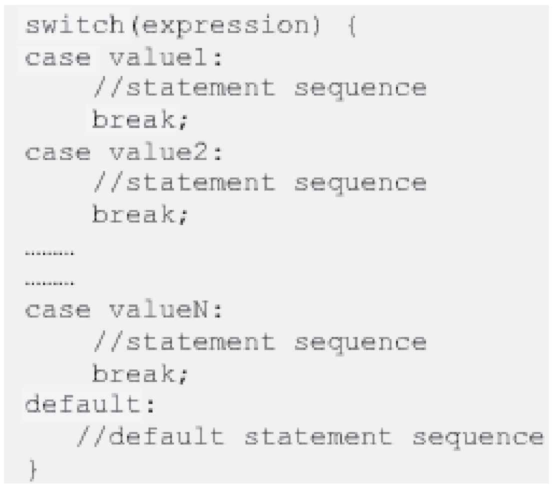
```
Grade = B+
```

3.1.3 'switch-case' Statement

A program is difficult to comprehend, when there are too many `if` statements representing multiple selection constructs. To avoid this, the `switch-case` approach can be used as an alternative for multiple selections. The use of the `switch-case` statement results in better performance.

The `switch-case` statement is used when a variable needs to be compared against different values.

Figure 3.1 shows the syntax of `switch-case` statement.



```
switch(expression) {
case value1:
    //statement sequence
    break;
case value2:
    //statement sequence
    break;
………
………
case valueN:
    //statement sequence
    break;
default:
   //default statement sequence
}
```

Figure 3.1: 'switch-case' Statement

- **`switch`**

  The `switch` keyword is followed by an integer expression enclosed in parentheses. The expression must be of type `int`, `char`, `byte`, or `short`. Each `case` value must be a unique literal. Thus, it must be a constant and not a variable. The `switch` statement executes the `case` corresponding to the value of the expression. The `break` statement terminates the statement sequence and continues with the statement following the `switch`. If there is no corresponding `case` value, the `default` clause is executed.

- **`case`**

  The `case` keyword is followed by an integer constant and a colon. Each case value is a unique literal. The `case` statement might be followed by a code sequence that are executed when the `switch` expression and the `case` value match.

- **`default`**

  If no `case` value matches the `switch` expression value, execution continues at the `default` clause. This is the equivalent of the "`else`" for the `switch` statement.

- **`break`**

  The `break` statement is used inside the `switch-case` statement to terminate the execution of the statement sequence. The control is then transferred to the first statement after the end of the `switch`. The `break` statement is optional. If there is no `break`, execution flows sequentially into the next `case` statement. Sometimes, multiple cases can be present without `break` statements between them.

The following code demonstrates the use of `switch-case` statement.

Code Snippet:

```
int choice = 3;
switch (choice)
{
     case 1:
         System.out.println("Addition");
         break;
     case 2:
          System.out.println("Multiplication");
          break;
     case 3:
          System.out.println("Division");
          break;
      case 4:
          System.out.println("Subtraction");
          break;
       default:
          System.out.println("Invalid choice");
}
```

Output:

```
Division
```

In the code, case 3 will be executed because value of choice is 3. So, the message, Division is displayed, and in the next statement when `break` is encountered the program exits.

➢ **Nested-switch-case Statement**

A `switch` statement can also be used as part of another `switch` statement. This is called a `nested switch`.

Since, a `switch-case` statement defines its own blocks, no conflicts arise between the case constants in the inner switch and those in the outer switch.

The following code demonstrates the use of `nested-switch-case` statements.

Code Snippet:

```
.....
switch(day) {
    case 0:
        switch(target) {
            case 1:
                System.out.println("Target is 1 to 7.");
              break;
        }
        break;
    case 1:
        System.out.println("Sunday");
        break;
    case 2:
        System.out.println("Monday");
        break;
    case 3:
        System.out.println("Tuesday");
        break;
    case 4:
        System.out.println("Wednesday");
        break;
.....
```

As shown in the code, the **`case 1:`** statement in the inner switch does not conflict with the **`case 1:`** statement in the outer switch. The variable **`day`** is only compared with the list of cases at the outer level. If **`day`** is 0, then target is compared with the inner list cases.

There are three important features of switch statements and they are as follows:

- The `switch` differs from if. The `switch` can only test for equality, whereas if can test for any type of `Boolean` expression. The `switch` looks only for a match between the value of the expression and one of its `case` constants.

- No two `case` constants in the same `switch` can have identical values, but a `switch` statement enclosed by an outer `switch` can have `case` constants in common.

- A `switch` statement is more efficient than a set of `nested-if` statements.

3.1.4 Comparison

The `if-else-if` and the `switch-case` decision-making statements have similar use in a program, but there are distinct differences between them.

Table 3.1 lists the differences of `if` and `switch` statement.

if-else-if	switch
Each `if` has its own logical expression to be evaluated as true or false.	Each `case` refers back to the original value of the expression in the `switch` statement.
The variables in the expression may evaluate to a value of any type.	The expression must evaluate to a `byte`, `short`, `char` or `int`.
Only one of the blocks of code is executed.	If the `break` statement is omitted, the execution will continue into the next block.

Table 3.1: Differences of if and switch Statement

Knowledge Check 1

1. You are trying to use the `if` statement to display the value “`100 Not Equal`”. Which of the following code will help you to achieve this?

(A)
```
int value = 100;
boolean bool = true;
if((bool==true) && ((value+=100)==200))
{
      System.out.printf(“Equal  “+value);
}
else
{
    System.out.println(value + “ Not Equal”);
}
```

(B)
```
int value = 100;
boolean bool = false;
if((bool==true) && ((value==100)==200))
{
     System.out.printf(“Equal  “+value);
}
else
{
    System.out.println(value + “ Not Equal”);
}
```

(C)
```
int value = 100;
boolean bool = false;
if((bool==false) && ((value+=100)==200))
{
      System.out.printf(“Equal  “+value);
}
else
{
     System.out.println(value + “ Not Equal”);
}
```

(D)
```
int value = 100;
boolean bool = false;
if((bool==true) && ((value+=100)==200))
{
     System.out.printf(“Equal  “+value);
}
else
{
     System.out.println(value + “ Not Equal”);
}
```

2. You are using the `switch-case` statement to display the output '`value is ten`' and '`value is twenty`'. Which of the following code will help you to achieve this?

(A)

```
int value = 10;
 switch(value)
{
     default:
        System.out.println(“Invalid value”);
        break;
     case 10:
         System.out.println(“value is ten”);
         break;
      case 20:
         System.out.println(“value is twenty”);
         break;
}
```

(B)

```
int value = 10;
 switch(value)
{
     default:
        System.out.println(“Invalid value”);
        break;
     case 10:
         System.out.println(“value is ten”);
         break;
      case 20:
         System.out.println(“value is twenty”);
}
```

(C)

```
int value = 10;
 switch(value)
{
     default:
        System.out.println(“Invalid value”);
        break;
     case 10:
         System.out.println(“value is ten”);
      case 20:
         System.out.println(“value is twenty”);
         break;
}
```

(D)
```
int value = 10;
 switch(value)
{
     default:
        break;
        System.out.println("Invalid value");
     case 10:
         System.out.println("value is ten");
      case 20:
         System.out.println("value is twenty");
}
```

3.2 Introduction to Loops

In this second lesson, **Introduction to Loops**, you will learn to:

- Identify the need for a loop and list the types of loops.
- Explain the `while` statement and the rules associated with it.
- Identify the purpose of the `do-while` statement.
- State the need of `for` statement.
- Describe nested loops.
- Compare the different types of loops.

3.2.1 Need for Loops

A computer program is a set of statements, which are usually executed sequentially. However, in certain situations it is necessary to repeat certain steps to meet a specified condition.

For example, if the user wants to write a program that calculates and displays the sum of the first 10 numbers 1, 2, 3, …, 10.

One way to calculate the same is as follows:

1+2=3

3+3=6

6+4=10

10+5=15

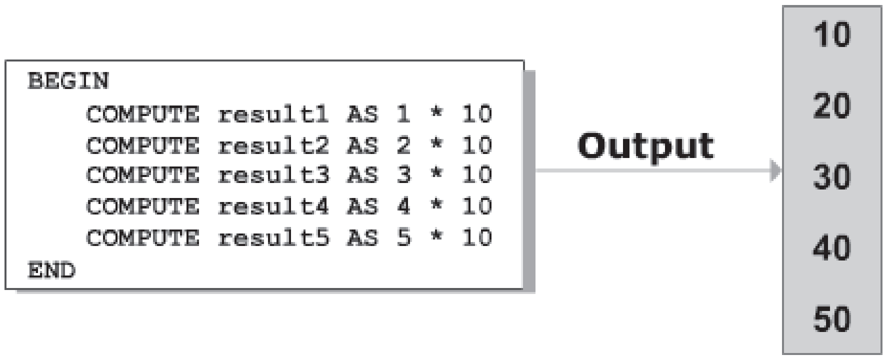
15+6=21

. . .

and so on.

This technique is suitable for relatively small calculations. However, if the program requires adding the first 200 numbers, it would be tedious to add up all the numbers from 1 to 200 using the mentioned technique. In such situations, iterations or loops come to our rescue.

Figure 3.2 shows the need for loops to display multiples of 10.



```
BEGIN
    COMPUTE result1 AS 1 * 10
    COMPUTE result2 AS 2 * 10
    COMPUTE result3 AS 3 * 10
    COMPUTE result4 AS 4 * 10
    COMPUTE result5 AS 5 * 10
END
```

Figure 3.2: Need for Loops

3.2.2 Definition and Types

A loop comprises a statement or a block of statements that are executed repeatedly until a particular condition evaluates to true or false.

Loops enable programmers to develop concise programs, which otherwise would require thousands of program statements.

The loop statements supported by Java programming language are:

- `while`
- `do-while`
- `for`

3.2.3 'while' Statement

The `while` statement in Java is used to execute a statement or a block of statements while a particular condition is true. The condition is checked before the statements are executed. The condition for the `while` statement can be any expression which returns a `boolean` value.

The following is the syntax of `while` statement.

Syntax:

```
while (expression)
{
     // one or more statements
}
```

The following code demonstrates the use of `while` statement.

Code Snippet:

```
int num = 1;
while(num <= 5)
{
   System.out.printf("\n%d * 10 = %d",num,(num * 10));
   num++;
}
```

As shown in the code, a variable of type integer, **`num`**, is declared to store the number. The variable **`num`** is initialized to 1 and is used in the `while` loop to start multiplication from 1.

The condition **`num <= 5`** ensures that the `while` loop executes as long as the value in **`num`** is less than or equal to 5. The execution of the loop stops when condition becomes false, that is, when the value of **`num`** reaches 6. The first statement within the body of the loop calculates the value of product by multiplying **`num`** with 10. The next statement prints this value and the last statement within the body of the loop changes the value of **`num`** by incrementing it by 1.

Output:

```
1 * 10 = 10
2 * 10 = 20
3 * 10 = 30
4 * 10 = 40
5 * 10 = 50
```

The body of the while loop will be empty if it contains a null statement and it is syntactically correct in Java.

The following code demonstrates the use of a null statement using while loop.

Code Snippet:

```
.....
int num1 = 1;
int num2 = 30;
while(++num1 < --num2);
System.out.println("Midpoint is: " + num1);
.....
```

As shown in the code, the value of **num1** is incremented and the value of **num2** is decremented. These values are then compared with one another. The loop repeats till the value of **num1** is equal to or greater than **num2**. Thus, upon exit **num1** will hold a value that is midway between the original values of **num1** and **num2**. The output of this code will be:

```
    The midpoint is: 16
```

3.2.4 Rules

The following points should be noted when using the while statement:

- The values of the variables used in the expression must be set at some point before the while loop is reached. This process is called the initialization of variables and has to be performed once before the execution of the loop.

- The body of the loop must have an expression that changes the value of the variable that is a part of the loop's expression. A variable is said to be incremented if its value increases in the body of the loop and is said to be decremented if its value decreases.

For example, if a while statement is written as follows:

```
while (true)
{
   . . .
}
```

The condition is simply a boolean value true. This leads to an infinite loop.

The following code demonstrates the use of an infinite loop.

Code Snippet:

```
.....
int count = 0;
while(count < 100) {
    System.out.println("This  goes on forever, HELP!!!");
    count = count + 10; \\Incrementing the value of count by 10.
    System.out.println("Count = " + count);
    count= count - 10;  \\Decrementing the value of count by 10.
    System.out.println("Count = " + count);
}
.....
```

As shown in the code, the value of **`count`** is always 0, which is less than 100. So, the expression always returns a **`true`** value. Hence, the loop never ends. A `break` statement can be used to terminate such programs. Thus, it will just go into the loop once and terminate, displaying the output as:

```
    This goes on forever, HELP!!!
    Count = 10
    Count  = 0
```

However, this is not to be practiced in real world scenarios.

3.2.5 'do-while' Statement

The `do-while` statement checks the condition at the end of the loop rather than at the beginning to ensure that the loop is executed at least once. The condition of the `do-while` statement usually comprises of an expression that evaluates to a boolean value.

The following is the syntax of `do-while` statement.

Syntax:

```
do
{
     statement(s);
}
while (expression);
```

The following code demonstrates the use of `do-while` statement.

Code Snippet:

```
int num = 1, sum = 0;
do
{
     sum = sum + num;
     num++;
}
while(num <= 10);
System.out.printf("Sum = %d",sum);
```

As shown in the code, two integer variables, **`num`** and **`sum`** are declared and initialized to 1 and 0 respectively. The loop block begins with a `do` statement. The first statement in the body of the loop calculates the value of `sum` by adding the current value of **`sum`** with **`num`** and the next statement in the loop changes the value of **`num`** by incrementing it by 1. Next, the condition, **`num <= 10`**, included in the `while` statement is evaluated. If the condition is met, the instructions in the loop are repeated. If the condition is not met (that is, when the value of **`num`** becomes 11), the loop terminates and the value in the variable **`sum`** is printed.

Output:

```
Sum = 55
```

3.2.6 'for' Statement

The `for` loops are especially used when the user knows how many times the statements need to be executed in the code block of the loop. It is similar to the `while` statement in its function. The statements within the body of the loop are executed as long as the condition is true. Here too, the condition is checked before the statements are executed.

The following is the syntax of `for` statement.

Syntax:

```
for(initialization; condition; increment/decrement)
{
  // one or more statements
}
```

where,

`initialization`: initializes the variables that will be used in the condition.

`condition`: comprises the condition that is tested before the statements in the loop are executed.

`increment/decrement`: comprises the statement that changes the value of the variable (s) to ensure that the condition specified in the condition section is reached. Typically, increment and decrement operators, such as `++`, `--`, and shortcut operators, such as `+=` or `-=` are used in this section. Note, that there is no semicolon at the end of the increment/decrement expressions.

The three declaration parts are separated by semicolons. When the loop starts the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable and acts as a counter that controls the loop. The initialization expression is executed only once. Next, the boolean expression is evaluated. It usually tests the loop control variable against a targeted value. If the expression is true, then the body of the loop is executed and if the expression is false then the loop terminates. Lastly, the iteration portion of the loop is executed. This expression usually increments or decrements the loop control variable.

The following code demonstrates the use of `for` statement.

Code Snippet:

```
int num, product;
for(num = 1; num <= 5; num++)
{
   product = num * 10;
   System.out.printf("\n%d *  10 = %d ",num,product);
}
```

In the initialization section of the `for` loop, the **`num`** variable is initialized to 1. The condition statement, **`num <= 5`**, ensures that the `for` loop executes as long as **`num`** is less than or equal to 5. The loop exits when the condition becomes false, that is, when the value of **`num`** becomes equal to 6. Finally, the increment statement **`num++`** in the increment/decrement section of the `for` statement increments the value of `num` by 1. The increment/decrement expression is evaluated after the first round of iteration.

Output:

```
1 *  10 = 10
2 *  10 = 20
3 *  10 = 30
4 *  10 = 40
5 *  10 = 50
```

The scope of the for loop can be extended by including more than one initialization or increment expressions in the for loop specification. The expressions are separated by the 'comma' operator and evaluated from left to right. The order of the evaluation is important if the value of the second expression depends on the newly calculated value.

The Following code demonstrates the use of for loop with the 'comma' operator to print the addition table for two variables.

Code Snippet:

```
.....
int i, j;
int max = 10;
System.out.println("The sum of two variables for a table of 10 is:");

for(i = 0, j = max; i <= max; i++, j--) {
    System.out.printf("\n%d + %d = %d", i, j, i+j);
}
.....
```

As shown in the code, three integer variables **i**, **j**, and **max** are declared. The variable **max** is assigned a value **10**. Further, within the initialization section of the **for** loop, the **i** variable is assigned a value **0** and **j** is assigned the value of **max**, that is, **10**. Thus, two parameters are initialized using a 'comma' operator. The condition statement, **i <= max**, ensures that the for loop executes as long as i is less than or equal to **max** that is **10**. The loop exits when the condition becomes false, that is, when the value of i becomes equal to **11**. Finally, the iteration expression again consists of two expressions, **i++**, **j--**. After each iteration, **i** is incremented by 1 and **j** is decremented by 1. The sum of these two variables which is always equal to **max** is printed.

The output of this code will be as follows:

```
The sum of two variables for a table of 10 is:
0 + 10 = 10
1 + 9 = 10
2 + 8 = 10
3 + 7 = 10
4 + 6 = 10
5 + 5 = 10
6 + 4 = 10
7 + 3 = 10
8 + 2 = 10
9 + 1 = 10
10 + 0 = 10
```

Alternatively, any or all expressions in the `for` loop may be left blank.

The following code demonstrates the use of `for` loop without the first expression.

Code Snippet:

```
.....
// initialization of num outside of for loop
int num=1;
for(;num ! = 40 ;num ++) {
    System.out.println("Enter a number: ");
    num = input.nextInt();
}
.....
```

The code will accept a value for **`num`** until the input is 40. This loop does not have any initialization expression. Instead, it has been initialized outside of for loop. The variable **`num`** is increased by 1. The loop terminates when **`num`** becomes 40.

If all the three expressions are left blank, an infinite loop will be created.

The following code demonstrates the use of such loop.

Code Snippet:

```
.....
for( ; ; ) {
    System.out.println("This will go on and on");
}
.....
```

The code will print '`This will go on and on`' unless and until the loop is terminated. `break` statements can be used to terminate such loops. Such codes lead to infinite loops. Infinite loop makes the program run indefinitely for a long time resulting in the consumption of all resources and stopping the system. Thus, it is a good practice to avoid using such loops in a program.

When the number of user inputs in a program is not known beforehand, an infinite loop can be used in a program where it will wait indefinitely for user input. Thus, when a user input is received, system processes the input and again starts executing the infinite loop.

3.2.7 Nested Loops

The placing of a loop in the body of another loop is called nesting. For example, a `while` statement can be enclosed within a `do-while` statement and a `for` statement can be enclosed within a `while` statement. When you nest two loops, the outer loop controls the number of times the inner loop is executed. For each iteration of the outer loop, the inner loop will execute all of its iterations.

There can be any number of combinations between the three loops. While all types of loop statements may be nested, the most commonly nested loops are formed by `for` statements.

The following code demonstrates the use of `nested-for` loop.

Code Snippet:

```
int i, j;
for (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 2; j++)
   {
        System.out.printf("\n%d %d", i , j );
   }
}
```

The first `for` statement is executed first, and the value of ‘i’ is verified to be less than or equal to three, if true, the inner `for` loop is executed. The inner loop is executed till the condition returns false (in this case, it is run twice) before control is handed back to the outer loop. When the outer loop is run a second time, the value of ‘i’ is incremented, then verified and the inner loop is executed again two times. In this manner, the execution continues till the outer loop’s condition returns ‘false’.

Output:

```
1 1
1 2
2 1
2 2
3 1
3 2
```

3.2.8 Comparison

The type of loop that is chosen while writing a program depends on the good programming practice.

A loop written using the `while` statement can also be rewritten using the `for` statement and vice versa. The `do-while` statement can be rewritten using the `while` or the `for` statement. However, this is not advisable because the `do-while` statement is executed at least once. When the number of times the statements within the loop should be executed is known, the `for` statement is used.

Table 3.2 lists the differences between `while/for` and `do-while` loops.

while/for	do-while
Loop is pre-tested. The condition is checked before the statements within the loop are executed.	Loop is post-tested. The condition is checked after the statements within the loop are executed.
The loop does not get executed if the condition is not satisfied at the beginning.	The loop gets executed at least once even if the condition is not satisfied at the beginning.

Table 3.2: Difference Between while/for and do-while

Knowledge Check 2

1. You want the output to be displayed as '`95 91 87 83`'. Can you arrange the steps in sequence to achieve the same?

(A)	`System.out.println(i);`
(B)	`i++;}`
(C)	`int i = 100;`
(D)	`i -= 5;`
(E)	`while(i >= 85){`

2. Which of the statements about `do-while` iteration statements are true and which statements are false?

(A)	`do-while` tests the condition after the first iteration of the loop
(B)	In `do-while`, if the condition is true, the control passes back to the top of the loop
(C)	In `do-while` , if the condition is true, the loop terminates
(D)	`do-while` statements is executed at least once
(E)	`do-while` tests the condition at the beginning of the loop

3. You want to display the output as '15105'. Can you identify the correct code?

(A)	`for(int i=15;i>=0;i-=5)`
	 `    System.out.print(i);`
	 `    System.out.println("");`

(B)	`for(int i=11;i<=15;i+=5)`
	 `    System.out.print(i);`
	 `    System.out.println("");`

(C)	`for(int i=15;i>0;i-=5)`
	 `    System.out.print(i);`
	 `    System.out.println("");`

(D)	`for(int i=15;i>0;i-=6)`
	 `    System.out.print(i);`
	 `    System.out.println("");`

4. Can you match the different types of loops against their corresponding description?

	Description		Loop
(A)	Executed at least once before the condition is checked at the end of the loop	(1)	`while`
(B)	Used when the user is sure about the number of iterations required	(2)	`do-while`
(C)	Variable used in the expression is initialized before the loop starts	(3)	`for`
(D)	This control structure is known as post-test loop		

3.3 Jump Statements

In this last lesson, **Jump Statements**, you will learn to:

- State the purpose of jump statements.
- Describe `break` statement.
- Describe `continue` statement.
- Explain labeled statement.
- Compare `break` and `continue` statements.

3.3.1 Purpose

At times, the exact number of times the loop has to be executed is known only during runtime. In such a case, the condition to terminate the loop can be enclosed within the body of the loop. At other times, based on a condition, the remaining statements present in the body of the loop need to be skipped. Java supports jump statements that unconditionally transfer control to locations within a program known as target of jump statements.

Java provides two keywords: `break` and `continue` that serve diverse purposes. However, both are used with loops to change the flow of control based on conditions.

3.3.2 'break' Statement

The `break` statement in Java is used in two ways. First, it can be used to terminate a case in the switch statement. Second, it forces immediate termination of a loop, bypassing the loop's normal conditional test.

When the break statement is encountered inside a loop, the loop is immediately terminated and the program control is passed to the statement following the loop.

The following code demonstrates the use of `break` statement.

Code Snippet:

```
int cnt, number;
for (cnt=1, number=0; cnt <= 100; cnt++) {
   Scanner input = new Scanner(System.in);
   System.out.println("Enter a number");
   number = input.nextInt();
   if(number==500)
     break;
}
```

As shown in the code, the user is prompted to enter a number, and this is stored in the variable `number`. This is repeated 100 times. However, if the user enters the number 500, the loop terminates and control is passed to the next statement.

3.3.3 'continue' Statement

Java provides another keyword named `continue` to skip statements within a loop and proceed to the next iteration of the loop.

Code Snippet:

```
int cnt, square, cube;
 for (cnt=1 ; cnt <= 10; cnt++) {
   if(cnt % 3 == 0)
      continue;
   square = cnt * cnt;
   cube = cnt * cnt * cnt;
    System.out.printf("\nSquare of %d is %d and Cube is %d", cnt, square, 
cube);
   }
```

The code declares a variable **`cnt`** and uses the `for` statement which contains the initialization, termination and increment expression. In the body of the loop, the value of **`cnt`** is divided by 3 and the remainder is checked. If the remainder is 0, the `continue` statement is used to skip the rest of the statements in the body of the loop. If remainder is not 0, the `if` statement evaluates to false, and the square and cube of **`cnt`** is calculated and displayed.

Output:

```
Square of 1 is 1 and Cube is 1
Square of 2 is 4 and Cube is 8
Square of 4 is 16 and Cube is 64
Square of 5 is 25 and Cube is 125
Square of 7 is 49 and Cube is 343
Square of 8 is 64 and Cube is 512
Square of 10 is 100 and Cube is 1000
```

3.3.4 Labeled Statement

A labeled statement is used only in case of nested loops. It is used to indicate the nested loop that is to be continued with the next iteration, or the nested loop to break from. A `break` keyword, when used with a `label`, exits out of the labeled loop.

The following code demonstrates the use of `labeled break` statement.

Code Snippet:

```
.....
int i;
outer:
    for(i=0; i<5; i++) {
        if(i == 2) {
            System.out.println("Hello");
            break outer;
        }
        System.out.println("This is the outer loop.");
    }
System.out.println("Good-Bye");
.....
```

As shown in the code, the loop is supposed to be executed five times. The first two times it displays the sentence '`This is the outer loop`'. In the third pass the value of **`i`** is equal to 2, thus it enters the `if` statement and prints '`Hello`'. Next, the `break` statement is encountered and the control passes to the label `outer:`. Thus, the loop is terminated and the last statement is printed.

The output of the code will be:

```
This is the outer loop.
This is the outer loop.
Hello
Good-Bye
```

The following code demonstrates the use of `labeled continue` statement.

Code Snippet:

```
.....
outer:
    for(int i=0; i<5; i++) {
        for(int j=0; j<5; j++) {
            System.out.println("Hello");
          continue outer;
        }
        System.out.println("This is the outer loop.");
    }
    System.out.println("Good-Bye");
.....
```

In the code, '`Hello`' will be printed five times. After the continue statement is executed, the flow continues with the next iteration of the loop identified with the label. Finally, when the condition in the outer loop evaluates to false, this loop will exit and '`Good-Bye`' will be printed.

Thus, the output of the code will be as follows:

```
Hello
Hello
Hello
Hello
Hello
Good-Bye
```

3.3.5 Comparison

Table 3.3 lists the features of jump statements.

break	continue
`break` statement is used with a condition within a loop. If the condition is true, then the flow of control passes to the statement outside the loop. The remaining iterations are not executed.	`continue` statement is used with a condition within a loop. If the condition is true, then all statements succeeding continue, if there are any, are ignored and the next iteration is carried out.
`break` can be used with all loops.	`continue` can be used with all loops.

Table 3.3: Features of Jump Statements

Knowledge Check 3

1. You are using a code snippet to print the value of sum as "6". Which of the following code snippet will help you to achieve this?

(A)
```
int sum = 0;
 int number = 1;
do {
   number++;
   sum += number;
   if (sum > 4) break;
}while (number < 5);
System.out.println(sum);
```

(B)
```
int sum = 0;
 int number = 0;
do {
   number++;
   sum += number;
   if (sum > 6) break;
}while (number < 5);
System.out.println(sum);
```

(C)

```
int sum = 0;
 int number = 0;
do {
   number++;
   sum += number;
   if (sum > 4) break;
}while (number < 5);
System.out.println(sum);
```

(D)

```
int sum = 0;
 int number = 0;
do {
   number++;
   sum += number;
   if (sum > 4) break;
}while (number == 5);
System.out.println(sum);
```

Module Summary

In this module, **Decision-Making and Iterations**, you learnt about:

- **Decision-Making statements**

  The decision-making statements in Java allow us to control the flow of program execution based upon certain conditions. The `if` statement tells your program to execute a certain section of code only if a particular test evaluates to true. The `if-else` statement also provides a secondary path of execution when an "`if`" clause evaluates to false. Unlike `if` and `if-else`, the `switch` statement allows for any number of possible execution paths.

- **Loops**

  Loops allow you to execute a block of statements repeatedly. Java uses three types of loop statements: `while`, `do-while` and `for`. The `while` statement executes a block of statements as long as the condition is true. The `do-while` statement executes a block of statements at least once and then keeps going as long as a condition is true. The `for` statement executes a block of statements for a specified number of times.

- **Jump statements**

  Jump statements unconditionally transfer control to another point within the program. The `break` statement exits to the nearest enclosing `switch`, `while`, `do-while` and `for` statements. The `continue` statement starts a new iteration of the outer `while`, `do-while`, or `for` statements. The frequent use of `break` and `continue` statements are not advisable in programming.

Module 4 Introducing Classes

Module Overview

Welcome to the module, **Introducing Classes**. This module focuses on classes and methods. This module aims to provide a brief understanding of the basic concepts of classes, methods and its syntax. It also gives a brief explanation of initializers.

In this module, you will learn about:

➢ Creating Classes and Objects

➢ Instance Variables

➢ Methods

➢ Initializers

4.1 Creating Classes and Objects

In this first lesson, **Creating Classes and Objects**, you will learn to:

➢ State the syntax for declaring classes and conventions for naming them.

➢ Describe constructor and state its syntax.

➢ State the syntax of creating objects.

4.1.1 Declaring Classes

A class declaration should contain the keyword `class` and the name of the class that is being declared. Besides this, following are some conventions to be followed when naming classes:

➢ Class name should be a noun.

➢ Class name can be in mixed case, with the first letter of each internal word capitalized.

➢ Class name should be simple, descriptive and meaningful.

➢ Class names cannot be Java keywords.

Class names cannot begin with a digit. However, they can begin with a dollar ($) symbol or an underscore character.

The following is the syntax for declaring a class in Java.

Syntax:

```
class <class_name> {
    . . .
}
```

The following code demonstrates the declaration of class in Java.

Code Snippet:

```
class Employee {
// body of class
}
```

Note:

The Javadoc comments for a class are as shown:

```
 /**
  * class description
  *
  * @version 1.0
  * @authorName Robert Robbsen
  */
```

`@version` adds "Version" subheading, for classes and interfaces, with version-text to the generated docs when the `-version` option is used with `javadoc` utility. This tag holds the current version number of the software. For example, `javadoc -version -d . Employee.java` will generate java documentation in the current folder with version number 1.0 for the class Employee present in `Employee.java`.

`@authorName` adds author entry with name-text specified to the generated docs. It is generated when -author option is used. It may contain multiple tags. For example, `javadoc -author -d. Employee.java` will generate java documentation in the current folder with author name Robert Robbsen for the class `Employee` present in `Employee.java`.

4.1.2 Constructor

Constructors are methods in a class that create objects or instances of a class. Besides this, the constructors are used for initializing variables and invoking any methods that may be required for initialization.

The constructor is invoked when an object is created. They do not have return types, but do have parameters. A no-argument, default constructor is provided by the complier for any class that does not have an explicit constructor.

The following is the syntax for declaring constructor in a class.

Syntax:

```
<classname>() {
   // Initialization code
}
```

The following code demonstrates the declaration of constructor within a class.

Code Snippet:

```
// Declaring constructor for class Car
Car() {
    priceInMillion = 4;
    modelId = 103;
}
```

Note: `Javadoc` comment for a constructor comprises description of the constructor as shown here:

```
/**
 * Constructor to initialize instances of class Car.
 */
```

4.1.3 Creating Objects

An object is created using the `new` keyword. On encountering the `new` keyword, the JVM allocates memory for the object, invokes the constructor of the class and returns a reference of that allocated memory. The reference is assigned to the object.

The following is the syntax for creating an object.

Syntax:

```
<class_name> <object_name> = new <constructor_name()>;
```

The following code demonstrates the creation of object in the Java program.

Code Snippet:

```
Car objCar = new Car();
```

Note: An object can be declared without using the `new` operator as shown:

```
<class_name> <object_name>;
```

However, in this case, the object `object_name` will not point to any memory location and memory will not be allocated. Using an object, created without using the `new` operator, in the program will result in compile time error. Before using such an object, the object should be initialized using the `new` operator.

Knowledge Check 1

1. Which of the statements about the syntax for declaring classes and the naming conventions of classes are true and which statements are false?

(A)	Class names cannot be a keyword in Java
(B)	Class names can be in mixed case
(C)	Declaration of the class need not be preceded with the keyword `class`
(D)	Class names can begin with a digit
(E)	Class names can begin with dollar symbol or an underscore character

2. Which of the statements about constructor and the syntax for declaring them are true and which statements are false?

(A)	Constructor is executed when a class is created
(B)	Constructors do not have parameters but have return types
(C)	The `new` keyword creates an object of the class
(D)	Constructors have the same name as that of the class
(E)	A default no-argument constructor has to be defined for every class

4.2 Instance Variables

In this second lesson, **Instance Variables**, you will learn to:

➢ State the purpose of instance variables.

➢ State the syntax of declaring instance variables.

4.2.1 Concept of Instance Variables

Instance variables are used to store information about an entity.

Consider a scenario, wherein a car dealer wants to keep track of the price of various cars in stock. So, to store the prices of various cars, you need variables. Accordingly, you need several local variables to store prices. However, an alternative is to create a class and declare a variable named `price` in it. This enables every instance created from this class to have its own copy of price variable referred as instance variable.

Figure 4.1 demonstrates the instances of class with their own copy of instance variable.

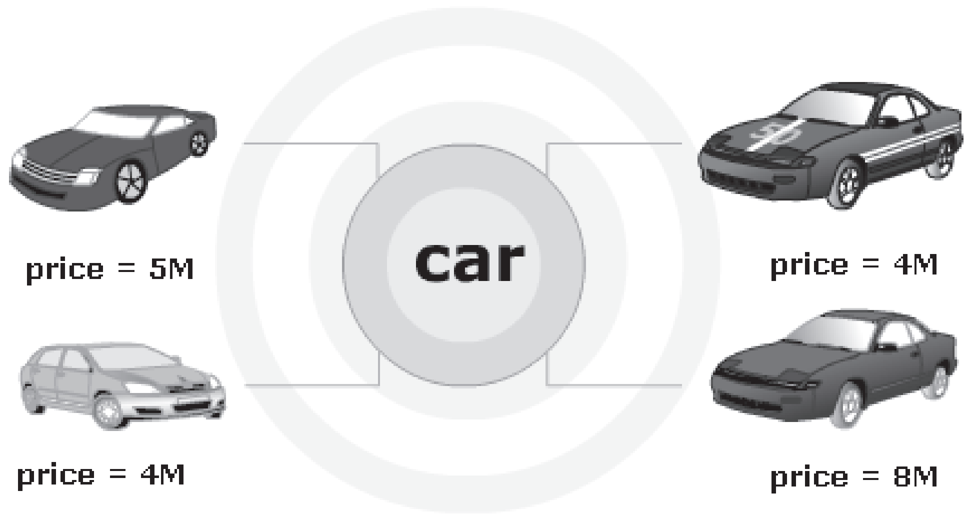


Figure 4.1: Concept of Instance variables

Note: The benefit of using instance variables over local variables is that you declare only one instance variable and use it for all instances of the class. The name of instance variable is shared across all instances, but each instance has its own copy of instance variable.

4.2.2 Declare and Access Instance Variables

Instance variables are declared in the same way as local variables. The declaration comprises the data type and a variable name.

An instance variable also has an access specifier associated with it. Instance variables are declared inside a class but outside any method definitions. They can be accessed only through an instance using the dot notation.

The following is the syntax for declaring an instance variable within a class.

Syntax:

```
[access_modifier] data_type instanceVariableName;
```

where,

`access_modifier` is an optional keyword specifying the access level of an instance variable. It could be `private`, `protected`, and `public`.

`data_type` specifies the data type of the variable.

`instanceVariableName` specifies the name of the variable.

The following code demonstrates the declaration of instance variables within a class in Java program.

Code Snippet:

```
1: class Book {
2:    int price;
3:    …
4:
5:    public static void main(String[] args) {
6:          Book objJava = new Book();
7:          obj.price = 34;
8:    }
9:    …
10: }
```

In the code, line 2 declares an instance variable named **`price`** of type `int`. Line 7 accesses the instance variable **`price`** and assigns it the value 34. Note, that to access an instance variable, you first create an instance. Next, you qualifying the instance variable name with an instance name followed by a dot.

Knowledge Check 2

1. You want to declare a class `Student` containing an instance variable `rollNo`. Further you want to create two instances of this class namely `objJohn` and `objMartin`, assign values to their respective instance variables and display them. Can you arrange the steps in sequence to achieve the same?

(A)	`objJohn.rollNo = 1151;`
	`Student objMartin = new Student();`

(B)	`public static void main(String[] args) {`
	`Student objJohn = new Student();`

(C)	`System.out.println(objMartin.rollNo);`
	`   } }`

(D)	`objMartin.rollNo = 1152;`
	`System.out.println(objJohn.rollNo);`

(E)	`class Student {`
	`int rollNo;`

4.3 Instance Methods

In this third lesson, **Instance Methods**, you will learn to:

➢ Explain the purpose of instance methods and its naming conventions.

➢ Describe variable-argument methods.

4.3.1 Concept of Instance Methods

An instance method is defined as the actual implementation of an operation on an object. It specifies the steps or the manner in which the requested operation is to be carried out.

An instance method is a method that is invoked by an instance and one that can access instance variables.

Following conventions have to be followed while naming a method:

➢ Cannot be a `Java` keyword.

➢ Cannot contain spaces.

➢ Cannot begin with a digit.

- Can begin with a letter, underscore or a “$” symbol.
- Should be a verb in lowercase.
- Should be descriptive and meaningful.
- Should be a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, and so forth.

Figure 4.2 shows the instance methods declared in the class, **`Student`** that are invoked by all the instances of the class.

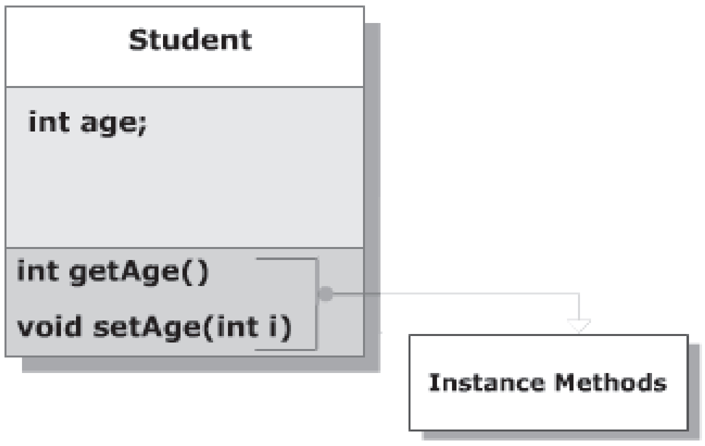


Figure 4.2: Instance Methods

The following is the syntax for declaring an instance method in a class.

Syntax:

```
<returntype> <method_name> ([list of parameters]) {
    // Body of the method
}
```

where,

`returntype`: specifies the data type of the value that is returned by the method

`method_name`: is the method name

`list of parameters`: are the values passed to the method

The following code demonstrates the declaration of instance methods within the class, **Student**.

Code Snippet:

```
// Class Declaration
class Student {
      // Instance variable
      int age;
      // Instance methods
      int getAge(){
        // Accessing instance variable
         return age;
      }
      void setAge(int i){
          age = i;
     }

     public static void main(String[] args)
     {
        Student stud = new Student();
        stud.age = 24;
       // Invoking the instance method through an object using dot operator
        int resultage = stud.getAge();
        System.out.println("Age of the student is:" + resultage);
     }
}
```

As shown in the code, **Student** class contains an instance variable, `age` and two instance methods, namely, `getAge()` and `setAge()`. The `main()` method creates an instance named `stud` for the class, Student. The instance, **stud** access the **age** variable and initializes it to 24. The instance methods can access the variables declared within the class, hence a call to `getAge()` method returns the value of variable age, that is further stored in a local variable, **resultage**.

Output:

```
Age of the student is: 24
```

Note: The Javadoc comments that are used for methods in Java are:

```
/**
 * Method description: Javadoc comments on description of the method.
 * @param tag: is followed by the name (not data type) and description of
 * the parameter.
 * @return tag:  is used to specify the return type of methods. It is not
 * used with methods having a void return type.
 */
```

Statements in the current method after the return statement are skipped, and the control returns to the statement that invoked the method. With methods that are declared `void`, use the form of `return` statement that does not return any value:

```
return;
```

Sometimes, a void method has to return explicitly depending on a condition. In this situation this form of `return` statement can be used.

For example,

```
// Assumes age is an instance variable
// in a class that is already defined
void setAge(int age){
    if (age<10) return;
        this.age=age;
}
```

4.3.2 Passing Arguments by Value

Java has the capability of passing arguments by value. This means that when the value from the calling method is passed as an argument to the called method, any changes made to that passed value in the called method will not modify the value in the calling method. Variables of primitive data types, such as `int` and `float` are passed by value. This is a safe way to pass a parameter.

The following code demonstrates the passing of arguments to the called method, **`checkVal()`** using pass by value method.

Code Snippet:

```
public class PrimitiveDatatypeClass{

    public static void main(String[] args) {
        int i = 3;
        //invoke checkVal() with i as argument
        checkVal(i);

        // print i to check its value
        System.out.println("After invoking checkVal, i = " + i);
    }

    // parameterized method
    // change value of parameter in checkVal()
    public static void checkVal(int j) {
            j=j+5;
    }
}
```

Output:

```
After invoking checkVal, i = 3
```

4.3.3 Passing Arguments by Reference

Call by reference enables the called method to change the value of the parameters passed to it from the calling method. Reference data type parameters are passed to the methods by value and not by reference. This means that when the method returns, the passed-in reference still references the same object as before. However, the values of the instance variables can be changed in the method.

When references are passed as parameters, the caller can change the values stored but not the reference variables.

The following code demonstrates the passing of arguments to the called method, **`drawLine()`** using pass by reference method.

Code Snippet:

```
public void drawLine(Line line, int pointX, int pointY) {

      line.setX(line.getX + pointX);
      line.setY(line.getY + pointY);

    //code to assign a new reference to line
    line = new Line(0, 0);
}
```

The method is invoked with the necessary arguments as follows:

```
obj.drawLine(myLine, 3, 6);
```

4.3.4 Variable Argument Methods

Variable argument allows calling a method with variable number of arguments. This is used when the number of arguments to be passed to a method is not known at the time of writing of a method.

Figure 4.3 shows the method declaration with variable number of arguments.

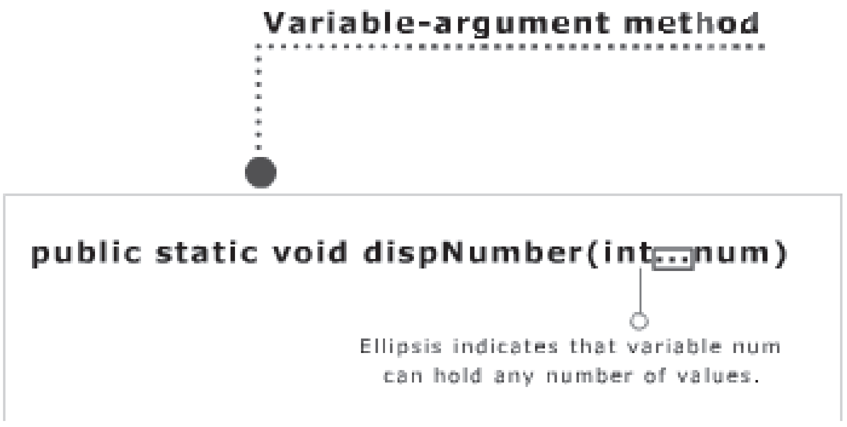

Variable-argument method
public static void dispNumber(int...num)
Ellipsis indicates that variable num can hold any number of values.

Figure 4.3: Variable Argument Methods

The following is the syntax for declaring a method with variable number of arguments.

Syntax:

```
<method_name>(type ... variableName){
// method body
}
```

The ellipsis (...) is used to identify the variable number of arguments.

The following code demonstrates the declaration of method with variable number of arguments within a class.

Code Snippet:

```
class Welcome {
    public static void dispNumber(int...num) {
        for (int i : num) {
            System.out.println("Model Number is " + i + ". ");
        }
    }
    public static void main(String[] args) {
        dispNumber(201, 301);
    }
}
```

Output:

```
Model Number is 201.
Model Number is 301.
```

The statement `for (int i:num)` is an enhanced form of for loop introduced in J2SE 5.0. This type of for loop representation is very useful for variable arguments. The statement allows i to iterate through list of variable arguments list, that is, **`num`**.

Knowledge Check 3

1. You are trying to define an instance method, display for a class named Complex. The class contains two instance variables namely **`real`** and **`imaginary`**. The method, display should accept an object of type **`Complex`** and display the values of its instance variables. Which of the following code will help you achieve this?

(A)	`void display(Complex obj) {`
	`        System.out.println(real);`
	`        System.out.println(imaginary);`
	`}`

(B)	`void display(Complex obj) {`
	`        System.out.println(obj.real);`
	`        System.out.println(obj.imaginary);`
	`}`

(C)	`Complex display(Complex obj) {`
	`      System.out.println(obj.real);`
	`      System.out.println(obj.imaginary);`
	`}`

(D)	`void display() {`
	`      Complex obj = new Complex();`
	`      System.out.println(obj.real);`
	`      System.out.println(obj.imaginary);`
	`}`

4.4 Initializers

In this last lesson, **Initializers**, you will learn to:

- Describe initializers.
- State the syntax of variable initializers.
- State the syntax of instance initializers.

4.4.1 Concept of Initializers

Initializers are small pieces of code embedded in curly braces that perform initialization. Besides using constructors for initializing, initializers can be used for initializing. The different types of initializers are as follows:

- **class block initializers**

  The class block initializer initializes complex classes. It consists of the `static` keyword, an open and close brace, and the initialization code.

- **class field initializers**

  The class field initializer initializes class variables or class fields. It evaluates an expression with an assignment operator and assigns the result to a class field before executing any of its methods.

- **object block initializer**

  The object block initializer initializes complex objects. It consists of an open and close brace and the initialization code.

- **object field initializers**

  The object field initializer initializes instance variables or object fields. It evaluates an expression with an assignment operator and assigns the result to an object field when an object is created and the constructor is called.

Note: `static` is a keyword in Java. A variable declared using the `static` keyword is a class variable and not an instance variable, such as `static int age`.

If initializers are declared, these are invoked before constructors are invoked during object instantiation.

4.4.2 Object Field Initializers

Object field initializers also known as instance variable initializers are declared inside a class. An instance variable initializer contains an equal sign and an expression.

The following is the syntax for initializing a field with an instance variable initializer.

Syntax:

```
class <classname>{
     // Object field initialization with value/expression
        <data_type> field1=<value>/<expression>
     }
```

4.4.3 Object Block Initializers

Object block initializers also known as instance initialization block are declared within the class, but outside the constructor and method definitions. An instance initializer begins with a brace bracket, followed by one or more statements and ends with a brace bracket.

The following is the syntax for declaring an instance initializer within a class.

Syntax:

```
class <classname>{

     // Object block initialization
    {
         //Initialization code
    }
}
```

The following code demonstrates the working of variable initializers and instance initializers.

Code Snippet:

```
public class MugsBeer {
  //instance field initialization
  float price=9.50F;
  String materialUsed;
  String make;
  //instance block initialization of materialUsed,make
  {
    make=”Ogilvy”;
    materialUsed=”Crystal”;
    System.out.println(“price, materialUsed & make initialized”);
  }

  MugsBeer() {
    System.out.println(“Beer Mugs Constructor”);
  }


  public static void main(String[] args) {
      MugsBeer x = new MugsBeer();
  }
}
```

Output:

```
price, materialUsed and make initialized
```

Knowledge Check 4

1. Which of the statements about initializers are true and which statements are false?

(A)	Class block initializer is used to initialize complex classes and the initialization code is preceded with `static` keyword
(B)	Class field initializer is used to initialize class variables
(C)	Object block initializer is used to initialize instance variables
(D)	Class field initialization code is preceded with a `static` keyword
(E)	Object block initializer consists of open and close brace and the initialization code

Module Summary

In this module, **Introducing Classes**, you learnt about:

- **Creating Classes and Objects**

  Constructors initialize values and invoke methods needed for initialization. Objects can be created by using the new operator.

- **Instance Variables**

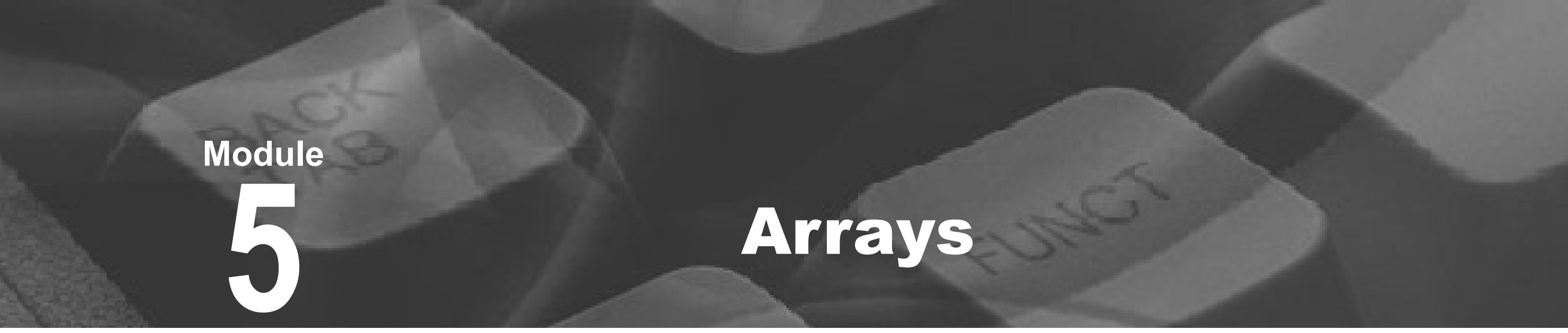
  Instance variables are declared inside a class and outside any method. An instance variable is dependent on its object.

- **Methods**

  A method specifies the steps or the manner in which the requested operation is to be carried out. Primitive data types or reference data types can be passed as parameters to the methods by value or by reference.

- **Initializers**

  Initializers are small pieces of code embedded in curly braces that perform initialization.



Module 5 Arrays

Module Overview

Welcome to the module, **Arrays**. An array is a type of storage where same type of data can be stored or maintained in consecutive memory locations. This module focuses more on single-dimensional and two-dimensional arrays. The module also covers the various classes that are used to manipulate strings. String is a class that provides various methods to manipulate characters.

In this module, you will learn about:

- Arrays
- Strings
- Other String classes

5.1 Arrays

In this first lesson, **Arrays**, you will learn to:

- Define an array.
- Explain single-dimensional arrays.
- Describe two-dimensional arrays.
- Explain searching and sorting in arrays.

5.1.1 Need of an Array

Variables can be used to store and manipulate values. Only one value can be stored in one variable at one point of time. This means that in case you need to store 20 values of the same type you need to define 20 variables for holding those values. The problem here is that a lot of memory is consumed for defining and initializing every single variable. Besides, values are not stored in the contiguous area of memory, that is, they are not stored sequentially. A single loop cannot be used to access and manipulate the values.

5.1.2 Definition

An array is a special data store that can hold several items of a single data type in contiguous memory locations.

For example, a class teacher needs to store the scores of 30 students in a class and calculate grades for each of them. The data that needs to be stored is of uniform type. This means that the scores of all the students would be stored as integers. However, it would be a tedious process to define 30 integer variables for storing the scores. This is where the concept of arrays comes in. You can declare an array of size 30 and of data type `int` that can store 30 integer values.

Figure 5.1 shows the allocation of an integer array in memory.

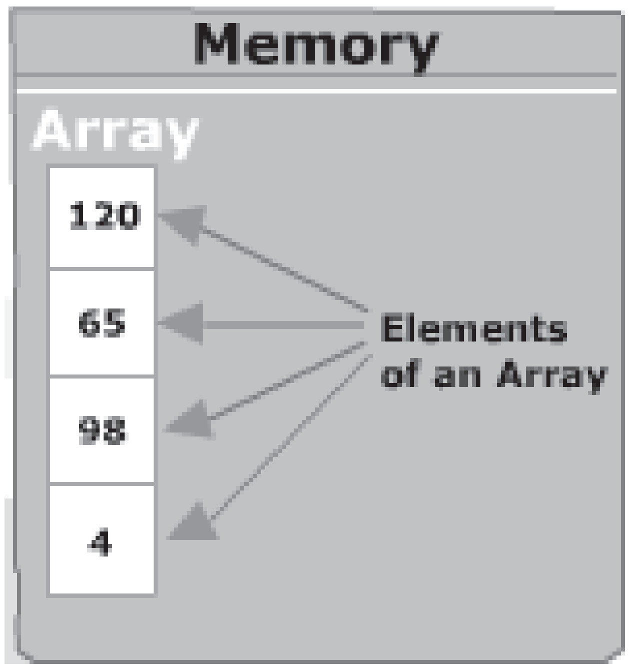


Figure 5.1: Allocation of an Array

5.1.3 Structure

Arrays are the best means of manipulating data that is available in the form of a collection. All the elements within an array must belong to the same data type.

Array have a single name and each of the elements in the array is accessible through an integer value called a subscript. Using the integer subscript, the array elements are accessed quickly and efficiently. An array subscript begins from zero and is also called an array index. The size of an array is the maximum number of elements that an array can hold. If during processing of an array, the value of array subscript exceeds this size, Java will throw runtime exception, and the program will abort immediately. Exception is an abnormal error condition in a program.

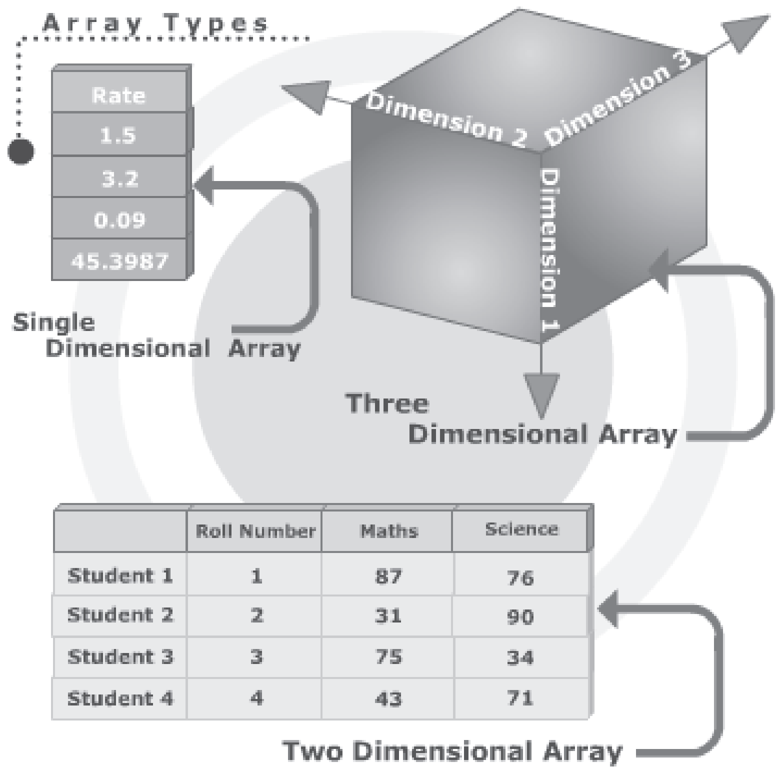
Benefits of using Arrays:

- Arrays are the best means of operating on multiple data elements of the same type at the same time.
- Arrays make optimum use of memory resource as compared to variables.
- Memory can be assigned to an array only at the time when the array is actually used. Thus, the memory is not consumed by an array right from the time you declare it.

5.1.4 Single-dimensional Arrays

Arrays in Java are of two types: single-dimensional and multi-dimensional. Single-dimensional arrays can have only one dimension and are visually represented as having several rows but a single column of data. Multi-dimensional arrays can have more than one dimension, but typically two or three dimensions are used. Memory can be allocated to an array variable by using the `new` operator.

Figure 5.2 shows the allocation of single-dimensional array in memory.



Rate
1.5
3.2
0.09
45.3987

	Roll Number	Maths	Science
Student 1	1	87	76
Student 2	2	31	90
Student 3	3	75	34
Student 4	4	43	71

Figure 5.2: Single-dimensional Array

The following is the syntax for declaring a single-dimensional array.

Syntax:

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

where,

`datatype`: any valid data type in Java

`[]`: denotes that the variable is an array

`arrayName`: array variable that will refer to the array

`new`: allocates memory for objects

`size`: number of elements that can be stored in an array

The following code shows the declaration for a single-dimensional integer array.

Code Snippet:

```
int[] studentScores = new int[25];
```

This declares an integer array with 25 elements.

Note: An array having four columns is not called a four-dimensional array; it is still a two-dimensional array. The number of subscripts needed to access elements depends upon the dimensions of an array.

5.1.5 Store and Display Values

An array can be initialized at the time of declaration or after declaration at the time of specifying the size of the array. The important point is that in order to use an array, it has to be declared and instantiated.

The various ways to initialize and display array elements are as follows:

➢ **Initialize individual array elements**

After initializing the array, the next step is to assign values to the array elements. Since, an array is a collection of multiple elements, you have to assign values to every array element specifically.

The following code demonstrates how to initialize and assign values to an array of integers called **studentScores**.

Code Snippet:

```
int[] studentScores = new int[5];
studentScores[0] = 45;
studentScores[1] = 75;
studentScores[2] = 63;
studentScores[3] = 25;
studentScores[4] = 89;
```

To assign a value to an array element, you need to specify the array element with its index.

➢ **Use Loop to Initialize**

The values for array elements can be accepted using the methods of the `Scanner` class or other ways with one or more loops as per the requirements. The `length` property can be used to find the total number of elements in an array.

The following code demonstrates how to accept values in an array using `for` loop.

Code Snippet:

```
int count; //counter variable;
 float[] itemRate = new float[3]; //array itemRate
 float total = 0; //total variable
 Scanner input = new Scanner(System.in);

 for(count = 0;count < 3;count++)
 {
    System.out.println("Enter Item rate: ");
    itemRate[count] = input.nextFloat(); //enter item rate
    total = total + itemRate[count];     //calculate total
 }

 System.out.printf("Total of all item rates is %.2f",total); //prints
total
 System.out.println("\nLength of the array is " +itemRate.length); //
prints the length
```

To hold the sum of item rates in terms of fractional numbers, an array, **`itemRate`**, is declared of type `float` and the total variable is initialized to zero. The `Scanner` class object allows you to accept values from the user with the help of the `for` loop. This loop begins from zero and accepts values as long as the condition **`count < 3`** evaluates to true.

Within the loop, the `nextFloat()` method is used to accept values from the user for item rates. The item rate adds to the current value of **`total`** to provide total value. Once the value of **`count`** reaches 3, the loop execution stops and the value of **`total`** and `length` of the array is printed.

➢ **Display Array (simple loop)**

Retrieving data from arrays is done by accessing the array elements by specifying the array index and using the value stored at that position in the array.

The following code shows how to retrieve the value of the fifth array element of an integer array called **`numArray`**.

Code Snippet:

```
int num;
num = numArray[4];
```

The following code demonstrates displaying of array values using a `for` loop.

Code Snippet:

```
int count; //counter variable;
// code for accepting array values from user
…
 //Displaying array values
 for(count = 0; count < rollNumber.length; count++)
 {
     System.out.printf("\nRollNumber:%d", rollNumber[count]);
}
```

In the code, it is assumed that an array called **`rollNumber`** is created for storing integer values. At the end of the program, all the values stored in the array are displayed to the user using a `for` loop.

➢ **Display Array (for-each loop)**

The `for-each` loop in Java 1.5 is a new way of iterating over arrays. This new `for` statement is called the enhanced `for` or `for-each` loop.

The following is the syntax of `for-each` loop.

Syntax:

```
for(type variable : array){
   statement(s);
}
```

The following code demonstrates the use of `for-each` loop in Java 1.5.

Code Snippet:

```
double sum= 0;
double[] numArray = {10.12, 67.99, 56.65, 45.43, 32.45};

System.out.println("Array Values are: ");
//for-each loop
for(double index : numArray){
   System.out.println(index);
   sum += index;
}
System.out.println("Sum = "+ sum);
```

The **`numArray`** array variable is declared and initialized with a few `double` values. The `for-each` loop is used to iterate over the array. The array value at the current subscript is displayed and the value is added to the variable, **`sum`**, with each iteration.

5.1.6 Two-dimensional Arrays

Two-dimensional arrays (2D arrays) are arrays having more than one dimension. Although it is possible to create three and higher dimensional arrays, two-dimensional arrays are commonly used.

Two-dimensional arrays too have subscripts beginning from zero onwards. Therefore, in a 2D array, the first element is stored at `arrayName[0][0]` (where [0][0] is the row and column subscript), the second element at `arrayName[0][1]` and so on. Maximum number of array elements in a 2D array is rowSize*columnSize.

Figure 5.3 shows the allocation of two-dimensional array in memory.

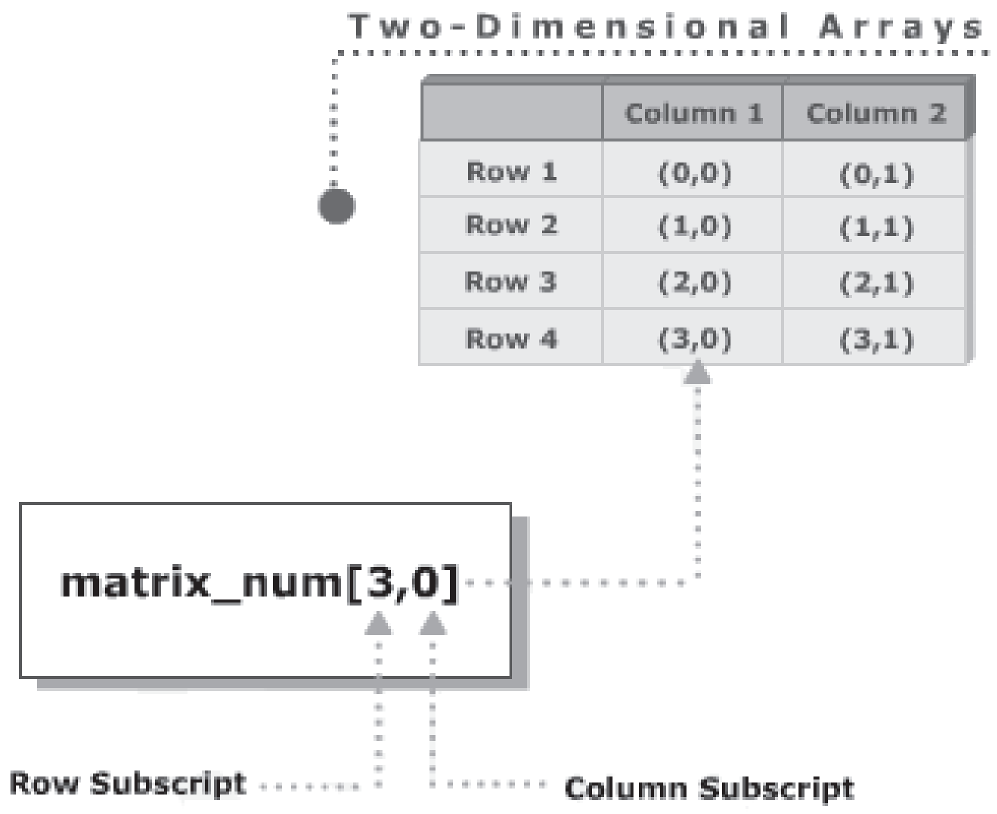


Figure 5.3: Structure

The following is the syntax for declaring a two-dimensional array.

Syntax:

```
datatype[][] arrayName = new datatype[rowsize][colsize];
```

where,

`datatype`: is any valid datatype in Java.

`rowsize` and `colsize`: are the number of rows and columns that the array will contain respectively. rowsize and colsize need not be the same. They must be positive integer values or constants containing positive integer values.

The `new` operator is used to allocate memory to the array elements.

The following code demonstrates the declaration for a 2D array.

Code Snippet:

```
int[][] matrix_num = new int[4][2];
```

This declares **matrix_num** to be an array of type `int`, which will contain maximum 4 rows and 2 columns. The maximum number of elements that can be stored in this array is 4 x 2 = 8 elements.

5.1.7 Use Loop to Initialize

An array can be initialized at the time of declaration or after declaration at the time of specifying the size of the array.

Figure 5.4 shows the accessing of an element at a particular index from a 2D array.

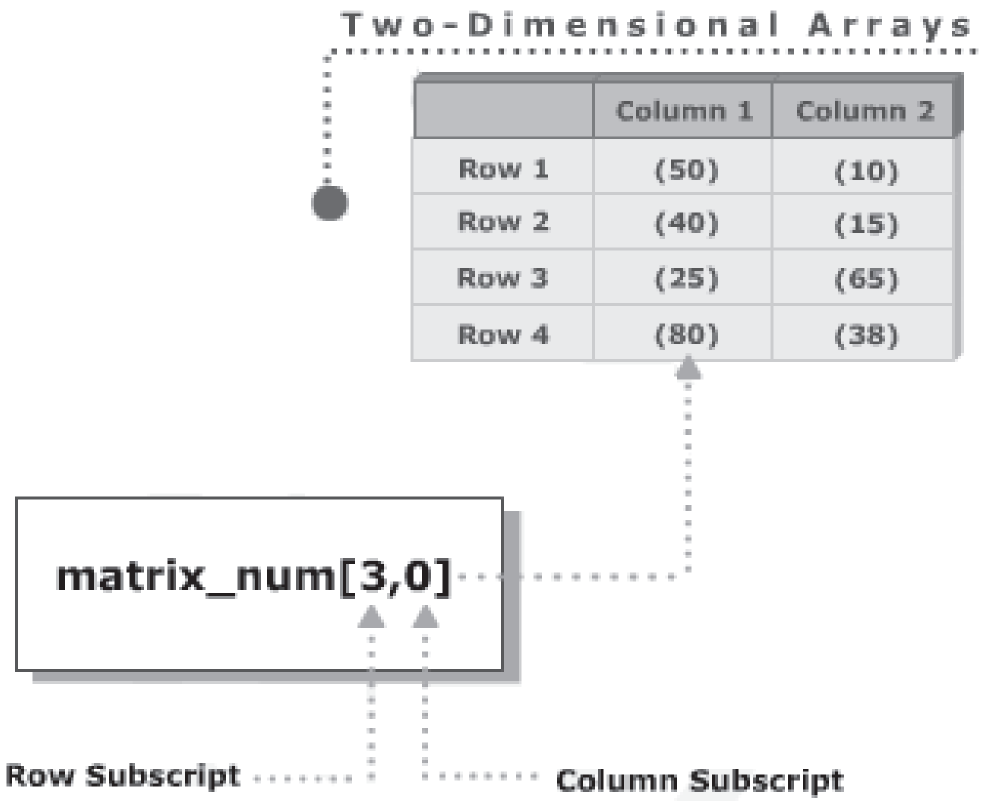


Figure 5.4: Loop Initialization

Two-dimensional arrays can be initialized in the following ways:

➢ **Assigning individual array elements**

The following code declaration assigns value for three subjects for two students in the array, **`stuMarks`**.

Code Snippet:

```
//initializes an integer array with 2 rows and 3 columns
int[][] stuMarks = new int[2][3];
stuMarks[0][0] = 55;
stuMarks[0][1] = 45;
//assignment of remaining elements
…
stuMarks[1][2] = 98;
```

➢ **At the time of declaration**

The following code snippet shows how to initialize values of an array at the time of declaration.

Code Snippet:

```
int[][] stuMarks = {{55,45,23}, {78,67,98}};
```

➢ **Using a looping construct**

The following code shows the use of loop construct to accept values for the user and storing them in the array elements.

Code Snippet:

```
int[][] stuMarks = new int[2][3]; //array with 2 rows and 3 //columns
int rowIndex = 0;
int colIndex = 0;
Scanner input = new Scanner(System.in);
int[][] stuMarks = new int[2][3]; //array with 2 rows and 3 columns
int rowIndex = 0;
int colIndex = 0;
```

```
Scanner input = new Scanner(System.in);
//row increment
for(rowIndex= 0;rowIndex < stuMarks.length;rowIndex++)
{
   //column increment
for (colIndex=0;colIndex<stuMarks[rowIndex].length;colIndex++)
   {
        System.out.println("Enter value: ");
        //accepting array values
        stuMarks[rowIndex][colIndex] = input.nextInt();
    }
}
```

The code defines a two-dimensional 2X3 array, **`stuMarks`**. Further, it declares index variable for the rows and columns of array. The code uses two `for` loops: one for the rows and one for the columns. The outer loop executes as long as the value of subscript, **`rowIndex`**, is less than two and the inner loop executes as long as the subscript value, **`colIndex`**, is less than or equal to three.

The `length` property is used to find the total number of rows and columns in a two-dimensional array. In the code snippet, if you give **`stumarks.length`**, it will print only the total number of rows. If you want to get the total number of columns, the **`stuMarks[currentrowindex].length`** can be used.

5.1.8 Display Array Values

The array elements of a two-dimensional array can be printed individually or by using loops.

The following code demonstrates the creation of an array with two rows and two columns, puts some values in it, and prints each value to standard output.

Code Snippet:

```
int[][] stuMarks = new int[2][2]; //array with 2 rows and 2 columns
stuMarks[0][0] = 45;
// remaining elements
…
stuMarks[1][1] = 86;
System.out.println("Element at row 0 column 0: "+stuMarks[0][0]);
System.out.println("Element at row 0 column 1: "+stuMarks[0][1]);
System.out.println("Element at row 1 column 0: "+stuMarks[1][0]);
System.out.println("Element at row 1 column 1: "+stuMarks[1][1]);
```

The following code assigns values to a two-dimensional array and prints it using a nested for loop.

Code Snippet:

```
int[][] stuMarks = new int[2][2]; //array with 2 rows and 2 columns
int rowIndex = 0;
int colIndex = 0;

// code for accepting values in a 2D array
…
// displaying values of a 2D array elements
 //row index
 for(rowIndex= 0;rowIndex < stuMarks.length;rowIndex++){
  //column index
    for (colIndex=0;colIndex<stuMarks[rowIndex].length;colIndex++){
       //display values
       System.out.printf("\nArray value at [%d, %d] = %d ",rowIndex,colInde
x,stuMarks[rowIndex][colIndex]);
 }
}
```

As shown in the code, the values stored in the array are displayed to the user by using a loop structure.

5.1.9 Sorting in Arrays

It is often necessary to arrange the elements in an array in numerical order from highest to lowest values (descending order) or vice versa (ascending order).

The process of sorting an array requires exchange of value. There are many different ways to sort arrays. The basic goal of all these methods is the same. They compare each array element to another array element and exchange them if they are in the wrong position.

The exchange sort process starts at the beginning of the set of values. This type of sorting compares the first element with each following element in the array and makes necessary swaps. This ends the first pass. Then it takes the second element and compares it with each following element in the array, making necessary swaps. This repeats until no swaps have occurred on the last pass.

Figure 5.5 demonstrates the concept of exchange sort.

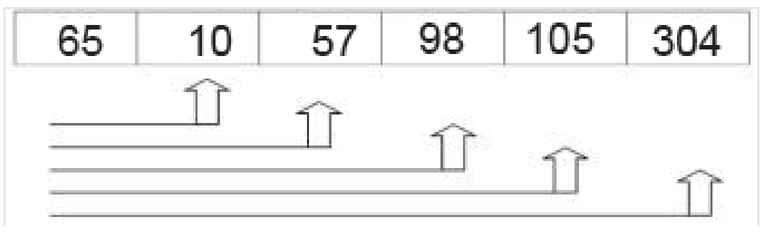


Figure 5.5: Exchange Sort

Table 5.1 shows the array elements sorted in descending order after every pass. A pass is defined as one full trip through the array comparing, and if necessary, swapping elements.

Original Array Values	65	10	57	98	105	304
After Pass 1	304	10	57	65	98	105
After Pass 2	304	105	10	57	65	98
After Pass 3	304	105	98	10	57	65
After Pass 4	304	105	98	65	10	57
After Pass 5	304	105	98	65	57	10

Table 5.1: Elements After Each Pass in Exchange Sort

The first two data items (65 and 10) are compared and the larger value is placed on the left hand side. In this case, 10 is less than 65; hence no swapping is done. Next, the first and third elements (65 and 57) are compared and here also, no swapping takes place. Then the first and fourth elements (65 and 98) are compared and the swapping is done. After the comparison is over with all the remaining elements in the array, the largest data item (304) is placed at the beginning of the list. At the end of the second pass, the second largest data item (105) will be in the second position. The process continues and the smallest item is placed at the end of the list.

The following code shows the actual Java code for sorting the elements in descending order.

Code Snippet:

```
int[] array = {65, 10, 57, 98, 105, 304};
int i, j;
int temp;
//array length
int arrayLength = array.length;
```

```
//element to be compared is identified in this loop
for (i = 0;i < (arrayLength -1);i++) {
    //represents the rest of the elements
    for(j = (i+1);j < arrayLength;j++) {
        if (array[i] < array[j]) {
            //swapping elements
            temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
}
//display array values after sorting
for(i = 0;i < array.length;i++) {
    System.out.print(array[i] + "\t");
}
```

The output of the code is displayed as follows:

```
    304             105             98              65                  57
    10
```

5.1.10 Searching in Arrays

There are various ways to search for an array value. Of all the search methods, sequential search is the simplest and easiest to implement. In sequential search, the search value is compared with the array values starting with the first element. Since the search is performed in a linear fashion, this method is also called as linear search. This is performed on an unsorted array, and is slow.

The most effective technique that can be applied to sorted records in an array is the binary search technique. This technique is faster than any other search method. This technique checks the element in the middle of the array. If the search value is equal to the middle element, the search is finished. If the search value is less than the middle element, then a binary search on the first half of the array is performed. If it is greater, then a binary search of the second half of the array is carried out. Before performing a binary search, the array must be sorted.

Figure 5.6 shows the binary search technique to search an element in a sorted array. The search value here is 11.

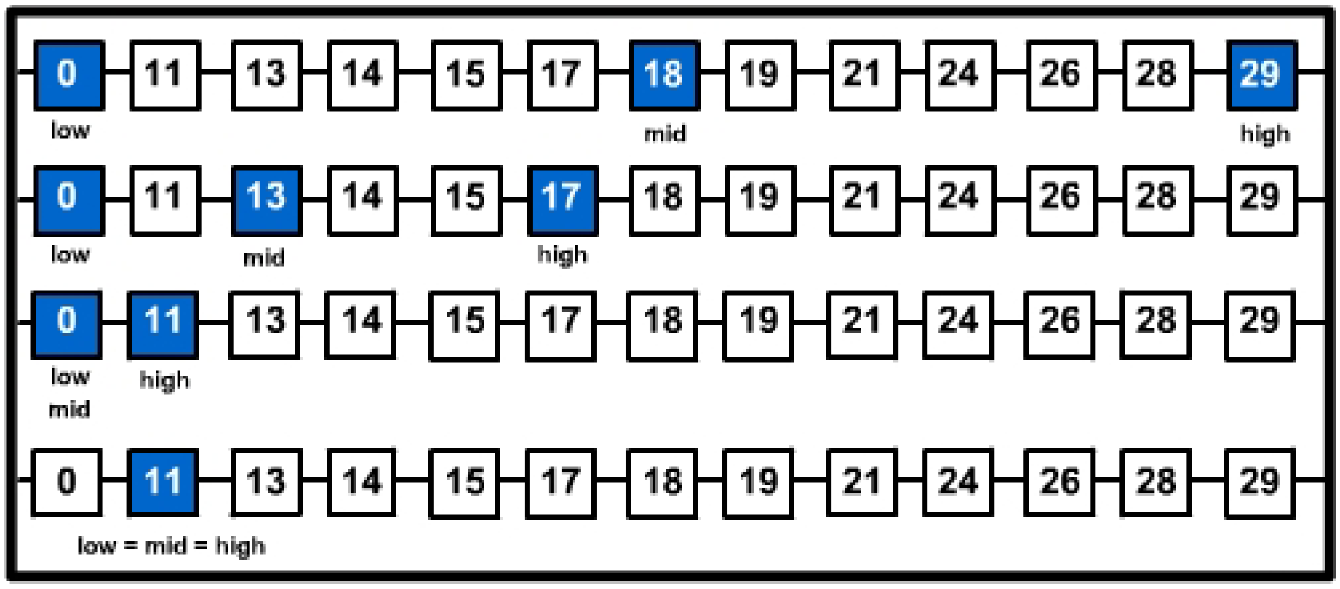


Figure 5.6: Binary Search Technique

The following code demonstrates the Java code for doing a binary search.

Code Snippet:

```
int[] array = {0, 11, 13, 14, 15, 17, 18, 19, 21, 24, 26, 28 , 29};
int low = 0;
int high = array.length - 1;
int searchValue = 11; // key value to search
int flag = 0; // flag variable to check the status
while (low <= high) {
    int mid = low + (high - low) / 2;  // Compute mid point
    if(searchValue == array[mid]){
        flag = 1;
        System.out.println("Element found at index "+ mid);
        break;
    } else if (searchValue < array[mid]) {
        high = mid - 1;  // Repeat search in first half
    }else if (searchValue > array[mid]){
        low = mid + 1;       // repeat search in second half
    } // end of else
} // end of while
if(flag == 0){

   System.out.println("Element not found in the array");
}
```

The output of the code is as follows:

```
    Element found at index 1
```

Knowledge Check 1

1. You are trying to create an array of 10 elements and store the first 10 even numbers within it and display the same. Which of the following code will help you to achieve this?

(A)	`int[] evenNumbers = new int[10];`
	`int number = 2;`
	`for(int count=0; count <10 ; count++) {`
	`   number += 2;`
	`   evenNumbers[count] = number;`
	`}`
	`for(int count=0; count <10 ; count++) {`
	`   System.out.println(evenNumbers[count]);`
	`}`

(B)	`int[] evenNumbers = new int[10];`
	
	`for(int count=0, number = 1; count <10 ; count++,number++){`
	`    number++;`
	`    evenNumbers[count] = number;`
	`}`
	`for(int count=0; count <10 ; count++) {`
	`    System.out.println(evenNumbers[count]);`
	`}`

(C)	
	```
	int[] evenNumbers = new int[10];

	for(int count=0, number = 2;count <10 ;count++,number++) {
	   evenNumbers[count] = number;
	   number = number + 2;
	}
	for(int count=0; count <10 ; count++) {
	    System.out.println(evenNumbers[count]);
	}
	```

(D)	
	```
	int[] evenNumbers = new int[10];
	int number = 0;
	for(int count=0; count <10 ; count++,number+=2) {
	   evenNumbers[count] = number;
	}
	for(int count=0; count <10 ; count++) {
	   System.out.println(evenNumbers[count]);
	}
	```

2. You are trying to compute the average of temperatures recorded for 50 cities in 365 days using a two-dimensional array temps. Average temperature is obtained by adding each day's temperature and dividing it by 365. Which of the following code will help you to achieve this?

(A)	
	```
	for (int city = 0; city < 50; city++) {
	  int total;
	  for (int day = 0; day < 365; day++){
	     total = total + temps[city][day];
	  }
	double average = total / 365.0;
	System.out.println("Average temperature for city "
	     + city  + " is " + average);
	}
	```

(B)

```
for (int city = 0; city < 50; city++) {
  int total = 0;
  double average = 0;
  for (int day = 0; day < 365; day++){
       total = total + temps[city][day];
       average = total / day;
  }
System.out.println("Average temperature for city "
        + city  + " is " + average);
}
```

(C)

```
for (int city = 0; city < 50; city++) {
  int total = 0;
  for (int day = 0; day < 365; day++){
     total = total + temps[city][day];
  }
double average = total / 365.0;
System.out.println("Average temperature for city "
     + city  + " is " + average);
}
```

(D)

```
for (int city = 0; city < 50; city++) {
  int total = 0;
  double average = 0;
  for (int day = 0; day < 365; day++){
       total = total * temps[city][day];
       average = total + day;
  }
System.out.println("Average temperature for city "
        + city  + " is " + average);
}
```

5.2 Strings

In this second lesson, **Strings**, you will learn to:

- Describe strings.
- Describe the various methods of `String` class.
- Describe a `String` array and its use.
- Describe command line arguments in Java and their uses.

5.2.1 Strings

Consider an application, that maintains employee information, such as Employee ID, name, type of work and salary details. With the help of the provided information, the possible data types for the employee information are:

The Employee ID and salary details can be represented as a numeric value. Employee name and the type of work are represented as text data.

So, what does text data actually mean? Basically, it is a collection of individual characters that form meaningful words. For example, the name of an employee is a sequential collection of characters that form some meaningful text and so is the case with the type of work. This type of text data can be stored using strings in Java. Java provides a class called `String` to represent text data.

Figure 5.7 shows the allocation of memory for storing strings.

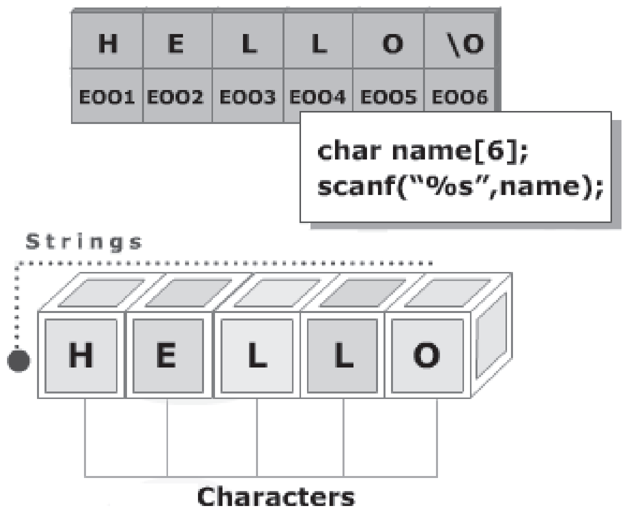


Figure 5.7: Strings

5.2.2 Create a 'String' Object

The `String` class is used to represent character strings. Strings are constant, which means their values cannot be changed after they are created.

In Java, strings are objects. An instance of a `String` class can be created with the `new` keyword.

The following code creates a new object of class `String`, and assigns it to the reference variable, **`str`**. Then, it assigns a value to the string variable.

Code Snippet:

```
String str = new String();

str = "Hello World";
```

The following code snippet demonstrates the usage of `String` objects.

Code Snippet:

```
String empName = new String(); //string object, empName
String typeOfWork = new String(); //string object, //typeOfWork

Scanner input = new Scanner(System.in);
System.out.println("Enter Employee name: ");
empName = input.nextLine(); //accepts employee name
System.out.println("Enter type of work: ");
typeOfWork = input.nextLine(); //accepts type of work

//Displaying details
System.out.println("Employee Name: "+empName);
System.out.println("Type of Work: "+typeOfWork);
```

The code accepts values into two string variables, **`empName`** and **`typeOfWork`** and prints them. The `nextLine()` method of the `Scanner` class can be used to input string values.

Output:

```
Enter Employee name: David Blake
Enter type of work: Programming
Employee Name: David Blake
Type of Work: Programming
```

5.2.3 Methods

The class `String` includes various methods to compare strings, find length of strings, remove white space characters of a string, and much more.

- **length()**

  The `length()` method can be used to find the length of a string.

  Syntax:

  ```
  int length();
  ```

  The following code demonstrates the `length()` method of `String` class.

  Code Snippet:

  ```
  String empName = new String(); //string object, empName
  Scanner input = new Scanner(System.in);
  System.out.println("Enter Employee name: ");
  empName = input.nextLine(); //accepts employee name
  //counts the number of characters including whitespaces and prints
  //total
    System.out.printf("Length of the string = %d",empName.length());
  //Displaying details
  System.out.println("Employee Name: "+empName);
  ```

  Output:

  ```
  Enter Employee name: David Blake
  Length of the string = 11
  Employee Name: David Blake
  ```

- **charAt()**

  The `charAt()` method can be used to get the character value at the specified index.

  Syntax:

  ```
  char charAt(int index);
  ```

  The index ranges from zero to `length() - 1`. The index of the first character starts at zero.

The following code demonstrates the `charAt()` method of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "David Blake";
System.out.println(empName.charAt(9)); //prints k
```

The code snippet prints the character at index 9, which is '`k`'.

- **concat()**

The `concat()` method can be used to concatenate the specified string to the end of a string.

Syntax:

```
String concat(String str)
```

If the length of the string is zero, the `String` object is returned, otherwise a new `String` object is returned.

The following code demonstrates the `concat()` method of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "David Blake";

System.out.println(empName.concat(" Martin")); //joins the two strings
```

Output:

```
David Blake Martin
```

- **compareTo()**

The `compareTo()` method compares two particular objects of `String` class and returns an integer as the result. The comparison is based on the Unicode value of each character in the strings.

Syntax:

```
int compareTo(Object anotherString)
```

The result is a negative integer if the argument string **`anotherString`** is lexicographically greater than the original string. The result is a positive integer, if the argument string **`anotherString`** is lexicographically lesser than the original string. The result is zero, if the strings are equal.

The following code demonstrates the `compareTo()` method of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "David Blake";
// returns a positive integer
System.out.println(empName.compareTo("Angelina Jack"));
//returns 0
System.out.println(empName.compareTo("David Blake"));
// returns a negative integer
 System.out.println(empName.compareTo("Elton John"));
```

Output:

```
3
0
-1
```

➢ **indexOf()**

This method returns the index of the first occurrence of the specified character or string within a string. If the character or string is not found, the method returns -1.

Syntax:

```
int indexOf(int ch)
int indexOf(String str)
int indexOf(int ch, int fromIndex)
int indexOf(String str, int fromIndex)
```

where,

`ch` is the character whose index will be returned.

str is the string whose index will be returned.

fromIndex is the index from where the indexOf method will start searching for the character ch or string str.

A value zero is returned if str is empty, but not null.

The following code demonstrates the various indexOf() methods of String class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "David Blake";

/*
  * Character 'D' at index 0
  * Character 'a' at index 1
  * Character 'v' at index 2
  * Character 'i' at index 3
  * Character 'd' at index 4
  * Character ' ' at index 5
  * Character 'B' at index 6
  * Character 'l' at index 7
  * Character 'a' at index 8
  * Character 'k' at index 9
  * Character 'e' at index 10
 */

//search starts at position 0
 System.out.println(empName.indexOf('i')); //returns 3

 //search starts at position 3
 System.out.println(empName.indexOf('a',3)); //returns 8

 //search starts at position 0
 System.out.println(empName.indexOf('a',0)); //returns 1
```

```
//search starts at position 0
System.out.println(empName.indexOf("lake")); //returns 7

//search starts at position 0
System.out.println(empName.indexOf("lake",0)); //returns 7

//search starts at position 8
System.out.println(empName.indexOf("lake",8)); //returns -1
```

The `String` class also contains methods to extract substrings, remove white space characters of a string, and much more.

- **lastIndexOf()**

The `lastIndexOf()` method is used to return the index of the last occurrence of a specified character or string within a string. The specified character or string is searched backwards that is the search begins from the last character.

Syntax:

```
int lastIndexOf(int ch)
int lastIndexOf(String str)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str, int fromIndex)
```

where,

`ch` is the character whose index will be returned.

`str` is the string whose index will be returned.

`fromIndex` is the index from where the `lastIndexOf()` method will start searching for the character `ch` or string `str`.

The following code demonstrates the `lastIndexOf()` methods of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "Java is the most elegant and the cleanest object-oriented
language available";

 //search starts from the end of the string
 System.out.println(empName.lastIndexOf('a')); //returns 72

//search starts at position 5 to zero
 System.out.println(empName.lastIndexOf('a',5)); //returns 3

//search starts at position 30 to zero
System.out.println(empName.lastIndexOf('l',30));//returns 18

//search starts at position 0
System.out.println(empName.lastIndexOf("the")); //returns 29

//search starts at position 10 to zero
System.out.println(empName.lastIndexOf("the",10));//returns 8

//search starts at position 7 to zero
System.out.println(empName.lastIndexOf("the",7));//returns -1
```

➢ **replace()**

The `replace()` method replaces all the occurrences of a specified character in the current string with a given new character. If the specified character does not exist, the reference of original String is returned.

Syntax:

```
String replace(char oldChar, char newChar);
```

The following code demonstrates the `replace()` method of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "David Blake";

//replace all 'a' with 'e'
System.out.println(empName.replace('a','e'));
```

Output:

```
Devid Bleke
```

➢ **substring()**

The `substring()` method can be used to retrieve a part of a string, that is, substring from the given string.

Syntax:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

where,

`startIndex` is the index from where the substring begins.

`endIndex` is the index where the substring ends.

The following code demonstrates the `substring()` method of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "David Blake";

System.out.println(empName.substring(2));
System.out.println(empName.substring(6,10));
```

Output:

```
vid Blake
Blak
```

➢ **toString()**

The `toString()` method can be used to return a `String` object.

Syntax:

```
String toString()
```

The following code demonstrates the `toString()` method of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "David Blake";
String newEmpName = empName.toString();
System.out.println(newEmpName.toString());
```

The code stores the value of **`empName`** string variable into another variable **`copyEmpName`**. Now, both the variables have the same value.

Output:

```
David Blake
```

➢ **trim()**

The `trim()` method returns a new string by trimming the leading and trailing whitespace from the current string.

Syntax:

```
String trim();
```

The following code demonstrates the `trim()` method of `String` class.

Code Snippet:

```
String empName = new String(); //string object, empName
empName = "\n\t\tDavid Blake.\t\n";
System.out.println(empName);
 String newEmpName = empName.trim(); //trims the whitespaces // from the
front and back of the string
 System.out.println(newEmpName);
```

The code snippet creates a new `String` variable, **`newEmpName`** that will store the same value as **`empName`**, but without whitespace characters.

Output:

```
                David Blake.


David Blake.
```

- **codePointAt()**

The `codePointAt()` method returns the character (Unicode code point) at the specified index. The key concept in Unicode is the code point. Unicode code points are just numbers. For example, in Unicode, the code point 65 has the typographic representation '`A`'.

Syntax:

```
int codePointAt(int index);
```

The following code demonstrates the `codePointAt()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
System.out.println(str.codePointAt(0));
```

Output:

```
65
```

- **codePointBefore()**

The `codePointBefore()` method returns the character (Unicode code point) before the specified index.

Syntax:

```
int codePointBefore(int index);
```

The following code demonstrates the `codePointBefore()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
System.out.println(str.codePointBefore(1));
```

Output:

```
65
```

- **codePointCount()**

The `codePointCount()` method returns the number of Unicode code points between two indices in the string.

Syntax:

```
int codePointCount(int start, int end);
```

The following code demonstrates the `codePointCount()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
System.out.println(str.codePointCount(0, 5));
```

Output:

5

- **startsWith()**

The `startsWith()` method returns a boolean value to test whether the string starts with a specified prefix.

Syntax:

```
boolean startsWith(String prefix);
```

The following code demonstrates the `startsWith()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
System.out.println(str.startsWith("Apt"));
```

Output:

true

- **endsWith()**

The `endsWith()` method returns a boolean value to test whether the string ends with a specified suffix.

Syntax:

```
boolean endsWith(String suffix);
```

The following code demonstrates the `endsWith()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
System.out.println(str.endsWith("tions"));
```

Output:

true

➢ **toUpperCase()**

The `toUpperCase()` method converts the characters in the string to upper case.

Syntax:

```
String toUpperCase();
```

The following code demonstrates the `toUpperCase()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
System.out.println(str.toUpperCase());
```

Output:

```
APTECH GLOBAL LEARNING SOLUTIONS
```

➢ **toLowerCase()**

The `toLowerCase()` method converts the characters in the string to lower case.

Syntax:

```
String toLowerCase();
```

The following code demonstrates the `toLowerCase()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
System.out.println(str.toLowerCase());
```

Output:

```
aptech global learning solutions
```

➢ **valueOf()**

The `valueOf()` method returns the string representation of the specified argument. The argument can have any one of the values: `boolean`, `char`, `float`, `double`, `int`, `long`, `char array`, or `object`.

Syntax:

```
static String valueOf(char[] data);
static String valueOf(char[] data, int offset, int count);
```

The following code demonstrates the `valueOf()` method of `String` class.

Code Snippet:

```
char[] array = {'A','p','t','e','c','h',' ','G','l','o','b','a','l'};
System.out.println(String.valueOf(array));
System.out.println(String.valueOf(array,7, 6));
```

Output:

```
Aptech Global

Global
```

- **toCharArray()**

The `toCharArray()` method copies the content of the string to a new character array.

Syntax:

```
char[] toCharArray();
```

The following code demonstrates the `toCharArray()` method of `String` class.

Code Snippet:

```
char[] array;
String str = "Aptech Global Learning Solutions";
array = str.toCharArray();
System.out.println(String.valueOf(array));
```

Output:

```
Aptech Global Learning Solutions
```

➢ **equalsIgnoreCase()**

The `equalsIgnoreCase()` method compares two strings, ignoring case, and returns a boolean value. If the strings are equal the method returns a `true` value, otherwise `false`.

Syntax:

```
boolean equalsIgnoreCase(String anotherString);
```

The following code demonstrates the `equalsIgnoreCase()` method of `String` class.

Code Snippet:

```
String str = "Aptech Global Learning Solutions";
String anotherString = "APTECH GLOBAL LEARNING Solutions";
System.out.println(str.equalsIgnoreCase(anotherString));
```

Output:

```
true
```

5.2.4 String Arrays

Sometimes, while programming, there is a need to store a collection of strings. You can create an array of string references in Java to achieve this.

The following code declares and instantiates an array of references to 10 `String` objects.

Code Snippet:

```
String[] studentNames = new String[10];
studentNames[0] = new String("David Blake");
studentNames[1] = new String("Mike Jordan");
```

The statement, `String[] studentNames = new String[10];` in the code simply sets aside memory for storage of 10 references to strings. No memory has been set aside to store the characters hat make up the individual strings. You must allocate the memory for the actual string objects separately at each index of an array, `studentNames[0] = new String("David Blake");`

Alternatively, you can declare, instantiate, and initialize an array using a single statement as follows:

```
String[] studentNames={"David Blake","John Pearson" };
```

➢ **Use a loop to initialize**

Individual components of a string array are referenced by the array name and by an integer which represents their position in the array. It is a tedious task to initialize each component individually for even a medium sized string array. It is often helpful to use a looping construct to initialize a string array.

The following code demonstrates the use of loop construct to initialize a string array.

Code Snippet:

```
//string array of 5 references
String[] studentNames = new String[5];
int count;
Scanner input = new Scanner(System.in);
 //for loop to iterate through the string array
 for (count = 0;count <studentNames.length;count++){
         System.out.println("\nEnter student name: ");
         //creating the actual string variable
         studentNames[count] = new String();
         //accepting values
        studentNames[count] = input.nextLine();
        }
```

Here, to store the names of five students, a `String` array **`studentNames`** is declared. A `for` loop is used to allocate memory to each individual array and also to accept values into the array.

➢ **Display Arrays**

To print the contents of an array, a loop construct is preferred. The '`%s`' format specifier along with the `printf()` method in Java 5.0 can be used to get a formatted output.

The following code creates a `String` array and displays its values using the `printf()` method.

Code Snippet:

```
//string array of 5 references
String[] studentNames = {
                           "David Blake",
                           "John Pearson",
                           "Mike Johnson",
                           "Bill Moore",
                           "Maria Jones" };
int count;
//display the string objects


for (count = 0;count <studentNames.length;count++){
         System.out.printf("\nStudent name: %s", studentNames[count]);
}
```

The array **`studentNames`** is declared and initialized with five string values. The strings are displayed in a `for` loop by using the `printf()` method.

Output:

```
Student name: David Blake
Student name: John Pearson
Student name: Mike Johnson
Student name: Bill Moore
Student name: Maria Jones
```

5.2.5 String Arguments

A Java application can accept any number of arguments from the operating system command line. The basic purpose of command line arguments is to specify the configuration information for the application.

The `main()` method is the entry point of a Java program, where you create objects and invoke other methods. There can be only one entry point in a Java program.

The `main()` method takes an argument, which is of the following form:

```
public static void main(String[] args){
…
} //end of main method
```

The parameter of the `main()` method is a `String` array that represents the command-line arguments.

➢ **Passing Command line arguments**

The command line arguments are entered while invoking the application. For example, you have a Java application, called `Games.class`, to execute certain statements based on the command line values. To execute the statements based on `Football` or `Hockey`, you would run it like this:

```
java Games Football Hockey
```

When the application is invoked, the values will be passed to the `main()` method as an array of strings. Each string in the array contains one of the command-line arguments. The first element, that is, `args[0]` is the first argument passed from the command line, not the name of the class. In this example, `args[0]` is `Football` not `Games.class`.

You can check for the existence of the arguments by testing the `length` property.

```
if(args.length==0) {
   System.out.println("You have not entered any arguments");
 }
```

The following code prints each of the command line arguments using a `for-each` loop.

Code Snippet:

```
public class Games {
        public Games() {
    }

    public static void main(String[] args) {
        // command line arguments
      if(args.length==0){
            System.out.println("You have not entered any arguments");
       } //end of if
     else {
        for(String str: args){
           System.out.println(str);
          } //end of for loop
     } //end of else
  } //end of main method

} //end of class Games
```

You run the application, `Games`, in the following manner:

```
java Games Football Hockey Volleyball Badminton
```

The output of the program will be:

```
Football
Hockey
Volleyball
Badminton
```

The application displays each word - `Football Hockey Volleyball Badminton` – on a line by itself. The whitespace character separates the command-line arguments. If you want to join all the arguments as a single argument, you could join them with double quotes. For that, you can run the application as follows:

```
java Games "Football Hockey Volleyball Badminton"
```

The output of the program will be:

```
Football Hockey Volleyball Badminton
```

Knowledge Check 2

1. Can you match the given methods against their corresponding descriptions?

	Description		Method
(A)	Used to get the character value at the specified index	**(1)**	lastIndexOf
(B)	Used to get the index position of the last occurrence of a specified character or string	**(2)**	indexOf
(C)	Used to get the index within a string	**(3)**	charAt
(D)	Used to compare two objects and returns an integer	**(4)**	substring
(E)	Used to retrieve part of a string from the given string	**(5)**	compareTo

2. Can you identify the correct parameter specifications for the main method?

(A)	String[] args
(B)	String args
(C)	String args[]
(D)	Strings[] args
(E)	STRING ARG
(F)	String argument
(G)	String argument()
(H)	String argument{}

5.3 Other String Classes

In this last lesson, **Other String Classes**, you will learn to:

- Explain `StringBuilder` class and its methods.
- Explain `StringTokenizer` class and its methods.

5.3.1 'StringBuilder' Class

The `StringBuilder` class provides various methods to manipulate the string object. Objects of `StringBuilder` class are growable and flexible. Characters or strings can be inserted in the `StringBuilder` object and they can also be appended at the end.

The `StringBuilder` objects can be constructed in various ways. The different forms are as follows:

- `StringBuilder()`: default constructor reserves room for 16 characters.
- `StringBuilder(int capacity)`: constructs an object with no characters in it but reserves room for characters specified in the capacity argument.
- `StringBuilder(String str)`: constructs an object that is initialized to the contents of the specified string, `str`.

5.3.2 ‘StringBuilder’ Methods

The `StringBuilder` class provides various methods for appending, inserting, deleting, and reversing strings.

➢ **append()**

The `append()` method appends values to the end of the `StringBuilder` object. This method can take different arguments, including `boolean`, `char`, `double`, `int`, `float`, `double`, and others, but the most common one will be the `String` argument.

Syntax:

```
StringBuilder append(String str)
StringBuilder append(int num)
StringBuilder append(Object obj)
```

For each append method, `String.valueOf()` method is called to convert the parameter into the corresponding string representation value and then the new string is appended to the `StringBuilder` object.

The following code snippet demonstrates the `append()` method of `StringBuilder` class.

Code Snippet:

```
StringBuilder str = new StringBuilder(“Java “);
str.append(“Programming “); //appends string “Programming”
System.out.println(str);
str.append(5.0); //appends float value “5.0”
System.out.println(str);
```

Output:

```
Java Programming
Java Programming 5.0
```

➢ **insert()**

The `insert()` method inserts one string into another. Like `append()`, it calls the `String.valueOf()` method to get the string representation of the value. The new string is inserted into the invoking `StringBuilder` object.

Syntax:

```
StringBuilder insert(int insertPosition, String str)
StringBuilder insert(int insertPosition, char ch)
StringBuilder insert(int insertPosition, float f)
```

Here, `insertPosition` indicates the position at which the new string is to be inserted to the invoking `StringBuilder` object.

The following code snippet inserts `5.0` between `Java` and `Programming`.

Code Snippet:

```
StringBuilder str = new StringBuilder("Java Programming");
str.insert(5,"5.0 „);
System.out.println(str);
```

Output:

```
Java 5.0 Programming
```

➢ **delete()**

The `delete()` method deletes characters from the invoking `StringBuilder` object.

Syntax:

```
StringBuilder delete(int start, int end)
```

Here, `start` specifies the index of the first character to remove and `end` specifies an index one character before the last character to remove. The statement, `StringBuilder delete(9, 12)` will delete characters at index 9, 10, and 11, but not 12.

The following code snippet demonstrates the `delete()` method of `StringBuilder` class.

Code Snippet:

```
StringBuilder str = new StringBuilder("Java Programming Language
5.0");
System.out.println("Before Deletion : "+ str);
str.delete(12,16); //deletes "ming"
System.out.println("After Deletion: "+str);
```

Output:

```
Before Deletion : Java Programming Language 5.0
After Deletion: Java Program Language 5.0
```

- **reverse()**

The `reverse()` method is used to reverse the characters within a `StringBuilder` object.

Syntax:

```
StringBuilder reverse()
```

The following code demonstrates the use of `reverse()` method.

Code Snippet:

```
StringBuilder str = new StringBuilder("Java Programming Language
5.0");
System.out.println("Original String : "+ str);
str.reverse(); //reversing the string
System.out.println("String after reverse: "+ str);
```

Output:

```
Original String : Java Programming Language 5.0
String after reverse: 0.5 egaugnaL gnimmargorP avaJ
```

- **charAt()**

The `charAt()` method returns the character value at the specified index.

Syntax:

```
char charAt(int index);
```

The following code snippet demonstrates the use of `charAt()` method.

Code Snippet:

```
StringBuilder   sb   =   new   StringBuilder(“Aptech   Global   Learning
Solutions”);
System.out.println(sb.charAt(7));
```

Output:

```
G
```

➢ **deleteCharAt()**

The `deleteCharAt()` method deletes a character at the specified position.

Syntax:

```
StringBuilder deleteCharAt(int index);
```

The following code snippet demonstrates the use of `deleteCharAt()` method.

Code Snippet:

```
StringBuilder   sb   =   new   StringBuilder(“Aptech   Global   Learning
Solutions”);
System.out.println(sb.deleteCharAt(5)); //deletes the char at index 5
```

Output:

Aptec Global Learning Solutions

➢ **getChars()**

The `getChars()` method copies specified number of character into an array.

Syntax:

```
void getChars(int begin, int end, char[] destArray, int destArraybegin);
```

The following code snippet demonstrates the use of `getChars()` method.

Code Snippet:

```
char[] array = new char[6];
StringBuilder   sb   =   new   StringBuilder(“Aptech   Global   Learning
Solutions”);
sb.getChars(0,6,array,0);
System.out.println(array);
```

The code copies characters from the `StringBuilder` object, **`sb`**, starting from index zero to index five. The copied characters are placed in the character array named array starting from index zero.

Output:

Aptech

- **length()**

The `length()` method returns the total number of characters from the `StringBuilder` object.

Syntax:

```
int length();
```

The following code snippet demonstrates the use of `length()` method.

Code Snippet:

```
StringBuilder   sb   =   new   StringBuilder(“Aptech   Global   Learning
Solutions”);
System.out.println(sb.length());
```

Output:

32

- **replace()**

  The `replace()` method replaces characters from the `StringBuilder` object with new characters.

  Syntax:

  ```
  StringBuilder replace(int begin, int end, String str);
  ```

  The following code snippet demonstrates the use of `replace()` method.

  Code Snippet:

  ```
  StringBuilder  sb  =  new  StringBuilder("Aptech  Global  Learning
  Solutions");
  System.out.println(sb.replace(23,32,"Services"));
  ```

  Output:

  Aptech Global Learning Services

- **setCharAt()**

  The `setCharAt()` method replaces a character from the `StringBuilder` object with a new character at the specified index.

  Syntax:

  ```
  void setCharAt(int index, char ch);
  ```

  The following code snippet demonstrates the use of `setCharAt()` method.

  Code Snippet:

  ```
  StringBuilder  sb  =  new  StringBuilder("Aptech  Global  Learning
  Solutions");
  sb.setCharAt(7,'g'); //replaces upper case 'G' with lower case 'g'
  System.out.println(sb);
  ```

  Output:

  Aptech global Learning Services

➢ **setLength()**

The `setLength()` method sets the length of the `StringBuilder` to a new value.

Syntax:

```
void setLength(int newLength);
```

If the new length is greater than the current length, all the new characters are set to null characters ('\u0000'). If the new length is less than the current length, the first `newLength` characters of the old array will be preserved, and the remaining characters will be truncated.

The following code snippet demonstrates the use of `setLength()` method.

Code Snippet:

```
StringBuilder  sb  =  new  StringBuilder("Aptech  Global  Learning
Solutions");
sb.setLength(35); // original length = 32, New Length = 35;
System.out.println(sb);
sb.setLength(13); //New Length = 13
System.out.println(sb);
```

Output:

```
Aptech Global Learning Solutions

Aptech Global
```

➢ **appendCodePoint()**

The `appendCodePoint()` method appends a Unicode code point to the `StringBuilder` object.

Syntax:

```
StringBuilder appendCodePoint(int codePoint);
```

The following code snippet demonstrates the use of `appendCodePoint()` method.

Code Snippet:

```
StringBuilder   sb   =   new   StringBuilder("Aptech   Global   Learning
Solutions");
System.out.println(sb.appendCodePoint(123));  //  Inserts  the  character
'{'
```

Output:

```
Aptech Global Learning Solutions{
```

➢ **capacity()**

The `capacity()` method returns the current capacity of the `StringBuilder` object. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

Syntax:

```
int capacity();
```

The following code snippet demonstrates the use of `capacity()` method.

Code Snippet:

```
StringBuilder sb = new StringBuilder(); // default capacity
System.out.println(sb.capacity()); //returns 16
sb = new StringBuilder("Aptech Global Learning Solutions");
System.out.println(sb.length()); // returns 32
// default capacity +  newly inserted character length
System.out.println(sb.capacity()); //returns 48
```

Output:

```
16

32

48
```

➢ substring()

The `substring()` method creates a new string from the `StringBuilder` object.

Syntax:

```
String substring(int startIndex);
String substring(int startIndex, int endIndex);
```

The second method returns a substring from the `StringBuilder` object. The substring begins at the specified `startIndex` and extends to the character at index `endIndex - 1`.

The following code snippet demonstrates the use of `subString()` method.

Code Snippet:

```
String str;
StringBuilder   sb   =   new   StringBuilder(“Aptech   Global   Learning
Solutions”);
str = sb.substring(7); // returns the substring starting at index 7
System.out.println(str);
// returns the substring starting at index 7 and ends at index 21
str = sb.substring(7, 22);
System.out.println(str);
```

Output:

```
Global Learning Solutions

Global Learning
```

5.3.3 'StringTokenizer' Class

Many text manipulation utilities require a tokenizer function which breaks up lines of text into subunits called tokens based on a specific delimiter. The most common delimiter is whitespace which yields words as the tokens. The `StringTokenizer` class in Java performs this type of task.

The `StringTokenizer` object can be created in the following ways:

- `StringTokenizer(String str)`: creates a string tokenizer for the specified string.
- `StringTokenizer(String str, String delim)`: creates a string tokenizer for the specified string based on delimiters specified by delim argument.
- `StringTokenizer(String str, String delim, boolean returnDelims)`: creates a string tokenizer for the specified string based on delimiters specified by `delim` argument. The delimiter characters are returned as tokens, if the `returnDelims` flag is true, otherwise the delimiter characters are skipped.

5.3.4 'StringTokenizer' Methods

The `StringTokenizer` class provides methods to count the number of tokens, the next element in a string, and so on.

- **countTokens()**

  The `countTokens()` method returns the number of tokens remaining in the specified string tokenizer object.

  Syntax:

  ```
  int  countTokens();
  ```

The following code snippet uses the `countTokens()` method to print the number of tokens in the `StringTokenizer` object.

Code Snippet:

```
String str = "Java Programming Language Fundamentals";

//creates a StringTokenizer object
StringTokenizer strToken = new StringTokenizer(str);

int count = strToken.countTokens(); //count the tokens
System.out.println("Number of Tokens: "+count);
```

Output:

```
Number of Tokens: 4
```

- **hasMoreElements()**

The `hasMoreElements()` method determines if there are any elements left to be read or tokenized.

Syntax:

```
boolean hasMoreElements();
```

The method returns a `boolean` value. A return value of true indicates that the string tokenizer object has more elements; a return value of false indicates that there are zero elements left to be read.

The following code snippet demonstrates the `hasMoreElements()` method in the `StringTokenizer` object.

Code Snippet:

```
String str = "Java Programming Language Fundamentals";

//creates a StringTokenizer object
StringTokenizer strToken = new StringTokenizer(str);
```

```
 //Checks whether the string tokenizer object has elements
if(strToken.hasMoreElements()) {
   System.out.println("Has more elements");
}
else {
   System.out.println("No elements");
}
```

Output:

```
Has more elements
```

➢ **hasMoreTokens()**

The `hasMoreTokens()` method tests whether the specified tokenizer's string has more than one tokens.

Syntax:

```
boolean hasMoreTokens();
```

The method returns true, if there is at least one token in the string after the current position, otherwise it returns false.

The following code snippet demonstrates the `hasMoreTokens()` method in the `StringTokenizer` object.

Code Snippet:

```
String str = "Java Programming Language Fundamentals";

//creates a StringTokenizer object
StringTokenizer strToken = new StringTokenizer(str);

//Checks whether the string tokenizer object has tokens
if(strToken.hasMoreTokens()) {
     System.out.println("Has more tokens");
}
else {
     System.out.println("No tokens");
}
```

Output:

```
Has more tokens
```

➢ **nextElement()**

The `nextElement()` method returns the next element of the specified `StringTokenizer` object.

Syntax:

```
Object nextElement();
```

The following code snippet prints the elements one by one from a `StringTokenizer` object.

Code Snippet:

```
String str = "Java Programming Language Fundamentals";

//creates a StringTokenizer object
StringTokenizer strToken = new StringTokenizer(str);

//prints the elements one by one
while(strToken.hasMoreTokens()) {
    System.out.println(strToken.nextElement());
}
```

Output:

```
Java
Programming
Language
Fundamentals
```

➢ **nextToken()**

The `nextToken()` method returns the next token from the tokenizer object.

Syntax:

```
String nextToken();
```

The following code snippet prints the tokens one by one from a `StringTokenizer` object.

Code Snippet:

```
String str = "Java Programming Language Fundamentals";

//creates a StringTokenizer object
StringTokenizer strToken = new StringTokenizer(str);
//prints the tokens one by one
while(strToken.hasMoreTokens()){
   System.out.println(strToken.nextToken());
}
```

Output:

```
Java
Programming
Language
Fundamentals
```

Knowledge Check 3

1. Can you match the methods against their corresponding descriptions?

	Description		Method
(A)	Appends values to the end of the StringBuilder object	(1)	nextElement
(B)	Inserts one string into another string	(2)	append
(C)	Tests whether a specified string tokenizer object has more tokens	(3)	nextToken
(D)	Returns an Object, the next element of the specified string tokenizer object	(4)	insert
(E)	Returns a string, the next string token from the tokenizer object	(5)	hasMoreElements

Module Summary

In this module, **Arrays**, you learnt about:

➢ **Arrays**

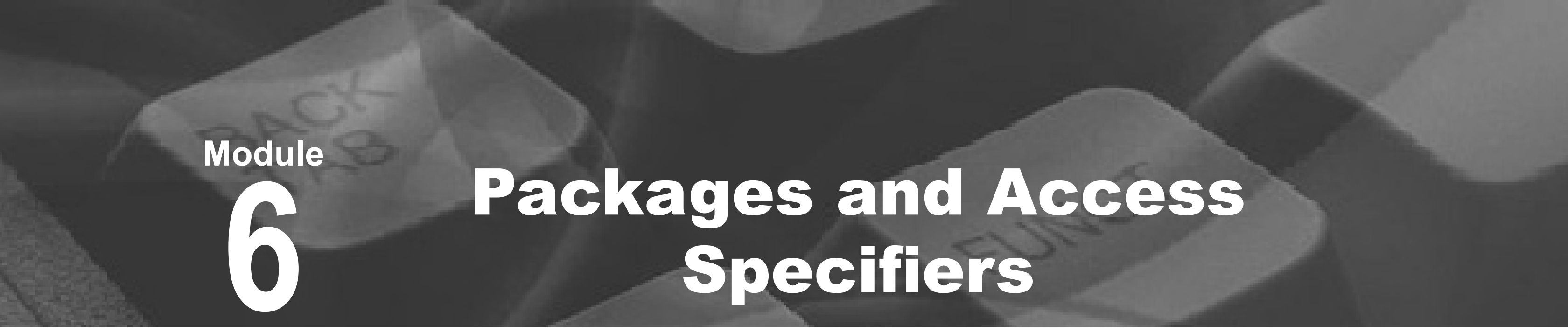
Arrays are data structures that enable you to store multiple data elements of the same type as a single entity. It is possible to assign values to individual array elements, manipulate them and retrieve their values. The array declaration statement in Java includes the data type of the array followed by its dimensions and finally, the name of the array. Java supports both single-dimensional and multi-dimensional arrays.

➢ **Strings**

Java provides the `String` class for storing and manipulating text data. This class represents an immutable string of character, that is, its value cannot be modified once it is been created. The most commonly used methods of `String` class include: `length()`, `charAt()`, `concat()`, `compareTo()`, `indexOf()`, `lastIndexOf()`, `replace()`, `substring()`, `toString()`, and `trim()`. The `string` array can be used to store a group of strings, which can be initialized upon declaration or through other methods. Command line arguments are values that are passed to the Java application when it is run. Command line arguments are usually used to set program options, or to pass along the location of a file that the program should load.

➢ **Other String classes**

The `StringBuilder` class provides various methods to manipulate a string object. Objects of `StringBuilder` class are growable and flexible. Characters or strings can be inserted in the `StringBuilder` object. They can also be appended at the end. The `StringTokenizer` class in Java is used to split a string into tokens according to a delimiter character.



Module 6 Packages and Access Specifiers

Module Overview

Welcome to the module, **Packages and Access Specifiers**. A package groups classes, interfaces, enumerations, and annotations. It also provides access protection and name space management. This module introduces the concept of packages and demonstrates how to create user-defined packages. The module also explores various Java keywords used for access control. Access specifiers or modifiers are discussed followed by field and method modifiers.

In this module, you will learn about:

- Introduction to Packages
- Access Control Keywords
- Field and Method Modifiers

6.1 Introduction to Packages

In this first lesson, **Introduction to Packages**, you will learn to:

- Identify the features of packages and their types.
- State the steps for creating and using user-defined packages.
- Explain static imports.

6.1.1 Packages

A Java package is a group of related classes and interfaces organized as one unit. Fully qualified name of the class includes the package it belongs to. For instance, `tools.drawing.Shapes` is the fully-qualified name of the class `Shape`.

A Java package has the following:

- **Features**

  Features of Java packages are as follows:

  - A package can have sub packages.

- A package cannot have two members with the same name.
- If a class or interface is bundled inside a package, it must be referenced using its fully qualified name, which is the name of the Java class including its package name.
- If multiple classes and interfaces are defined in a single Java source file within a package, then only one of them can be `public`.
- Package names are written in lowercase.
- Standard packages in the Java language begin with `java` or `javax`.

➢ **Types**

The Java API library consists of a vast set of packages. Classes and interfaces are bundled by the function and purpose they serve.

By default, every Java application or applet has access to the core package in the API, the `java.lang` package.

Types of Java packages are as follows:

- Predefined packages
- User-defined packages

Predefined packages are part of the Java API. User–defined packages are created by the developers.

Predefined packages that are commonly used are as follows:

```
java.io

java.util

java.awt
```

Note: Consider a cabinet drawer holding a set of related files and folders. If all files and folders were lumped together in one location, it would lead to an unorganized mess and chaos. This would cause waste of time while searching for a particular file. Organizing the files into relevant drawers helps to maintain orderliness and improve efficiency. A package in Java has the same concept.

6.1.2 User-Defined Packages

Java allows the user to import the classes from user-defined packages using an `import` statement.

```
import firm.Employee;//Imports the Employee class from the package firm
```

```
import firm.*;//Imports all classes from package firm
```

The steps to create a Java package have been described one by one.

Step 1

Select a name for the package.

While deciding a name for the package, naming conventions rules has to be followed and they are as follows:

➢ Package names should be written in lowercase.

  Package names should not begin with `java` or `javax`, because they are used for packages that are part of the Java API.

➢ Package names cannot begin with a digit and have hyphens, they can however have an underscore.

Step 2

Create a folder with the same name as the package.

The naming structure for a package is hierarchical, so programs containing classes and interfaces should be placed under a folder of the same name as the package.

Step 3

Place the source files in the folder created for the package.

Add the `package` statement as the first line in all the source files under that package. There can only be one `package` statement in a source file.

The following is the syntax to add the `package` statement in the source file.

Syntax:

```
package <packagename>;
```

Step 4

Compile and Execute the application.

While executing, make sure to use the fully qualified name of the class, that is, the name of the class including its package name.

The following code demonstrates how to add a `package` statement in the Java source file.

Code Snippet:

```
package firm;

public class Employee {

int empId;
String empName;
String address;

public Employee()
{
      System.out.println("Constructor of Employee");
 }

 public static void main(String args[])
     {
         Employee e=new Employee();
     }
}
```

Save the source file as `Employee.java` in a folder named `firm`. Compile the code as follows:

```
javac Employee.java
```

Or, compile the source file with `-d` option as follows:

```
javac -d . Employee.java
```

Where `-d` option stands for directory and `.` stands for current directory. The command will create a sub folder named firm and place the compiled class file inside it.

From the parent folder of the source file, execute the code with its fully qualified name:

```
java firm.Employee
```

To make use of the class elsewhere, the `import` statement is used. One example of using import statement, to import **Employee** class from the package firm is shown in the following statement:

```
    import firm.Employee; //imports a single class
```

The following code demonstrates how to use the class, **`Employee`** from the package firm in another class **`Resources`** which is in a different package.

Code Snippet:

```
package company;
import firm.Employee;
public class Resources {
    public void testMethod() {
        Employee objEmployee = new Employee();
    }
}
```

To indicate that class **`Resources`** belongs to package **`company`**, the `package` statement is added as the first line of the code. To use the class **`Employee`** in the class **`Resources`**, it needs to be imported with the `import` statement. At the time of compiling the class **`Resources`**, it is mandatory to ensure that the classpath includes the directory in which the other package (in this case, firm) is present.

All `import` statements in a class must be placed after the `package` statement and before the class declaration. The `import` and `package` statements placed in a file affect all the classes in a file and cannot be applied selectively to particular classes in a file.

In the `package` statement, names of packages and subpackages are separated by periods. Each component of the package name must be a directory name on the local machine. For example, if the `package` statement is as follows:

```
    package demo.management.list.src;
```

Then, there must be directories created in the following hierarchy:

```
    demo\management\list\src.
```

6.1.3 Static Imports

Normally, to access static members of a class within another class, it is required to use fully qualified name of the static member. This can however be relatively cumbersome, especially when many static members are used in a single statement. Java provides a workaround for this situation through static imports. The static import statement allows static members to be used without qualifying them with the class name and without inheriting from the type that contains the static members. `static imports` allow a program to import static members either individually or as a whole.

For example, if the class **`Machines`** belonging to package **`mnc.factory`** has a static member **`boltSize`** which needs to be used in the class **`Gadgets`**, then a `static import` statement for it can be written. Once this is done, **`boltSize`** can be used anywhere inside **`Gadgets`** without using a qualifying name.

The following code demonstrates the use of `static imports` statement.

Code Snippet:

```
import static mnc.factory.Machines.boltSize;
class Gadgets {
                 public void assign() {
                    boltSize = 20; }
              }
```

Static imports can also be used with built-in libraries, such as:

```
    import static java.lang.Math.PI;
    double area = PI*radius*radius; // Using static const PI
```

Though static imports is a useful feature, it should not be used too frequently. Using too many static imports in a program can hamper readability and also cause maintenance problems.

Knowledge Check 1

1. Which of these statements concerning packages are true and which statements are false?

(A)	A package can comprise of classes, interfaces and sub packages
(B)	A class when defined without any package declaration, results in a compile time error
(C)	The fully qualified name of a Java class includes its package name
(D)	A package does not provide access protection and namespace management
(E)	While creating packages, the package statement must be the first statement in a source file

2. Which of the following statements for creating and using user-defined packages are true and which of these statements are false?

(A)	Selecting a name for the package
(B)	Creating a folder with the same name as the class
(C)	Placing the source files in the folder created for the package
(D)	Copying the class files to the parent folder
(E)	Executing the class files from the root folder

6.2 Access Control Keywords

In this second lesson, **Access Control Keywords**, you will learn to:

➢ State the purpose of access modifiers.

➢ Identify the use of `public` access specifier.

➢ State the purpose of `private` access specifier.

➢ State the use of `protected` access specifier.

➢ State the use of `default` or `package` access specifier.

➢ List the rules for using access specifiers.

6.2.1 Access Modifiers

Access specifiers or modifiers are used to control the access of classes and class members. The access specifiers also determine whether classes and the members of the classes can be invoked by other classes or interfaces.

There are four levels of access in Java:

➢ public

➢ private

➢ protected

➢ default or package access

Object-oriented principles are implemented in programs through the use of access specifiers. Access control helps to prevent misuse of class details as well as hides the implementation details that are not required by other programmers. If a class or its member is allowed access, it is said to be accessible. Accessibility affects inheritance and how members are inherited by the subclass.

Table 6.1 shows the relationship between access specifiers and various elements in a Java program.

Access Specifier	Can Be Applied To				
	Data Field	Method	Constructor	Class	Interface
public	Yes	Yes	Yes	Yes	Yes
private	Yes	Yes	Yes	No	No
protected	Yes	Yes	Yes	No	No
default/no modifier (package)	Yes	Yes	Yes	Yes	Yes

Table 6.1: Access specifiers for Various Elements

Thus, as seen in table 6.1, top level classes and interfaces can have `public` or `default` access specifier. Data fields, constructors, and methods can have any one of the four access modifiers. Local variables defined inside methods are not given any access specifiers. Only class-level variables, also known as instance variables, can have access specifiers.

Note: A package is always accessible by default.

6.2.2 Using 'public' with Classes

The `public` access specifier when applied to a Java class allows the class to be accessible everywhere, even outside its package.

The following code demonstrates the declaration of class, **`Employee`** with `public` access specifier.

Code Snippet:

Program1: Employee.java

```
package firm;
public class Employee {
public Employee() {
      System.out.println("Constructor of Employee");
 }
}
```

The following code creates an object of class, **`Employee`** in another class, **`Accountant`**.

Code Snippet:

Program 2: Accountant.java

```
class Accountant {
 public static void main(String args[])     {
         firm.Employee e = new firm.Employee();
     }
}
```

In this example, **`Employee`** is declared as a `public` class in a package named **`firm`**. Since it is `public`, it is accessible everywhere. This is demonstrated by creating an object of type **`Employee`** in a class named **`Accountant`** which is not part of the same package. If the class **`Employee`** was not declared as `public` in this example, it would not have been accessible in the **`Accountant`** class.

6.2.3 Using 'public' with Members

The `public` access specifier, when applied to any member of the class allows the member to be accessible from anywhere in a program.

The following code demonstrates the declaration of members with `public` access specifier in class, **`Employee`**.

Code Snippet:

Program 1: Employee.java

```
package firm;
public class Employee {
public int empId;
public Employee(){
empId = 0;
}

public void displayId(){
      System.out.println("The Employee's id is " + empId);
             }
}
```

The following code access the `public` members of the class, **`Employee`** in another class, **`Accountant`**.

Code Snippet:

Program 2: Accountant.java

```
import java.util.*;
class Accountant {
 public static void main(String args[])     {
       firm.Employee emp = new firm.Employee();
       System.out.println("Enter employee id");
       Scanner scn = new Scanner(System.in);
       emp.empId = scn.nextInt();
       emp.displayId();
     }
}
```

In this example, **`empId`**, **`Employee()`**, and **`displayID()`** are declared as `public` members of **`Employee`** class. Thus these members can be invoked from another class named **`Accountant`**.

6.2.4 Using 'private' with Members

Data encapsulation is implemented using data hiding, which is a feature of the object-oriented (OO) approach and Java being an OO language, supports this feature through the use of the `private` access specifier.

When a class or its member is declared with a `private` access specifier, it is accessible only within its own class and not from anywhere else. The member becomes inaccessible to other classes. Typically, data that is sensitive and important and that which must not be changed in any way by others is declared as `private`.

The following code demonstrates the declaration of class, **`PayRoll`** and its members with `private` access specifier.

Code Snippet:

```
package company;
// class declaration
private class PayRoll {
  // member variable
    private int employeeId; //employee identification number
    private int netSalary; //net salary
        // constructor
        private PayRoll(int employeeId, int netSalary ) {

          //private fields can be used inside method
         this.employeeId = employeeId;
            this.netSalary = netSalary;
    }
    // member method
    private int getEmployeeId(){
             return employeeId;
     }
}
```

In this code, since the class has been declared as `private`, it is not accessible from outside the package. There must be another `public` class in the same package with the `private` class.

The member variables **`employeeId`** and **`netSalary`** are declared as `private`. These member variables can be used by methods in the same class, but not from outside the class.

The method **`getEmployeeId()`** is a private method and is not accessible from outside the class. Since, the constructor has been declared as `private`, it will not allow the creation of class instances, because it is inaccessible. In practice, constructors are rarely declared as `private`.

6.2.5 Using 'protected' with Members

When a class member is declared with a `protected` access specifier, it is accessible only within its own class and the inheriting classes. The `protected` access specifier acts like `private` access specifier, except for the subclasses.

The following code demonstrates the declaration of members as `protected` in class, **`PayRoll`**.

Code Snippet:

Program 1: PayRoll.java

```
package firm;
// Declaring class
public class PayRoll {
// constructor
 protected PayRoll() {  }
```

```
    // member variables
    protected int employeeId; //employee identification number
    private int netSalary; //net salary
    // member methods
    protected PayRoll(int employeeId, int netSalary ) {
         this.employeeId = employeeId;
         this.netSalary = netSalary;
    }
}
```

In this code, the member variable **`employeeId`** has been declared as `protected` which means it can be used in **`PayRoll`** class and it's inheriting classes. **`PayRoll`** is created inside the package firm.

6.2.6 Default or Package Access

If no modifier is specified for a class, the package access is the default access. This means that the class can be accessed by any other class in the same package.

Likewise if a class member has no modifier, the member will be considered to have package access, which is again the default access. Classes outside the package will not be able to invoke any of the members.

The following code demonstrates the behavior of `default` access specifier.

Code Snippet:

```
package firm;
// Class declaration with package access
class Pay {

    // Member variable declaration with package access
    int employeeId; //employee identification number
    int netSalary; //net salary
    // Constructor with package access
    Pay() {

          }
    //member method with package access
    int getemployeeId(){
    return employeeId;
    }
}
public class Bonus {
public static void main(String args[]){
        firm.Pay p = new firm.Pay();
        System.out.println("in Bonus");
    }
}
```

In this code, the class `Pay`, its member variables and methods have not been given any access modifiers. The package access is therefore `default` and can be used in any class within the same package. The class **`Bonus`** makes use of the class, **`Pay`**.

Table 6.2 summarizes the access specifier with their visibility within the same package and outside it.

Access Specifier	Elements Visible In			
	Class	**Package**	**Subclass**	**Outside the Package**
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No

Table 6.2: Access Specifiers and their Scope

6.2.7 Rules

Java has rules and constraints for usage of access control and they are as follows:

While declaring members, a `private` access specifier cannot be used with `abstract`, but it can be used with `final` or `static`.

- No modifier can be repeated twice in a single declaration.
- A constructor when declared `private` will be accessible in the class where it was created.
- A constructor when declared `protected` will be accessible within the class where it was created and in the inheriting classes.

Knowledge Check 2

1. Can you match the description against the correct Java access specifier?

	Description		Access Specifier
(A)	Allows members of a class to be accessible in all classes belonging to that package in which the class is defined	(1)	`private`
(B)	Allows members of a class to be accessible in that class and inheriting classes	(2)	`public`
(C)	Allows members of a class to be accessible only within the class and within methods of the class	(3)	`protected`
(D)	Allows members of a class to be accessible from any class	(4)	`default`
(E)	Allows members of a class to be accessible from another class in another package		

2. Which of the following code will generate compile-time error?

(A)

```
Class First {
    int x = 7;
}
public class Second {
    public static void main(String[] args) {
        First fst = new First();
        System.out.println(fst.x);
    }
}
```

(B)

```
class First {
    private int x = 7;
}
public class Second {
    public static void main(String[] args) {
        First fst = new First();
        System.out.println(fst.x);
    }
}
```

(C)	
	```
	class First {
	    public int x = 7;
	}
	public class Second {
	    public static void main(String[] args) {
	        First fst = new First();
	        System.out.println(fst.x);
	    }
	}
	```

(D)	
	```
	class First {
	    protected int x = 7;
	}
	public class Second {
	    public static void main(String[] args) {
	        First fst = new First();
	        System.out.println(fst.x);
	    }
	}
	```

6.3 Field and Method Modifiers

In this last lesson, **Field and Method Modifiers**, you will learn to:

- Define field and method modifiers.
- State the use of `volatile` modifiers.
- Explain the use of `native` modifiers.
- Describe the `transient` modifier and its use.
- List the rules for using access control specifiers.

6.3.1 Field and Method Modifiers

Field and method modifiers are keywords used to identify fields and methods that need to be declared for controlling access to users.

Some of these can be used in conjunction with access specifiers, such as `public` and `protected`. Java provides the `volatile`, `transient`, and `native` keywords that act as field and method modifiers.

6.3.2 'volatile' Modifier

The `volatile` modifier allows the content of a variable to be synchronized across all running threads. Therefore, when the value of the variable changes or is updated, all threads will reflect the same change.

The `volatile` modifier is applied only to fields. Constructors, methods, classes, and interfaces cannot have this modifier. This modifier is not frequently used. `volatile` variables are useful in multiprocessor environments and are not frequently used otherwise.

6.3.3 'native' Modifier

The `native` modifier is used only with methods and indicates that the implementation of the method is in a language, other than in Java.

Constructors, fields, classes, and interfaces cannot have this modifier. The methods declared using the `native` modifier are called `native` methods. The Java source file typically contains only the declaration of the `native` method and not its implementation.

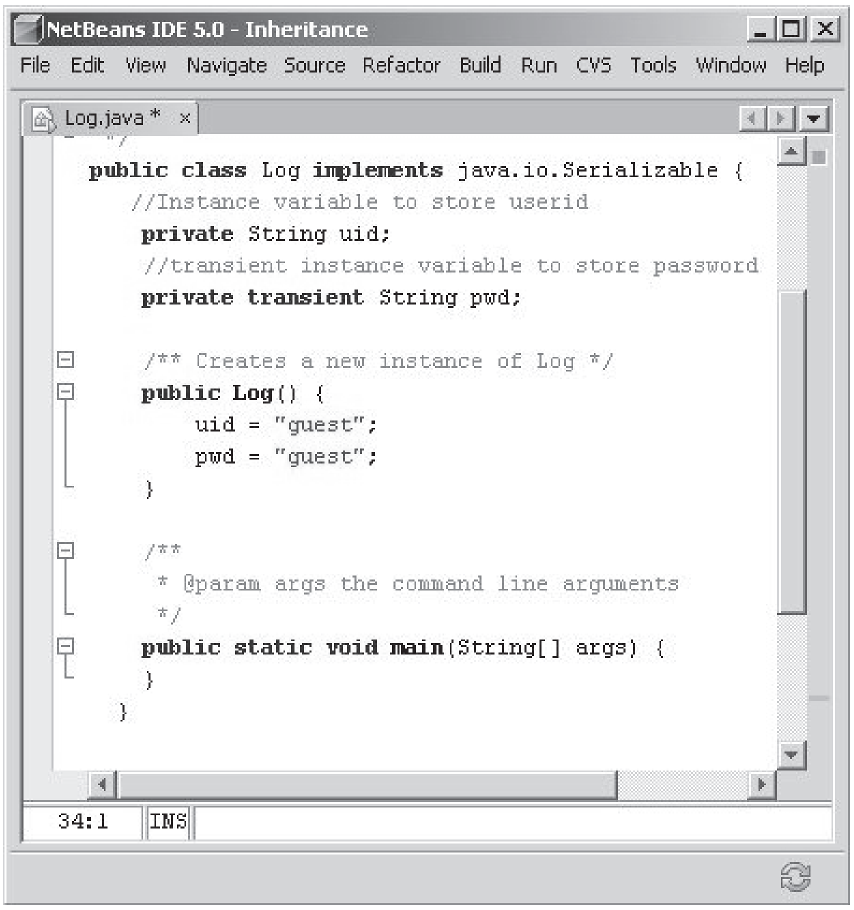
Native methods violate Java's platform independence feature, and hence must not be used, unless absolutely required. The callers of native methods need not even know that the method was declared in the class, because they invoke the method normally, just as other methods are invoked. Some classes in the Java API have native methods. The `Object` class, for instance, present in the `java.lang` package, has many `native` methods.

6.3.4 'transient' Modifier

The `transient` modifier is used to declare fields that are not saved or restored as a part of the state of the object.

Often, `serialization` proves to be quite expensive. In such scenarios, the `transient` keyword reduces the amount of data being serialized, improves performance and reduces costs.

Figure 6.1 shows the declaration of `transient` modifier.


NetBeans IDE 5.0 - Inheritance
File Edit View Navigate Source Refactor Build Run CVS Tools Window Help
Log.java *
34:1
INS

```
    public class Log implements java.io.Serializable {
       //Instance variable to store userid
        private String uid;
        //transient instance variable to store password
        private transient String pwd;

        /** Creates a new instance of Log */
        public Log() {
            uid = "guest";
            pwd = "guest";
        }

        /**
         * @param args the command line arguments
         */
        public static void main(String[] args) {
        }
    }
```

Figure 6.1: Transient Modifiers

Note: Serialization in Java converts an object's internal state into a binary stream of bytes. The stream can be written to a disk or stored in memory.

6.3.5 Rules for Using Field Modifiers

Some of the rules for using field modifiers are as follows:

- `final` fields cannot be volatile.
- `native` methods in Java cannot have a body.
- Declaring a `transient` field as `static` or `final` should be avoided as far as possible.
- `native` methods violate Java's platform independence characteristic. Therefore, they should not be used frequently.

Knowledge Check 3

1. Which of these statements about modifiers are true and which statements are false?

(A)	Field and method modifiers are used to identify fields and methods that need to be declared for controlling access to users
(B)	Field and method modifiers must be used, failing which compile time errors are caused
(C)	Some field and method modifiers can be used in conjunction with access specifiers like `public` and `protected`
(D)	`native` methods in Java must have a body
(E)	`native` methods violate Java's platform independence characteristic

2. Which of these statements about modifiers are true and which of these statements are false?

(A)	`final` fields cannot be volatile
(B)	A `transient` modifier is used to declare fields that are not saved or restored as part of the state of the object
(C)	`native` methods are not coded in Java
(D)	`volatile` modifier is applied only to constructors, methods, classes, and interfaces
(E)	`native` modifiers can be applied to constructors, fields, classes, and interfaces

Module Summary

In this module, **Packages and Access Specifiers**, you learnt about:

- **Introduction to Packages**

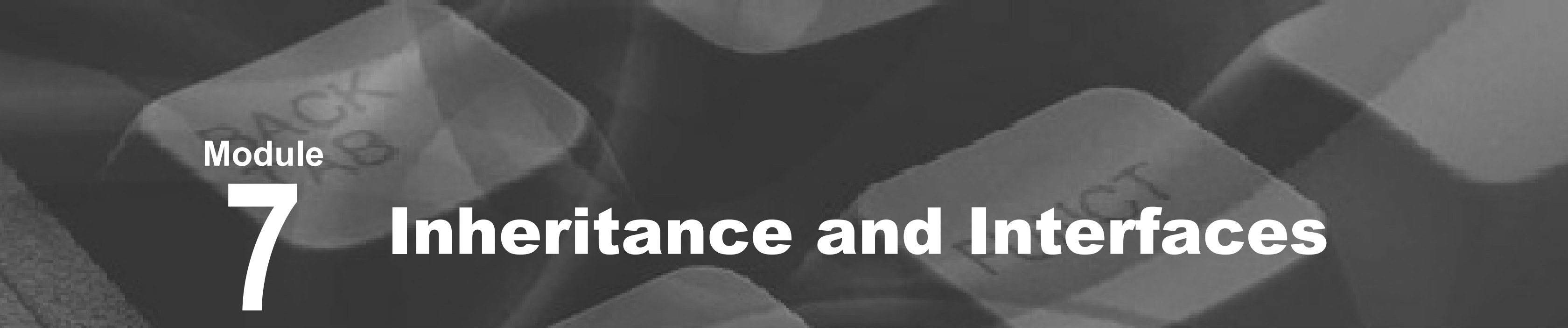
  A group of related Java classes and interfaces organized as a set is termed as a package. The Java API library consists of a vast set of packages. The package statement is used to create packages and is added as the first line in all the source files under a particular package. There can only be one package statement in a source file.

- **Access Control Keywords**

  Access specifiers or modifiers control the visibility of classes and class members. The public access specifier enables a member to be accessible everywhere. When a class member is declared with private access specifier, it is accessible only within its own class and not anywhere else. When a class member is declared with protected access specifier, it is accessible only within its own class and inheriting classes. By default, a class or its member is accessible only within its own class or package.

- **Field and Method Modifiers**

  Field and method modifiers apply access control to fields and methods. The volatile modifier enables data in a variable to be synchronized across all the threads. Hence, when the value of the value of the variable changes, all threads will see the same change. The native modifier indicates that the implementation of a method is in a language, other than Java. The transient modifier is used to declare fields that are not saved or restored as a part of the state of the object.



Module 7 Inheritance and Interfaces

Module Overview

Welcome to the module, **Inheritance and Interfaces**. This module focuses on introducing the concept of inheritance, discussing various aspects related to it, and working with interfaces. This module begins with a brief explanation of inheritance. It then describes overriding of methods and the use of a `super` keyword. Next, the overloading of methods is discussed followed by the description of the use of `abstract` and `final` keywords. Finally, the lesson ends with a discussion on interfaces.

In this module, you will learn about:

- Inheritance
- Overloading of methods
- Using `abstract` keyword
- Using `final` keywords
- Interfaces

7.1 Inheritance

In this first lesson, **Inheritance**, you will learn to:

- Explain the concept of inheritance.
- State the purpose of method overriding.
- State the use of a super keyword.
- Explain covariant return types.

7.1.1 Inheritance in Real World

In a family, traits of parents are often inherited by their children. In addition to having unique traits and behavior of their own, children inherit those of their parents.

This is evident also among other entities. For example, a sports car, a luxury car, and a family car are different types of cars.

Each has unique characteristics and behavior, but since they belong to a common category - car, they have certain common features, such as wheels, brakes, gearbox, and movement. In other words, these different categories of cars have inherited common features from the common category, car.

Figure 7.1 shows the process, whereby characteristics and behavior are transmitted from a parent to a child entity, is called inheritance.

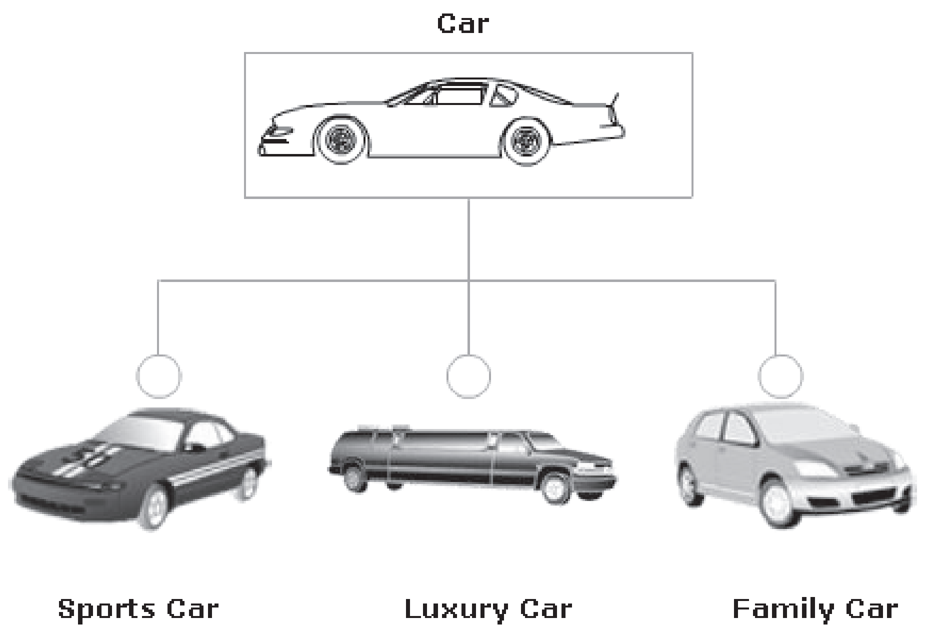


Figure 7.1: Inheritance

> **Note**: Inheritance aims to avoid creating characteristics and behavior that already exist and instead reuses the existing ones and builds on them to create new entities.

7.1.2 Inheritance in Programming

Inheritance is an important and integral part of object-oriented programming. Inheritance enables you to define a very general class first, and then define more specialized classes later by simply adding some new details to the general class definition. This saves work and encourages code reuse and customization, because the specialized class inherits all the properties and methods of the general class, only new properties and methods need to be implemented. In other words, a user can simply derive the new class from the existing class without having to write or debug the new code.

Inheritance plays a vital role in Java programming and it is imperative to understand following details and how it works.

Note: The class which inherits the fields and methods is called subclass (also called derived class or child class). The class from which the subclass inherits the fields and methods is called superclass (also called base class or parent class).

7.1.3 Subclassing

The `extends` keyword is used to create a subclass. Some rules about subclassing are as follows:

- A class can be directly derived from only one class.
- If a class does not have any superclass, then it is implicitly derived from `Object` class which is the parent of all Java classes.
- A subclass can inherit all the protected members of its superclass. The `protected` keyword is an access specifier which, when used with a member in the superclass, allows that member to be used by other classes derived from the superclass.
- Unlike other members, constructors cannot be inherited.

The following is the syntax to create a subclass in Java.

Syntax:

```
public class <class_name2> extends <class_name1> {
…
…
}
```

where,

`<class_name2>` is the name of the subclass

`<class_name1>` is the name of the superclass

The following code demonstrates the declaration of a superclass and a subclass inherited from it.

Code Snippet:

```
class Car {
public int mileage;
public String color;
protected String make;
```

```
       public void accelerate(){
       System.out.println("Car is Accelerating");
       }
}

public class LuxuryCar extends Car{

      // Luxury defines an additional feature
      // named perks
      public String perks;

      public static void main(String[] args) {
         Car objCar = new Car();
         LuxuryCar objLuxuryCar= new LuxuryCar();
         objCar.accelerate();
         System.out.println("Now inside LuxuryCar");
         // This will call the inherited
         // method accelerate
         objLuxuryCar.accelerate();
     }
}
```

Here, **`Car`** is a class having members, such as **`mileage`**, **`color`**, and **`make`**. It has one method named **`accelerate()`**. The class, **`LuxuryCar`** is derived from the class **`Car`** using the keyword `extends`, and hence becomes a subclass of the **`Car`** class. The class, **`LuxuryCar`** inherits all public and protected members of the **`Car`** class. It also defines a new member named **`perks`**. Though, the **`LuxuryCar`** class has not defined a method named **`accelerate()`**, the inherited method is executed, when a call to it is made through the object, **`objLuxuryCar`**.

Output:

```
Car is Accelerating
Now inside LuxuryCar
Car is Accelerating
```

7.1.4 Overriding

When a subclass defines a new method having the same signature as the superclass method, then the process is called overriding.

The purpose of overriding is to define new or different behavior for the objects of the subclass.

For instance, taking a real-world example, a 1970's model of a Toyota car also had a basic action of driving and a latest model of a Toyota car also has the same basic action but with greater and better performance. So, though the basic action still remains the same, how the models implement the action is different in each case.

7.1.5 Method Overriding Rules

Some rules that you should remember when overriding methods are as follows:

- The overriding method must match in name, type, number of arguments and return type with the overridden method.
- An overridden method cannot have a weaker access specifier than the access specifier of the method it overrides. For example, a protected method in a superclass can be overridden by a public method having same signature, but a public method in a superclass cannot be overridden by a protected method in the subclass.

7.1.6 'super' Keyword

Typically, when a subclass overrides a superclass method, and invokes it later, it is the subclass's method that will be called. Sometimes, it is required to call the overridden method in the superclass from within a subclass. This can be performed by using the keyword `super`. The `super` keyword can also be used to access the instance variables of the superclass.

The basic purpose of `super` keyword is to enable access to those members of the superclass that are presently hidden by members in the subclass.

The following code demonstrates the use of `super`.

Code Snippet:

```
class Car {
    public void accelerate() {
        System.out.println("Car is accelerating");
    }
}
```

```
public class LuxuryCar extends Car {

    // Overriding the method accelerate
    public  void accelerate() {
       // Calling accelerate of superclass
       super.accelerate();
       System.out.println("Now inside LuxuryCar");
       System.out.println("Luxury Car is Accelerating");
    }
    public static void main(String[] args) {
        LuxuryCar objLuxuryCar = new LuxuryCar();
        objLuxuryCar.accelerate();
    }
}
```

In this example, the class **Car** has a method **accelerate()**. Another class, the **LuxuryCar** is derived from the class, **Car**. It defines its own version of **accelerate()**, thus overriding the inherited method. Within this overridden method in the class **LuxuryCar**, it invokes the **accelerate()** method of the class **Car** using the `super` keyword. In the main method, an object is created only for the class, **LuxuryCar**. The **accelerate()** method is then invoked through this object.

Output:

```
Car is Accelerating
Now inside LuxuryCar
Luxury Car is Accelerating
```

7.1.7 'super' Keyword with Constructors

Unlike other members of a class, constructors are not inherited when a class is derived from another class. The `super` keyword is used to access the superclass's constructor. It must be the first statement in the constructor of the subclass.

The following is the syntax of `super` keyword used to access the superclass's constructor.

Syntax:

```
super([parameters]);
```

The following code demonstrates the use of `super` keyword to access superclass's constructor.

Code Snippet:

```
class Car {
    protected String make;

    public Car() {
        System.out.println("Constructor of Car invoked");
    }
    public void displayData() {
        System.out.println("Data of Car");
    }
}

public class LuxuryCar extends Car {
     public String make;
     public LuxuryCar(){
         // Accessing base class constructor
         super();
         // Accessing base class instance variable
         super.make= "Car";
         // accessing base class method
         super.displayData();
         System.out.println("Constructor of Luxury Car");
    }
    public static void main(String[] args) {
        LuxuryCar objLuxuryCar= new LuxuryCar();
    }
}
```

In the code, the class **`Car`** has a constructor. The class **`LuxuryCar`** is derived from the class **`Car`**. **`LuxuryCar`** also has a constructor and invokes the constructor of the superclass using the `super` keyword. In addition to this, instance variable and method of base class are accessed using the `super` keyword. In the main method, an object is created only for the class **`LuxuryCar`**. The **`accelerate()`** method is then invoked through this object.

Output:

```
Constructor of Car
Data of Car
Constructor of Luxury Car
```

7.1.8 Covariant Return Types

A new feature in the J2SE 5.0 allows an overriding method to return an object whose type is a subclass of the type returned by the overridden method in the superclass. This is called a covariant return type. The main advantage of a covariant return type is that it reduces type casting and type-checking to a great extent. For example, consider a superclass named **`Student`** which implements two methods named **`getMarks()`**. One method returns an instance of the `java.lang.Number` class and the other returns the `java.lang.Integer` class.

The following code demonstrates the implementation of a method with different return types.

Code Snippet:

```
class Student {
    public Number getMarks() {
        return new Number();    // Returns an object of the Number class.
    }
    public Integer getMarks() {
        return new Integer();  // Returns an object of the integer class.
    }
}
```

An attempt to compile this class will result in the following error.

```
    Student.java: 6: getMarks() is already defined in Student
        public Integer getMarks() {
                              ^
    1 error
```

The reason for this error is that if a class were to call the **`getMarks()`** method on the **`Student`** class the compiler would not know which of the two methods it should invoke. Since, `Integer` is a sub-class of the `Number` class either version of the method could be correctly called.

Now, consider that the superclass, **`Student`**, which has a single implementation of the **`getMarks()`** method. Also, consider that a subclass named **`ExchangeStudent`** overrides the same method **`getMarks()`**, which returns an object of type `Integer` which is a sub-type of the `Number` class.

The following code demonstrates overriding of superclass method in the subclass which returns a sub-type of the Number class.

Code Snippet:

```
class Student {
    public Number getMarks() {
        return new Number();    // Returns an object of the Number class.
 }
}
class ExchangeStudent {
    public Integer getMarks() {
        return new Integer();  // Returns an object of the integer class.
    }
}
```

If this code is compiled in a JDK version earlier than the Java 2 Platform, Standard Edition (J2SE) 5.0 then a compile time error occurs. However, it compiles correctly under the J2SE 5.0 JDK.

Knowledge Check 1

1. Which of the statements about the concept of inheritance are true and which statements are false?

(A)	If a class does not have any superclass, then it is implicitly derived from `Object` class which is the parent of all Java classes
(B)	The process, whereby characteristics and behavior are transmitted from a parent to a child entity, is called inheritance
(C)	Inheritance enables you to define specialized classes by adding new details to the generic parent class
(D)	Inheritance saves time and encourages code reuse and customization
(E)	A class can be derived from several classes

2. Which of the statements about the keyword super are true and which statements are false?

(A)	`super` keyword allows the method of a superclass to be invoked from within a subclass method
(B)	`super()` invokes the default constructor of the superclass
(C)	`super()` can be placed anywhere in the constructor of the subclass
(D)	The overriding method must have the same signature as the overridden method
(E)	The access specifier of overriding method and the overridden method have to public

7.2 Overloading of Methods

In this second lesson, **Overloading of Methods**, you will learn to:

- Explain the concept of overloading.
- Describe overloading using different number of parameters and same data types.
- Describe overloading using same number of parameters and different data types.
- Discuss constructor overloading.
- State the use of this keyword.

7.2.1 Concept of Overloading

Method overloading is the ability of a class to define several methods with the same name.

A generic example for overloading would be the `print()` method. The same method can be used to print a document, an image, and so forth. Instead of defining methods with different names to achieve this, it would be sensible to have one name shared across multiple methods.

Figure 7.2 shows method overloading in a class.

```
print (int number) {
   ...;
}
print (String name) {
   ...;
}
```

Figure 7.2: Overloading

7.2.2 Overloading: Different Parameter List

Overloaded methods may comprise different number of parameters and same data types.

The following code demonstrates method overloading with different parameter list.

Code Snippet:

```
public class PayRoll {
    int basicsalary;
    int hra;
    int tax;
    int netsalary;

    public void CalcSalary(int empid){
        System.out.println("Calculating salary for employee");
         netsalary= basicsalary + hra - tax;
    }

    public void CalcSalary(int empid, int leavetaken){
         System.out.println("Calculating salary for employee based on leave
taken");
        netsalary= basicsalary + hra – tax – (leavetaken * 500) ;
    }

    public void CalcSalary(int empid, int leavetaken, int incentives){
         System.out.println("Calculating salary for employee based on leave
taken and incentives");
             netsalary= basicsalary + hra - tax -(leavetaken * 500) +
incentives;
     }

    public static void main(String[] args) {
         PayRoll objPay = new PayRoll();
         objPay.CalcSalary(10);
         objPay.CalcSalary(10,4);
         objPay.CalcSalary(10,4,5000);
    }
}
```

As shown in the code, the class defines a method **`CalcSalary()`** which is overloaded a number of times. The method signatures for **`CalcSalary()`** differ in number of parameters but have the same data type, `int`. When **`CalcSalary`** is called with one `int` argument, it invokes **`CalcSalary()`** that is defined with a single `int` parameter. When **`CalcSalary()`** is called with two `int` arguments, it invokes **`CalcSalary()`** that is defined with two `int` parameters.

7.2.3 Overloading: Different Data Types

Overloaded method signatures may comprise same number of parameters and different data types.

The following code demonstrates overloading of methods with fixed number of arguments with different data types.

Code Snippet:

```
public class Printing {

    public void print(int fileId) {
        System.out.println("Printing an integer value");
    }

    public void print(String filename) {
         System.out.println("Printing a string value");
    }

    public void print(double sum){
         System.out.println("Printing a double value");
    }

    public static void main(String[] args) {
        Printing objPrint = new Printing();
        objPrint.print(101);
        objPrint.print("Welcome");
        objPrint.print(456.75);
   }
}
```

In the code, class **`Printing`** defines three **`print()`** methods, all having the same name, same number of parameters but of different data types. Each of these methods has a single parameter. However, these methods can be overloaded as they comprise different data types.

7.2.4 Constructor Overloading

Similar to methods, constructors can be overloaded. The rules for method overloading and constructor overloading are the same.

Hence, constructor overloading can be either with same number of parameters and different data types or different number of parameters and same data types. A constructor having parameters is known as parameterized constructor. Thus, constructor overloading allows multiple ways of creating instances.

The following code demonstrates constructor overloading.

Code Snippet:

```
public class Printing {

    public Printing() {
        System.out.println("Printing a document");
    }

    public Printing(int fileId) {
        System.out.println("Printing a document using fileId");
    }
}
```

As shown in the code, the two constructors are overloaded; one with no parameters and the other with one parameter.

7.2.5 'this' Keyword

The `this` keyword in Java is used in a constructor or instance method to refer to the current object in memory. The keyword allows access to a member of the current object. `this` keyword is used to access members that are presently shadowed or hidden by a local variable with the same name.

The following code demonstrates the use of `this` keyword.

Code Snippet:

```
public class Printing {
    int fileId;

    public void print(int fileId){
        this.fileId= fileId;
    }
}
```

As shown in the code, the class **`Printing`** declares a member **`fileId`**. Next, the **`print`** method accepts a parameter named, **`fileId`**. Now, it is required to assign to the member the value of the parameter that has been passed. However, as both of them have the same names, the member is shadowed or hidden by the parameter. Hence, to access the member, the `this` keyword is used.

Knowledge Check 2

1. Identify the code that overloads a method **`Draw()`**.

(A)

```
public class Shape {

    public void Draw(){
  System.out.println("drawing shape");
    }

    public void Draw(float height){
        System.out.println("drawing shape");
    }

    public void Draw(float width){
         System.out.println("drawing shape");
    }
}
```

(B)

```
public class Shape {

    public void Draw(){
  System.out.println("drawing shape");
    }

    public void Draw(float height){
  System.out.println("drawing shape");
    }

    public void Draw(int width){
  System.out.println("drawing shape");
    }
}
```

(C)

```
public class Shape {

    public void Draw(){
  System.out.println("drawing shape");
    }

    public void Draw(int height){
  System.out.println("drawing shape");
    }

    public void Draw(int width){
        System.out.println("drawing shape");
    }
}
```

(D)

```
public class Shape {

     public void Draw(){
   System.out.println("drawing shape");
     }

     public void overload Draw(float height){
   System.out.println("drawing shape");
     }

     public void overload Draw(int width){
   System.out.println("drawing shape");
     }
}
```

7.3 Using 'abstract' Keyword

In this third lesson, **Using abstract Keyword**, you will learn to:

- State the use of `abstract` methods.
- Explain the purpose of `abstract` classes.

7.3.1 'abstract' Methods

When a method has only declaration and no implementation, that is, no statements in the body, then it is called an abstract. An abstract method in Java is prefixed with the `abstract` keyword. An abstract method is only a contract that the subclass will provide its implementation.

An abstract method does not specify any implementation. The method declaration does not contain any braces and is terminated by a semicolon.

The following is the syntax for declaring abstract methods.

Syntax:

```
abstract <method return type> <methodname>([parameter list]);
```

The following code demonstrates the declaration of abstract method.

Code Snippet:

```
public abstract void DraftMemo();
```

Here, `DraftMemo()` is an abstract method. As can be seen, the `abstract` keyword is used to declare the method as abstract. The method declaration is terminated with a semicolon.

7.3.2 'abstract' Classes

A class may be required to serve as a framework that provides certain behavior for other classes. A subclass provides their application or requirement-specific behavior that adds to the behavior of the existing framework. Such, a framework is created by using the `abstract` keyword.

Abstract classes cannot be instantiated but can be inherited. The subclass must implement abstract methods declared in base class, otherwise it must be declared as `abstract`.

The following is the syntax for declaring class as abstract.

Syntax:

```
abstract class <classname> {

[abstract <method return type> <methodname>();]
. . .
}
```

The following code demonstrates the declaration of abstract class.

Code Snippet:

```
abstract class OutlineProposal {

    public abstract void DraftMemo();
    public void CalcCosts(){
        System.out.println("Costs");

    }
}
```

As shown in the code, **`OutlineProposal`** is an abstract class. It defines a framework for other classes deriving from it to follow. **`DraftMemo()`** is an abstract method declared in the class, **`OutlineProposal`**. The implementation of this method varies for each subclass derived from the class, **`OutlineProposal`**.

Knowledge Check 3

1. You want to declare an abstract class **`Shape`** having abstract method **`area()`**. Further, you provide an implementation of abstract method **`area()`** in a subclass **`Square`**. The `main()` method creates an instance of **`Square`** class and invokes its **`area()`** method. Can you arrange the steps in sequence to achieve the same?

(1)	`public static void main(String[] args) {`
	`Square obj = new Square();  obj.side = 10;`

(2)	`class Square extends Shape {`
	`int side;`

(3)	`void area() {`
	`System.out.println("Area= " + side * side); }`

(4)	`abstract class Shape {`
	`abstract void area();  }`

(5)	`obj.area();   }`
	`}`

7.4 Using 'final' Keyword

In this fourth lesson, **Using final Keyword**, you will learn to:

- Describe `final` variables.
- Describe `final` methods.
- State the purpose of `final` classes.

7.4.1 'final' Variables

Many programming languages have a specific keyword to define constant identifiers or to hold values that are not likely to change during the execution of the program. In Java, the `final` keyword is used with variables to indicate that they are constant identifiers.

Constant variables are declared with the `final` keyword and are assigned a value at the time of declaration. A compile time error is raised if a `final` variable is reassigned a value in a program after its initial declaration. The `final` variables indicate that the values will not change anytime later.

The following is the syntax for declaring `final` variable.

Syntax:

```
public final <data type> <variable name> =  <value>;
```

The following code demonstrates the declaration of `final` variable.

Code Snippet:

```
public final int MAX_COLS = 100;
```

Here, a variable named `MAX_COLS` is declared as `final` and initialized with a value of **`100`**.

7.4.2 'final' Methods

To prevent a method from being overridden or hidden in a Java subclass, it is declared as `final`. A method should be declared as `final` if a change in implementation affects the consistent state of the object.

Methods that are declared private or part of the `final` class are implicitly `final`. The `final` methods can be invoked just like any other method. The `final` methods cannot be declared as `abstract`.

The following is the syntax for declaring `final` method.

Syntax:

```
<access specifier> final <return type> method_name([parameters]) {
    …
}
```

The following code demonstrates the declaration of `final` method.

Code Snippet:

```
public class Proposal {
    final void finalMemo() {
          System.out.println("This is the final memo");
    }
}
```

Here, **`finalMemo()`** is a `final` method and cannot be overridden or hidden by any other method.

7.4.3 Final Classes

A class that cannot be subclassed is called a `final` class in Java. The `final` keyword is applied to the class declaration to achieve this.

A primary reason why classes may be declared as `final` would to be to limit extensibility and to prevent the modification of the class definitions. Once a class is defined as `final`, other classes cannot be derived from it, and hence cannot be modified. A `final` class may or may not have `final` methods. It is not necessary for a class having a `final` method to be declared as `final`. The `final` classes can be instantiated, that is, a `final` class cannot have any instances of its own.

The following is the syntax for declaring `final` class.

Syntax:

```
<access specifier> final class <class name>{ }
```

The following code demonstrates the declaration of `final` class, **`Specifications`**.

Code Snippet:

Programmer A defines a class as follows:

```
final class Specifications{
    public void beginText(){
         System.out.println("Beginning Text of the specifications");
    }
    public void bodyText(String text){
        System.out.println(text);
    }

    public void endText(){
        System.out.println("Ending Text of the specifications");
    }
}
```

The following code demonstrates the initialization of `final` class, **`Specifications`**.

Code Snippet:

Programmer B defines another class which makes use of the earlier class:

```
public class MainDisplay {
    public static void main(String args[]) {
        Specifications objSpec = new Specifications();
        System.out.println("Enter text for the specifications");
        Scanner scan = new Scanner(System.in);
        String text = scan.next();
        objSpec.beginText();
        objSpec.bodyText(text);
        objSpec.EndText();
    }
}
```

Here, Programmer B cannot modify the class, **`Specifications`** in any way but can invoke its methods. Thus, the class defined by Programmer A acts like a read-only class.

Knowledge Check 4

1. You are trying to declare a class **`JavaProgrammer`** such that its method **`setExperience()`** cannot be overridden and its integer constant **`MIN _ EXPERIENCE`** is set to two. Which of the following codes can help you achieve the same?

(A)	`class JavaProgrammer {`
	`    final MIN_EXPERIENCE = 2;`
	
	`    abstract void setExperience(int exp) {`
	`        this.experience = exp;`
	`    }`
	`}`

(B)

```
class JavaProgrammer {
     final int MIN_EXPERIENCE = 2;

     final void setExperience(int exp) {
          this.experience = exp;
     }
}
```

(C)

```
final class JavaProgrammer {
     final int MIN_EXPERIENCE = 2;

     void setExperience(int exp) {
          this.experience = exp;
     }
}
```

(D)

```
class JavaProgrammer {
     final float MIN_EXPERIENCE = 2;

     void final setExperience(int exp) {
          this.experience = exp;
     }
}
```

7.5 Interfaces

In this last lesson, **Interfaces**, you will learn to:

- ➢ Discuss the concept of interfaces.
- ➢ Describe how to use interfaces.
- ➢ Explain extending interfaces.
- ➢ Explain IS-A relationship.

7.5.1 Introduction to Interfaces

An interface in Java is a contract that lays down rules to be followed by the types which implement it.

Consider a new employee who joins an organization to work with them usually signs a contract which states that he/she will abide by the rules laid down by the organization and comply with their guidelines.

In the programming world too, a contract can be defined for classes, such that once they accept the contract they will abide by it. Such a contract is an interface.

Note: Though Java does not support multiple inheritance, interfaces provide a workaround by allowing classes to implement one or more interfaces in addition to inheriting a class. This can act similar to multiple inheritance.

7.5.2 Implementing Interfaces

An interface in Java is defined as a reference type and is similar to a class except that it has only `final` and `static` variables, `abstract` method signatures. An interface cannot be instantiated. It can be implemented by other classes or extended by other interfaces. When an interface is implemented by a class, all the variables of the interface will act as constant variables for the class. In other words, they cannot be changed by the implementing class.

A class that implements an interface is required to provide implementations for all the methods of the interface or else should be declared `abstract`. For example, if an interface has four methods, and a class implements only three methods, then the class must be declared as `abstract`.

The following is the syntax for declaring an interface.

Syntax:

```
<access_specifier> interface <interface_name> {
  // static final variable declarations
  // abstract method declarations
}
```

The following code demonstrates the implementation of an interface, **`Numbers`**.

Code Snippet:

```
interface Numbers {
      final int maxnum=3;
     public void printNums();
}
```

```
// Class implementing interface, Numbers
class IntegerNum implements Numbers {

     public void printNums(){
         for(int i = 0;i < maxnum; i++)
             System.out.println("Value of number is " + i);
         }
}class FloatNum implements Numbers {

     public void printNums(){
             for(float i = 0.0f; i<maxnum; i++)
             System.out.println("Value of number is " + i);
     }
}

class DisplayNum {
    public static void main(String args[]) {
       IntegerNum obj1 = new IntegerNum();
      obj1.printNums();
       FloatNum obj2 = new FloatNum();
      obj2.printNums();
    }
}
```

As shown in the code, the interface **`Numbers`** is implemented by two classes, **`IntegerNum`** and **`FloatNum`**. These two classes provide an implementation for the **`printNums()`** method which is declared in the interface.

Output:

```
Value of number is 0
Value of number is 1
Value of number is 2
Value of number is 0.0
Value of number is 1.0
Value of number is 2.0
```

7.5.3 Interfaces can be Extended

In Java, a simple class can extend another class, which in turn extends a third class and so on. This concept known as inheritance can also be used for interfaces. An interface is an abstract class. Hence, an interface can also extend another interface just like a class extends another java class. In fact, the Java language exhibits multiple-inheritance only in case of interfaces. This means that a single interface can implement more than one interface.

The following code demonstrates extending of an interface.

Code Snippet:

```
interface Game extends Player, Animation {
      // Variables and method declarations of both Player and Animation
interfaces available here.
}
```

Here, the interface **`Game`** implements the interfaces **`Player`** and **`Animation`** both. Hence, it can be said that the interface **`Game`** inherits the methods of the interfaces **`Player`** and **`Animation`** both.

7.5.4 IS-A Relationship

It is a concept based on class inheritance or interface implementation. An IS-A relationship expresses class hierarchy in case of class inheritance. For example, if a class **`Ferrari`** extends the class **`Car`**, then the statement 'Ferrari IS-A Car is true'. If the class **`Car`** itself extends another class **`Vehicle`**, then the relationship 'Ferrari IS-A Vehicle' is also true.

Figure 7.3 depicts an IS-A relationship which can be interpreted as an arrow in a graphic depicting class hierarchy.

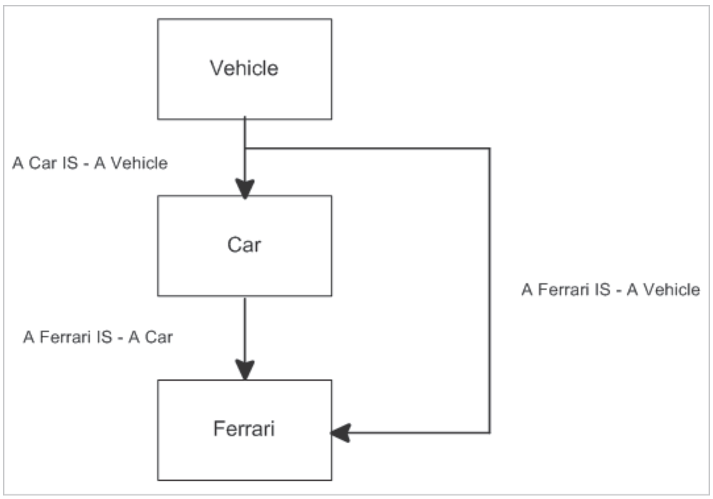


Figure 7.3: IS-A relation

The IS-A relationship can also be used in the case of interface implementations. An IS-A relationship in case of interfaces is expressed in Java with the keyword `implements`. A Java class can implement multiple interfaces. This is called multiple-interface-inheritance in Java. It is used to force a subclass to follow user-defined rules on overridden methods.

Knowledge Check 5

1. Given an interface **`Transaction`** containing the following methods:

```
void deposit(int amount); void withdraw(int amount);
```

Identify which of the following codes correctly implement the **`Transaction`** interface?

(A)
```
class BankAccount implements Transaction {
      void deposit(double amount) {
            // Code to deposit amount
      }
      void withdraw(double amount) {
            // Code to withdraw amount
      }
}
```

(B)
```
class BankAccount {
      void deposit(int amount) {
            // Code to deposit amount
      }
      void withdraw(int amount) {
            // Code to withdraw amount
      }
}
```

(C)
```
class BankAccount implements Transaction {
     public  void deposit(int amount) {
            // Code to deposit amount
      }
     public  void withdraw(int amount) {
            // Code to withdraw amount
      }
}
```

(D)
```
class BankAccount implements Transaction {
      void deposit(int amount) {
            // Code to deposit amount
      }
      void withdraw(float amount) {
            // Code to withdraw amount
      }
}
```

Module Summary

In this module, **Inheritance and Interfaces**, you learnt about:

➢ **Inheritance**

Through Inheritance the characteristics and behavior are transmitted from a parent entity to a child entity. The inheriting class is called the subclass and the inherited class is called the superclass. Every Java class has exactly one superclass which can be either a user-defined class or the `Object` class.

In Overriding, a subclass defines a new instance method having the same signature as the superclass method. The `super` keyword allows an overridden method of a superclass to be invoked from the subclass method. It can also be used to access instance variables of the superclass. Since, constructors are not inherited, `super()` is used to invoke the constructor of a superclass in a subclass constructor.

➢ **Overloading of Methods**

Method overloading is the ability of a class to define several methods with the same name. A method can be overloaded only by having either different number of parameters and same data types or same number of parameters and different data types. Similar to methods, constructors can be overloaded. The same rules that apply for method overloading also apply to constructors. The `this` keyword in Java is used within a constructor or instance method to refer to the current object in memory.

➢ **Using `abstract` Keyword**

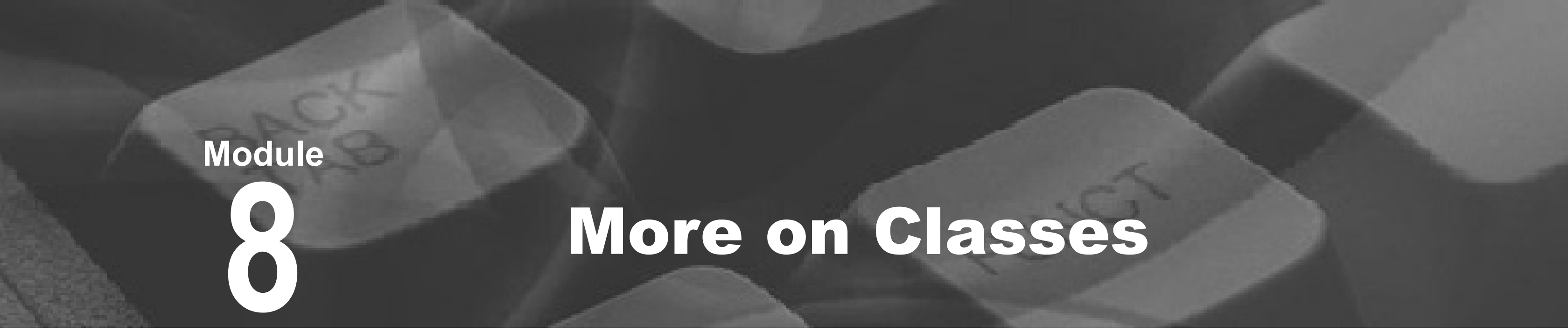
When a method has only declaration and no action statements in the body, it is called an `abstract` method. A Java class containing one or more `abstract` methods is known as an `abstract` class.

➢ **Using `final` Keyword**

A `final` variable is declared with the `final` keyword and assigned a value at the time of declaration. A compile time error is raised if a `final` variable is reassigned a value in a program after its initial declaration. To prevent a method from being overridden or hidden in a Java subclass, it is declared as `final`. A class that cannot be sub classed is called a `final` class in Java. The `final` keyword is applied to the class declaration to achieve this.

➢ **Interfaces**

An interface is defined as a reference type and is similar to a class except that it has only `final` variables and `abstract` method signatures. An interface cannot be instantiated. It can be implemented by other classes or extended by other interfaces.



Module 8 More on Classes

Module Overview

Welcome to the module, **More on Classes**. This module gives you a brief understanding of the different types of nested classes.

In this module, you will learn about:

- Scope of Variables
- Class Variables
- Nested Class

8.1 Scope of Variables

In this first lesson, **Scope of Variables**, you will learn to:

- Define scope of variables.
- Describe primitive variables.
- Describe reference variables.

8.1.1 Scope of Variables

There are two types of variables in Java. These are as follows:

- Primitive variable
- Reference variable

8.1.2 Primitive Variables

The first type is Primitive variable. A Primitive variable is used to store primitive data type values. Primitive variables can be of two types, depending on where they have been declared. They are declared as instance variables and local variables.

➢ **Instance Variables**

An instance variable is a variable that is declared inside a class, but outside any method. Instance variables are the fields of a class and are initialized only when the class is instantiated. The resulting object created has its own copy of each field of the class depending on the type of keyword used with the variable.

These variables are active until the class is active. In other words, these variables can be accessed, until a reference to the class or object exists.

➢ **Local Variables**

Variables declared inside a method are called local variables. These variables are created when the method is called. These variables are destroyed when the method is exited. Local variables can only be accessed inside the method. They cannot be accessed from any other method or anywhere else inside the class.

The following code snippet demonstrates the concept of primitive instance and local variables.

Code Snippet:

```
public class Vehicle {
    private int horsepower;    // Instance Variable
    public void getEngineType() {
        int numberOfCylinders;    // Local Variable
        horsepower = 1000;    // Can be accessed
    }
    public void getVehicleType() {
        // Cannot access this variable from this method.
        numberOfCylinders = 8;
    }
}
```

8.1.3 Reference Variables

Reference variables are used to store references to objects. They are declared to be of a certain type which can never be changed. However, they can be used to refer to any object of the declared type or a sub-type of the declared type. They can be declared as instance variables and local variables.

➢ **Instance Variables**

A reference variable when declared as an instance variable is accessible throughout the class. These variables are initialized by default, when the class is instantiated. The scope of such an instance variable is the class which declares it and the life of such a variable is until there is a reference to the class.

➢ **Local Variables**

Local variables cannot be marked `abstract` or `static`, but they can be marked `final`. Local variables do not get default values like instance variables. Hence, as a rule, they must be initialized before they can be used in the method. A local variable cannot be referenced from any code outside the method in which it is declared. It is possible to give the same name to a local variable as that of an instance variable. This technique is known as shadowing. This practice is generally discouraged, but if the application developer does want to give the same name to the local and instance variables for better readability of code then the use of the this keyword is recommended.

The following code snippet demonstrates the concept of reference instance and local variables.

Code Snippet:

```
public class PixelPoint {
    Pixel pix; // Instance Reference Variable

    public void showPixel() {
        Pixel newPixel; // Local Reference Variable
    }
    public void setPixel(Pixel pix) {
        // Initializing instance reference variable using this keyword.
        this.pix = pix;
     }
}
```

Knowledge Check 1

1. Which of the following variables can be accessed until a reference to the class or object exists?

(A)	Instance variables
(B)	Local variables
(C)	Static variables
(D)	Abstract variables
(E)	Final variables

2. Which of the following variables must be initialized before they can be used in the method?

(A)	Instance variables
(B)	Local variables
(C)	Static variables
(D)	Abstract variables
(E)	Final variables

8.2 Class Variables

In this second lesson, **Class Variables**, you will learn to:

- Describe class variables.
- Declare and access class variables.
- Class and instance variables.
- Describe `static` methods.
- Advantages and disadvantages of `static` methods.
- State the syntax of `static` initializers.

8.2.1 Class Variables

Class variables are declared using the `static` keyword. All instances of the class share the same value of the class variable. The value of a class variable can be accessed and modified by using class methods or instance methods. Once the value is modified, all instances of the class are updated to share the same value.

8.2.2 Declare Class Variables

Fields that have the `static` modifier in their declaration are called static fields or class variables. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

The following is the syntax to declare class variables.

Syntax:

```
<static> <data_type> <variable_name>;
```

where,

`<static>` implies that all instances of the class share the same static variable

`<data_type>` is the data type of the variable

`<variable_name>` is the name of the variable

The following code snippet shows the declaration of class variables.

Code Snippet:

```
class Book{
    static int price=0;
}
```

8.2.3 Access Class Variables

Class variables are referenced by the class name itself. This makes it clear that they are class variables.

The following is the syntax to access class variables.

Syntax:

```
<class_name>.<variable_name>
```

where,

`<class_name>` is the name of the class where the variable is declared

`<variable_name>` is the variable name

The following code snippet demonstrates how to declare and reference class variables.

Code Snippet:

```
class LocalDemo{
    static int price=1;
    public static int getCost() {
        return LocalDemo.price;
    }
}
```

8.2.4 Class and Instance Variables

Table 8.1 lists the differences between instance and class variables.

Instance variables	Class variables
They are initialized only when the class is instantiated.	They are static variables and are assigned a default value even before the class is instantiated.
They can be accessed only by using an object reference.	They can be accessed by using the object reference as well as the class name. Since, they are called class variables it is advisable not to use object references to access static or class variables.
Each new instance of the class has its own copy of the variable.	All instances of the class share the same copy of the static variable.

Table 8.1: Differentiation between Instance and Class variables

8.2.5 Static Methods

Static methods, also known as class methods, do not have reference to any instance variable in a class. These are used to access class variables and methods. However, static methods cannot access instance variables or methods directly. The keyword `this` cannot be used inside a static method.

The following is the syntax for declaring static methods.

Syntax:

```
<static><return_type><method_name>(<parameter_list>) {
    //body of method
}
```

where,

`<static>` is the keyword for declaring a static method

`<return_type>` is the data type of the value returned by the method

`<method_name>` is the name of the method

`<parameter_list>` is the list of parameters

Static methods can be accessed by using a class name or an instance name. However, accessing static methods using instance is not recommended, as it violates the principle that class methods should be accessible to only classes, not instances.

The following is the syntax to access static methods.

Syntax:

```
<class name>.<class method>
```

The following code demonstrates how to declare and access static methods.

Code Snippet:

```
class StatMethDemo {

    public static int sum() {
        int sum = 0;
        int pt = 5;
        for (int i=0; i<pt; i++) {
            sum += pt;
        }

        System.out.println("Total Sum is: " + sum);
        return sum;
    }
   public static void main(String args[]) {
       StatMethDemo.sum();
   }
}
```

Output:

```
Total Sum is: 25
```

8.2.6 Advantages and Disadvantages of Static Methods

The advantages of static methods are as follows:

1. Static methods can be invoked by using the class name directly.

2. Static methods can be used to implement behaviors that are not influenced by the state of any instances.

3. Static methods can be re-defined in instances.

4. Static methods can be accessed by using the `dot` operator with an instance of the class. This is however a syntax anomaly since the compiler actually replaces the instance reference with the class name before invoking the method.

The disadvantages of static methods are as follows:

1. A static method can be called by using an un-initialized object reference. The compiler only checks for the type of the object and whether the method being called is a static method. Hence, a static method can be accessed even by using an un-initialized object reference variable.

2. A static method cannot be overridden, although it can be hidden.

3. Non-static attributes of the class cannot be accessed from within a static method.

4. Other non-static methods of the class or instance cannot be accessed from within a static method.

5. Since static methods can be invoked without using an instance of the class containing it, the use of this keyword is prohibited.

8.2.7 Static Initializer

A static initializer is a block of code embraced in curly braces `{ }`. It is preceded by the keyword `static`. It initializes the `static` variables in a class.

The following is the syntax to declare a static initializers block.

Syntax:

```
static {
    // initialization code
}
```

The code snippet demonstrates the use of static initializer.

Code Snippet:

```
class StatInit{
    static int x = 6;
    static int y;
    // Static initializer
    static{
        for(int count = 0; count < 6; count++){
           y += x;
        }
    }

    public static void main(String args[]) {
        System.out.println("Value of x is: " + x);
        System.out.println("Value of y after 6 counts is: " + y);
    }
}
```

Output:

```
Value of x is: 6
Value of y after 6 counts is: 36
```

Note: There can be more than one static initialization block in a class. These can be placed anywhere in the class. Java compiler does not allow forward referencing of class variables in static initialization block. In other words, a static initialization block can reference only those class variables that have been declared before it.

Knowledge Check 2

1. Which of these statements about class variables are false?

(A)	Static initialization block is preceded by the keyword `static`.
(B)	The `static` keyword in a class variable implies that instances of the class cannot share the same static variable.
(C)	Static method can access only class variables and methods directly.
(D)	Static initialization block can reference only those class variables that have been declared before it.
(E)	Static initializer can be placed only in the main class.

8.3 Nested Class

In this last lesson, **Nested Class**, you will learn to:

- Describe Nested Class.
- Explain Member Class.
- Explain Local Class.
- Explain Anonymous Class.

8.3.1 Nested Class

A nested class is a class defined within another class. It can have access to members of the outer or enclosing class, even if the members are declared `private`.

Nested classes can be used for the following reasons:

- **Allows logical grouping of classes**

  If a class works as a helper class to another class, then it is logical to embed the second class within the first and keep them together.

- **Increases encapsulation**

  If class **`A`** needs access to private members of class **`B`**, then class **`A`** can be declared as nested class of class **`B`**. As a result, class **`A`** can access all the `private` members of Class **`B`** and at the same time class **`A`** will be hidden from the outside world.

- **More maintainable code**

  Nesting small classes within top-level classes places the code closer to where it is used. This makes it easier to maintain the code.

8.3.2 Advantages and Types

Nested classes have several advantages. Some of them are as follows:

- Readable
- Maintainable

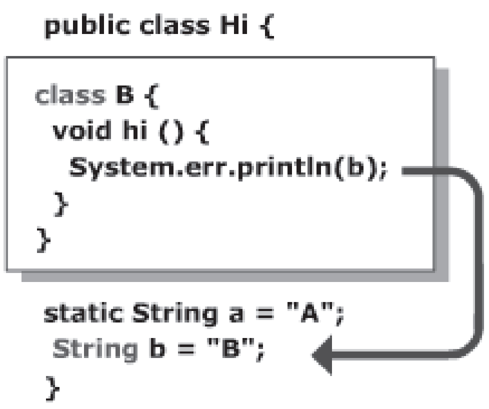
The different types of nested classes are as follows:

- Member classes or non-static nested classes
- Local classes
- Anonymous classes
- Static Nested classes

8.3.3 Member Classes

A member class is a non-static inner class and is declared as a member of an outer or an enclosing class. It cannot have static modifiers because it is associated with instances. It can access all fields and methods of the outer class, but the reverse is not true. An outer class cannot access a member of an inner class, even if it is declared as `public` because members of an inner class are declared within the scope of inner class. A member class can be declared as `public`, `protected`, `private`, `abstract`, `final`, or `static`.

Figure 8.1 shows how to declare member classes within an outer class.



```
public class Hi {
 class B {
  void hi () {
    System.err.println(b);
  }
 }
 static String a = "A";
  String b = "B";
 }
```

Figure 8.1: Member Classes

The following is the syntax to access a member class.

Syntax:

```
…
<OuterClass>.<InnerClass>
…
```

The following code demonstrates how to declare and access a member class.

Code Snippet:

```
class Member { // Top-level class
String x ;

//Declaring Inner class instance as part of member variable of
//outer class
Inner instanceInner;
   public static void main(String args[]) {
            // Declaring instance of Inner class
            Member.Inner theInner = new Member().new Inner();
            theInner.disp();

   }
public Member(){
      x = "Welcome to MemberClass Demo";
    // Instance creation of Inner class
    instanceInner = new Inner();
}

void displayInner(){
   // Not allowed! Outclass members cannot access inner class //members
    System.out.println(y);
  }
   class Inner {
      // local instance variable of Inner class
      int y=5;
       void disp() {
           System.out.println(x);
       }
    }
 }
```

Note: You can declare an instance variable of an inner class in the outer class as its member variable. The instance variable can be initialized in the constructor of the outer class in the same way as its other members.

8.3.4 Local Classes

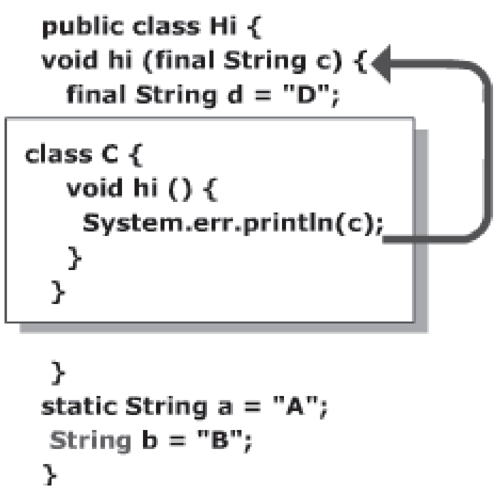
A local class is declared within a method, constructor or an initializer. In other words, a local class is declared within a block of code and is visible only within that particular block. It cannot have a static modifier. It has the ability to refer to local variables in the scope that defines them. Modifiers, such as `public`, `protected`, `private`, or `static` cannot be used in local classes.

Local classes have the following features:

➢ Local classes are associated with an instance of containing class, and can access any members, including private members, of the containing class.

➢ Local classes can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition, provided that these are declared as `final`.

Figure 8.2 shows how to declare local classes within a method.



```
public class Hi {
void hi (final String c) {
  final String d = "D";
class C {
    void hi () {
      System.err.println(c);
    }
  }

  }
static String a = "A";
 String b = "B";
}
```

Figure 8.2: Local Classes

The following code demonstrates the use of a local class.

Code Snippet:

```
package pack;

public class Outer {

    /** Creates a new instance of Outer */
    public Outer() {
    }
    // accessible from local class
    private int i = 100;


    public static void main(String args[]) {
        Outer objOuter = new Outer();
        objOuter.innmethod();
        System.out.println(objOuter.innmethod());
    }
```

```
         int innmethod () {
           // Not accessible from local class
           int j=5;
          // Accessible from local class
          final int k=10;
          class InnClass {  // local class
              void disp() {
               // final variable k is accessible
               System.out.println("k="+k);
              }
        }
      return i;
    }
}
```

Output:

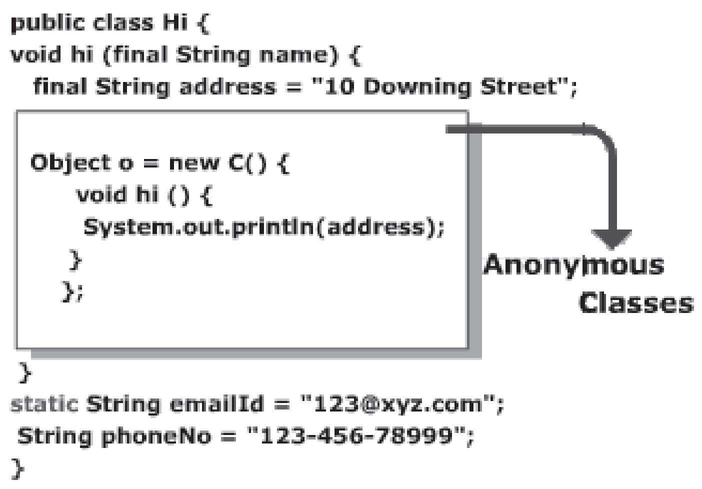
```
100
```

8.3.5 Anonymous Classes

An anonymous class does not have a name. It is a type of local class. It also does not allow the use of `extends`, `implements` clauses, and access modifiers, such as `public`, `private`, `protected`, and `static`. Since, it does not have a name, it cannot define a constructor. If constructor declaration is needed, then a local class can be used. An anonymous class cannot define any static fields, methods, or classes.

Anonymous interfaces are not possible to implement because an interface cannot be implemented without a name.

Figure 8.3 shows how to declare anonymous classes.



```
public class Hi {
void hi (final String name) {
  final String address = "10 Downing Street";

 Object o = new C() {
     void hi () {
      System.out.println(address);
    }
   };

}
static String emailId = "123@xyz.com";
 String phoneNo = "123-456-78999";
}
```

Figure 8.3: Anonymous Classes

The following code demonstrates the use of anonymous class.

Code Snippet:

```
class Book {
    public void disp() {
    }

    public static void main(String args[]) {
        Title objTitle = new Title();
        objTitle.objBook.disp();
    }
}

class Title {
    // Anonymous class definition
    Book objBook = new Book() {
        public void disp() {
            System.out.println(“Basics of Java Programming”);
        }

    }; // ; indicates end of statement
}
```

Output:

```
Basics of Java Programming
```

8.3.6 Static Nested Class

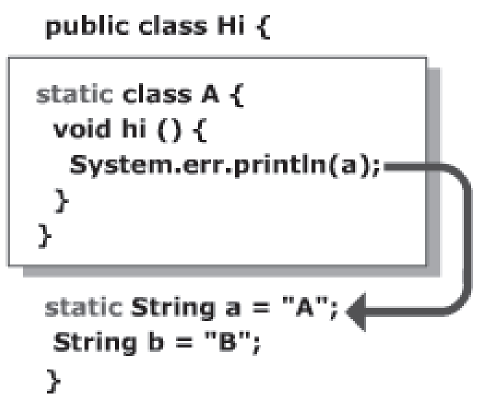
A static nested class is associated with its outer class. It cannot refer directly to instance variables or methods defined in its enclosing class, but it can access class methods or variables directly. An object needs to be instantiated to access the instance variables and methods of the enclosing class.

A static nested class can be accessed using the enclosing class name.

```
EnclosedClass.StaticNestedClass
```

Static nested classes can have `public`, `private`, `protected`, `package`, `final`, and `abstract` access specifiers. `Private` and `protected` static nested classes are used to implement the internal features of a class or a class hierarchy. `public` static nested classes are often used to group classes together or give access to `private` static variables and methods to a group of classes.

Figure 8.4 shows the declaration of static nested classes.



```
public class Hi {
 static class A {
  void hi () {
   System.err.println(a);
  }
 }
 static String a = "A";
 String b = "B";
}
```

Figure 8.4: Static Nested Classes

The following code demonstrates the use of static nested class.

Code Snippet:

```
package pack;
class Stat {
// accessible to static nested class
private static final String name = "John";
private static String company = "Dunn";
// not accessible to static nested class
private int age = 50;

    public static class Out {
         void welcome() {
           System.out.println("Welcome to Aptech Limited !!");
           System.out.println(name+company);
         }
    }
}
public class StaticNestedDemo {

     static class StatInn {void desc() {
        System.out.println("Aptech Limited is a Global Learning Solutions
organization");}
    }
    public static void main(String args[]) {
        // Constructing instance of static nested class
        Stat.Out n = new Stat.Out();
        n.welcome();
        // Declaring constructor of public static nested class directly
        StatInn objStat = new StatInn();

        objStat.desc();
    }
}
```

Output:

```
Welcome to Aptech Limited !!
JohnDunn
Aptech Limited is a Global Learning Solutions organization
```

Knowledge Check 3

1. Can you match the different types of terms against their corresponding description?

Description		Term	
(A)	Cannot refer directly to instance variables or methods defined in its enclosing class	(1)	Member class
(B)	Can be declared as `public`, `protected`, `private`, `abstract`, `final`, or `static`	(2)	Local class
(C)	Is declared inside a method, constructor, or an initializer	(3)	Static nested class
(D)	Cannot define a constructor	(4)	Nested class
(E)	Is defined within another class	(5)	Anonymous class

2. Which of the statements about classes are true and which statements are false?

(A)	A nested class can access private members of outer class.
(B)	An anonymous class does not allow the use of extends keyword.
(C)	A static nested class cannot access class methods or variables directly.
(D)	Public static nested classes are used to group classes.
(E)	Modifiers can be used in local classes.

Module Summary

In this module, **More on Classes**, you learnt about:

➢ **Scope of Variables**

There are two types of variables in Java. The first type is Primitive variable which is used to store primitive data type values. The second type is Reference variable which is used to store the reference to an object. Both of them can be declared as instance variables and local variables.

➢ **Class Variables**

Class Variable is allocated memory initially, when the class is loaded. Only one variable is used for all the objects of the class. Static methods use class variables, not any instance variables of the class.

➢ **Nested Class**

A nested class is defined within another class. A nested class is also known as inner class. Nested classes can be static or non-static. Non-static nested classes are also called member classes.

A nested class is defined within another class. The different types of nested classes are Member classes or non-static nested classes, local classes, anonymous classes, and static nested classes.

Module 9

Exceptions

Module Overview

Welcome to the module, **Exceptions**. This module focuses on defining, creating, and handling of exceptions in Java programs. This module aims for clear understanding of the different kinds of exceptions and exception handling mechanisms in Java. Further, it discusses the flow of programs without using exceptions, and then using exceptions, and also the use of assertions in programs.

In this module, you will learn about:

- Introduction to Exceptions
- Exception handling in Java
- User defined exceptions
- Assertions

9.1 Introduction to Exceptions

In this first lesson, **Introduction to Exceptions**, you will learn to:

- Explain the concept of Exceptions.
- Identify the different types of Exceptions.

9.1.1 Causes for Exceptions

An exception is an abnormal condition that arises out of an extraordinary situation disrupting the flow of program's instructions. Exceptions report error conditions in a program.

In programs, exceptions can occur due to any of the following reasons:

- **Programming errors**

  These exceptions arise due to errors present in APIs, such as `NullPointerException`. The program using these APIs cannot do anything about these errors.

- **Client code errors**

  The client code attempts operations, such as reading content from a file without opening it. This will raise an exception. The exception will provide information about the cause of the error and is handled by the client code.

- **Errors beyond the control of a program**

  There are certain exceptions that are not expected to be caught by the client code, such as memory error or network connection failure error. These are errors which have been raised by the run-time environment.

9.1.2 Classification of Exceptions

In Java there are two types of exceptions and they are as follows:

- **Checked Exceptions**

  Checked exceptions are generated in situations that occur during the normal execution of a program. Some common examples of checked exceptions are: requesting for missing files, invalid user input, and network failures. These exceptions should be handled to avoid compile-time errors. If an exception occurs during the execution of a method, the method can handle the exception itself or throw the exception back to the calling method to denote that a problem has occurred. The calling method can again handle the exception itself or throw it to its calling method. This process can continue until the exception reaches the top of a thread and the thread is terminated. This is known as the **Call-Stack** mechanism. The main advantage of this practice is that a developer has the flexibility of putting the error handling code wherever he chooses.

  These types of exceptions are derived from the `Exception` class. The program either handles the checked exception or moves the exception further up the stack. It is checked during compilation.

- **Unchecked Exceptions**

  Unchecked exceptions are generated in situations that are considered non-recoverable for a program. Common examples of these situations are attempting to access an element beyond the maximum length of an array. An application is not required to check for these kinds of exceptions. Runtime exceptions are also examples of unchecked exceptions. They generally arise from logical bugs. Unchecked exceptions that arise on account of environmental problems or errors are impossible to recover from and are known as Errors. Exhausting a program's allocated memory is a common example of an error.

  These exceptions are derived from the `RuntimeException` class, which in turn is derived from the `Exception` class. The program need not handle unchecked exception. It is generated during the execution of the program.

Figure 9.1 shows the exception hierarchy in Java.

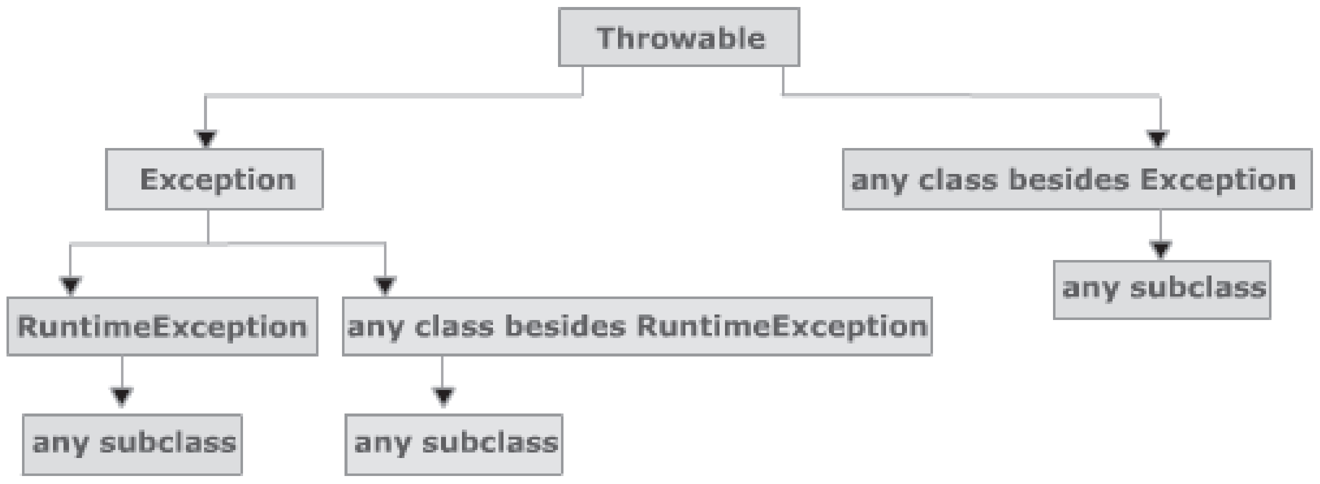


Figure 9.1: Classification of Exceptions

When an error occurs in a method during runtime, an object is created containing information about the error, its type, and the state of the program. This object is referred to as exception object. Exception objects are instances of classes that are derived from the base class `Throwable`.

9.1.3 Types of Checked Exceptions

All `Checked` exceptions are derived from the `Exception` class.

Table 9.1 lists some of the common `Checked` exceptions.

Exception	Description
`InstantiationException`	This exception occurs if an attempt is made to create an instance of the `abstract` class.
`InterruptedException`	This exception occurs if a thread is interrupted.
`NoSuchMethodException`	This exception occurs if the Java Virtual Machine is unable to resolve which method is to be called.
`RuntimeException`	This exception may be thrown during normal operation of Java Virtual Machine if some erroneous condition occurs.

Table 9.1: Common Checked Exceptions

9.1.4 Types of Unchecked Exceptions

All Unchecked exceptions are directly or indirectly derived from the RuntimeException class.

Table 9.2 lists some of the common Unchecked exceptions.

Exception	Description
ArithmeticException	Derived from RuntimeException and indicates an Arithmetic error condition.
ArrayIndexOutOfBoundsException	Derived from IndexOutOfBoundsException class and is generated when an array index is less than zero or greater than the actual size of the array.
IllegalArgumentException	Derived from RuntimeException. Method receives an illegal argument.
NegativeArraySizeException	Derived from RuntimeException. Array size is less than zero.
NullPointerException	Derived from RuntimeException. Attempt to access a null object member.
NumberFormatException	Derived from IllegalArgumentException. Unable to convert the string to a number.
StringIndexOutOfBoundsException	Derived from IndexOutOfBoundsException. Index is negative or greater than the size of the string.

Table 9.2: Common Unchecked Exceptions

Knowledge Check 1

1. Which of these statements about exception are true and which statements are false?

(A)	OutOfMemory error can be handled in a program.
(B)	Exceptions in Java are objects of Exception class.
(C)	Checked exceptions can be handled in a program.
(D)	All unchecked exceptions are directly derived from RuntimeException.
(E)	InterruptedException occurs if the Java Virtual Machine is unable to resolve the invoker method.

2. Can you match the Exceptions on the right with their Descriptions on the left?

	Descriptions		Exceptions
(A)	This exception is derived from `IllegalArgumentException` and occurs when an attempt is made to convert a string to a number.	(1)	NullPointerException
(B)	This exception occurs if an attempt is made to create an instance of an interface or an abstract class.	(2)	NumberFormatException
(C)	This exception is derived from `RuntimeException` class and is thrown when an attempt is made to access a null object member.	(3)	ArrayIndexOutofBoundsException
(D)	This `unchecked` exception is thrown when an array size is specified as less than zero.	(4)	InstantiationException
(E)	This exception is generated when an array index is less than zero or greater than the actual size of the array.	(5)	NegativeArraySizeException

9.2 Exception Handling in Java

In this second lesson, **Exception Handling in Java**, you will learn to:

- State the use of `try-catch` block.
- Describe the use of `finally` block.
- State the flow of execution in an exception handling block.
- Describe the use of `throw` and `throws` keyword.
- Describe the use of multiple `catch` blocks.

9.2.1 Using the 'try-catch' Block

Java supports exception handling mechanism. The code that raises exceptions can be enclosed in a `try` block enabling the adjoining `catch` block to handle the raised exception. In addition, Java provides methods for reporting the cause of exception. Handling of exception involves logging the information on the exception and deciding whether to abort or continue execution of the statements succeeding the `try-catch` block.

The type of argument that the `catch` block will handle is determined by its argument. The type represents an `Exception` class that must be derived from the `Throwable` class. The code in the `catch` block will be executed only when exception of that type is invoked, otherwise the control passes to the statement following the `try` block.

Figure 9.2 shows the exception handling mechanism using `try-catch` block.

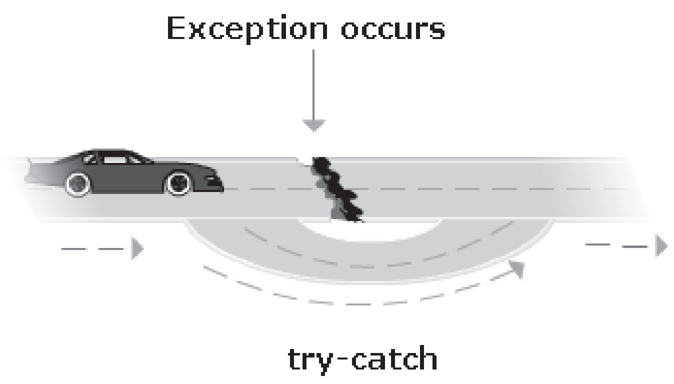


Figure 9.2: Using 'try-catch'

The following is the syntax to declare a `try-catch` block.

Syntax:

```
try {
  statement1;
  statement2;
} catch (ExceptionType objectName1) {
  }
```

The following code demonstrates the use of `try-catch` block.

Code Snippet:

```
// Exception handling mechanism
try{
     // The following statement within try block will raise divide-by-zero,
     // a type of ArithmeticException exception
    System.out.println(1/0);
}
```

```
// Exception handler method; ArithmeticException is exception type
catch (ArithmeticException e){
     System.out.println("This operation cannot be carried out");
     // method printStackTrace() prints detailed information about the
     //exception
     e.printStackTrace();
)
```

9.2.2 Using the 'finally' Block

In case of an unexpected exception, the program exits without executing the remaining part of the code, giving rise to undesirable situations. For example, during file read operation if the file read operation fails due to unexpected exception, the file remains open. There is a chance of the file getting corrupted. Thus, the `finally` block is used along with the `try-catch` block. Using the `catch` block with `try-finally` is optional. The `finally` block will always be executed, even if an unexpected exception occurs. The cleanup code is placed inside a `finally` block, such as closing the files in file read operation.

Figure 9.3 shows the exception handling mechanism using `try-catch` and `finally` block.

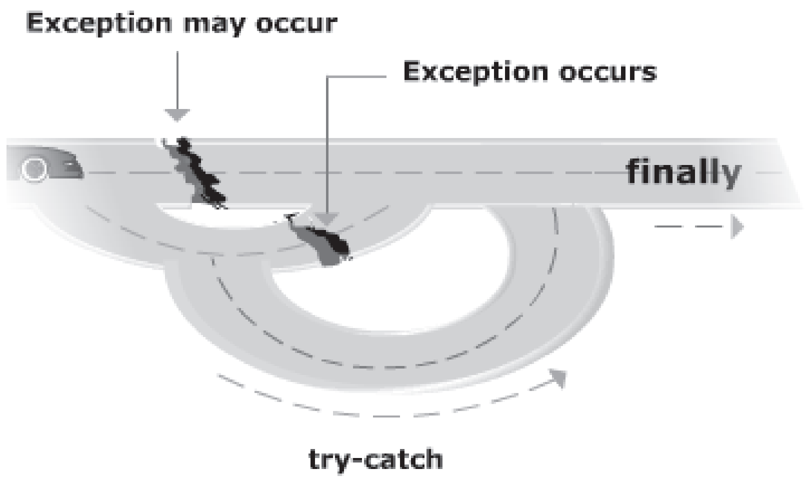


Figure 9.3: Using 'finally'

The following is the syntax to declare the `finally` block.

Syntax:

```
try{
     // May raise expected/unexpected exception(s)
     Statement1;
Statement2;
}
// catch block optional
  catch(ExceptionB eb){
     // Handler Code
     Statement1;
}
  finally{
     //clean up code
     Statement1;
}
```

The following code shows the use of `finally` block.

Code Snippet:

```
finally {
    // file object out
    if (out != null) {
        System.out.println("Closing file");
        out.close();
    } else {
        System.out.println("file not open");
    }
}
```

9.2.3 Execution Flow in case of Exception

Consider the following code snippet without a `try-catch` block:

Code Snippet:

```
int  i=10,j=0; //1
System.out.println("i/j="+ i/j); //2
int k=i+10; //3
```

Line number 3 is never executed as the division of the integer by 0 in line number 2 will result in throwing of an exception by the runtime system. Consequently, an `Exception` object will be created. However, there is no exception handling mechanism in the code. Therefore, the exception will be handled by the default handler of JVM. The default handler will print a message providing the exception object name and its stack trace from the point where the exception occurred. Finally, it terminates the program.

Note:

Stack trace:

The stack trace shows the sequence of method invocations that led up to the exception.

Exception object:

A Java exception is an object that describes an abnormal condition occurring in a part of code. The description contains information on error type and the state of the program. When an exception occurs in a method, an object representing that exception is created by the method and handed over to the runtime system.

Throw exception:

Creating an exception object and handing it to the runtime system is called throwing an exception.

9.2.4 Execution Flow with 'try-catch' Blocks

Java provides `try-catch` block to handle an exception.

The following code discusses earlier lines of code and encloses it in `try-catch` block.

Code Snippet:

```
int  i=10,j=0;
try{
  System.out.println("i/j="+ i/j);
  }
  catch(ArithmeticException e){
    System.out.println("Exception Caught"+ e.getMessage());
  }
int k=i+10;
```

In the example, the `catch` block with argument `ArithmeticException` handles the `Exception` object, the execution control flows to the `catch` block from the `try` block, and a message is printed with information on the exception object. Finally, it flows to the last line.

Note:

Call stack:

When a method throws an exception, the runtime system tries to find an appropriate method to handle it starting from the method that caused the exception. In this process, the system invokes a set of methods. The list of invoked methods that is called by the runtime system to reach to the method where the exception occurred is known as the `call stack`.

Exception handler:

The runtime system searches the `call stack` for a method that contains a piece of code that is able to handle the exception. This piece of code is called an `exception handler`.

Catching an exception:

When an appropriate exception handler matching an exception object is found, then it is known as `catching an exception`.

9.2.5 Execution Flow of Exception

If the last line needs to be always executed even if unexpected exception occurs, then it has to be placed inside a finally block along with `try-catch` block.

The following code demonstrates the `finally` block to place code that needs to be always executed.

Code Snippet:

```
int  i=10,j=0;
try{
    System.out.println("i/j="+ i/j);
  }
  catch(ArithmeticException e){
    System.out.println("Exception Caught"+ e.getMessage());
  }
  finally {
    int k=i+10;
  }
```

In the example, an `Exception` object is thrown in the `try` block that is handled in the `catch` block, because the exception object matches its argument, `ArithmeticException`.

Next, the control passes to the finally block and the statement in it is executed, and lastly the control passes to the statement following the finally block.

9.2.6 Using 'throw' and 'throws' Keywords

The throws keyword indicates that a method might throw the declared exception during its execution. The throw keyword is used to manually throw an exception after performing some validation in a program. After the throw keyword is encountered, the flow of execution is altered and the subsequent statements are not executed. The exception generated by the throw statement is then propagated to the previous calling method on the call stack.

The throw statement requires a single argument: a Throwable object that is an instance of the Throwable class.

Sometimes, a method can throw more than one exception. A comma-separated list of all exceptions thrown by a method is given with the method declaration. The throws keyword is used by the method to raise any checked or unchecked exception that it does not handle and enables the caller of the method to guard themselves against exception.

More than one exceptions can be listed with the throws clause and are separated by a comma. Except for Error or RuntimeException and their subclasses, the throws clause is necessary for all exceptions.

Figure 9.4 shows the throw and throws.

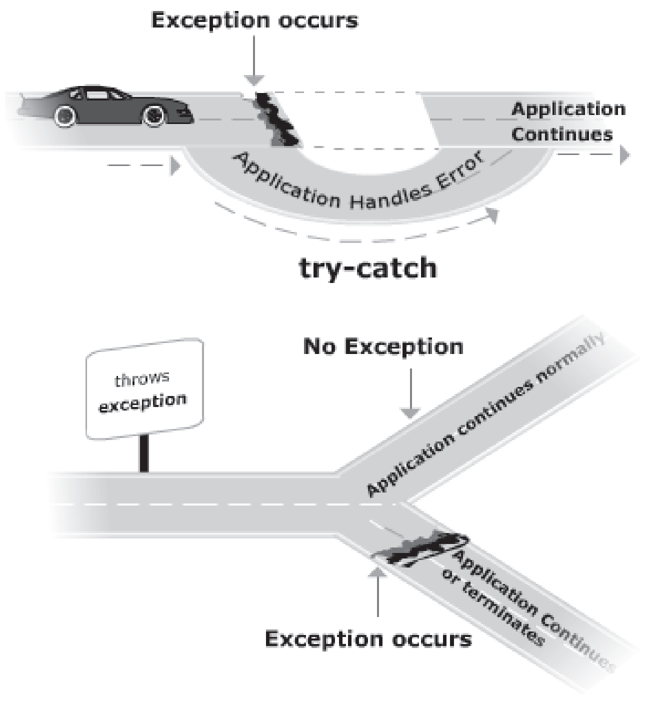

Exception occurs
Application Continues
Application Handles Error
try-catch
throws exception
No Exception
Application continues normally
Application Continues or terminates
Exception occurs

Figure 9.4: Using 'throw' and 'throws'

The following is the syntax of `throw` keyword to manually throw an exception.

Syntax:

```
throw throwableObject;
```

The following code demonstrates the use of `throw` keyword.

Code Snippet:

```
public class CalculatePrice {
    float price;
    void setPrice(float itemPrice) throws ArithmeticException {
         if (itemPrice<0.0) throw new ArithmeticException();
         price=itemPrice;
    }
}
```

If a negative value has been entered, the method **`setPrice`** throws an exception.

Output:

```
java.lang.ArithmeticException
  at CalculatePrice.setPrice(CalculatePrice.java:33)
   CalculatePrice.main(CalculatePrice.java:41)
```

9.2.7 Using Multiple 'catch' Blocks

A `try` block can have multiple catch blocks. This is required when the code in the `try` block can raise multiple exceptions. In such cases, multiple `catch` blocks containing different exception types are provided. When an exception is raised, each `catch` block is inspected one at a time until a matching block is found. Once a match is found, the remaining `catch` blocks are ignored, and execution proceeds after the last `catch` block.

When multiple `catch` blocks are used, `catch` blocks with exception subclasses must be placed before the superclasses, otherwise, the superclass exception will `catch` exceptions of the same class and its subclasses. Consequently, `catch` blocks with exception subclasses will never be reached.

The following is the syntax to declare multiple `catch` blocks.

Syntax:

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

The following code demonstrates the use of multiple `catch` blocks.

Code Snippet:

```
try {
// potential exception generator statements
}

// First catch block
catch (FileNotFoundException e) {
    System.out.println("Caught FileNotFoundException: " + e.getMessage());
}


// Second catch block
catch (IOException e) {
    System.out.println("Caught IOException: " + e.getMessage());
}
catch (Exception e) {
    System.out.println("Caught Exception: " + e.getMessage());
}
```

Note: As a good practice, always place `catch` blocks with exception subclasses first and the base class of all exceptions, Exception at the end. Otherwise, `catch` blocks with subclass exceptions will be unreachable and, during compile time, display unreachable code error.

Knowledge Check 2

1. You want the program to print the following:

```
Message 2Message 3
```

Can you identify the correct code?

(A)

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

  try {
    StringBuilder[] s = new StringBuilder[10];
    s[0].reverse();
  }
  catch (ArrayIndexOutOfBoundsException err) {
  System.out.print("Message 1");
  }
  catch (Exception err) {
    System.out.print("Message 2");
  }

  System.out.print("Message 3");
}
```

(B)

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

  try {
    StringBuilder[] s = new StringBuilder[10];
    s[0].reverse();
  }
    System.out.print("Message 1");
  catch (ArrayIndexOutOfBoundsException err) {
  }
  catch (Exception err) {
    System.out.print("Message 2");
  }

  System.out.print("Message 3");
  }
```

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

	  try {
	    StringBuilder[] s = new StringBuilder[10];
	    s[0].reverse();
	  }
	  catch (ArrayIndexOutOfBoundsException err) {
	    System.out.print("Message 1");
	  }
	  System.out.print("Message 2");
	  catch (Exception err) {
	      }
	  System.out.print("Message 3");
	  }
	```

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

	  try {
	    StringBuilder[] s = new StringBuilder[10];
	    s[0].reverse();
	  }
	  catch (ArrayIndexOutOfBoundsException err) {
	    System.out.print("Message 1");
	  }
	  catch (Exception err) {
	  System.out.print("Message 2");
	  System.out.print("Message 3");
	      }
	  }
	```

2. Can you arrange the steps in the right sequence for the program to compile?

(A)	`try{`
	`    System.out.println("i/j="+ i/j);   }`

(B)	`catch(Exception e){`
	`    e.printStackTrace();   }`

(C)	`finally {`
	`    int k=i+10;  System.out.println("k="+k); } }`

(D)	`public static void main(String args[]) {`
	`  int  i=10,j=1;`

(E)	`catch(ArithmeticException ex){`
	`    ex.printStackTrace();   }`

3. Which of these statements about the flow of execution, throw and throws and multiple catch blocks are true and which statements are false?

(A)	The throw and throws keyword must be used by a method that may throw unchecked exceptions.
(B)	In multiple catch statements, exception subclasses are placed before any of their superclasses.
(C)	A statement having a throw keyword need not be encompassed in a try-catch block.
(D)	Multiple exceptions can be thrown using a single throw statement.
(E)	Multiple exceptions can be claimed using the throws keyword.

9.3 User-defined Exceptions

In this third lesson, **User-defined Exceptions**, you will learn to:

- Explain user-defined exceptions.
- Explain implementation of an user-defined exceptions.
- Explain handling and throwing of user-defined exceptions.
- Exception chaining.

9.3.1 User-defined Exceptions

User-defined exceptions are custom exceptions. These exceptions are created when predefined exceptions are not sufficient to handle situations specific to an application. User-defined exception classes are subclassed from the base class `Exception`. These exceptions are handled and thrown in the same manner as predefined exceptions.

9.3.2 Creating User-defined Exceptions

The `Exception` class is the base class of all user-defined exceptions. User-defined exceptions provide business solutions that can be used in the programs. The `Throwable` class is the super class of all exceptions and errors in the Java language.

All the methods of the `Throwable` class are inherited by the `Exception` class, since `Exception` is a subclass of `Throwable` class. Hence, user-defined exception classes can use all the methods of the `Throwable` class.

The following is the syntax to create user-defined exception class.

Syntax:

```
// any user defined exception has to extend base class
class MyException extends Exception{
// exception-handling code
}
```

The following code demonstrates the declaration of user-defined exception class.

Code Snippet:

```
public class InvalidPriceException extends Exception{
    // class variables
    private String exceptionMesg;
    //Default Constructor
    public InvalidPriceException(){
        exceptionMesg="";
    }
    // Constructor with one arguement
    public InvalidPriceException(String str){
        exceptionMesg=str;
    }

    // overrides base class method


     public String getMessage()
     {
     return exceptionMesg;
     }
}
```

9.3.3 Throwing User-defined Exceptions

It is important for a developer to know when to make use of exceptions. There may be a tendency to excessively use the Exception API, just because it is convenient.

It is important to note that every time an exception is thrown the execution of the program is hampered.

The only guideline to follow for deciding when to throw an exception is that if a method encounters an abnormal condition that it can't handle, it should throw an exception. This raises another question about what can be classified as an 'abnormal condition'.

An abnormal condition, therefore, would be any condition that wouldn't reasonably be expected as part of the 'normal functioning' of a method. Exceptions can be used to test whether the parameters to a public method or public constructor are legal.

The following is the syntax to throw user defined exceptions.

Syntax:

```
throw new ExceptionType;
```

where,

`ExceptionType` is an instance of a user-defined exception class subclassed from `Throwable` class.

The following code demonstrates the throwing of a user-defined exception.

Code Snippet:

```
void setPrice(float itemPrice) throws InvalidPriceException{
     if (itemPrice <0) throw new InvalidPriceException();
     price=itemPrice;
     }
```

If negative price has been entered, the method `setPrice` throws an exception.

Output:

```
InvalidPriceException: Invalid Price
  at CalculatePrice.setPrice(CalculatePrice.java:33)
   CalculatePrice.main(CalculatePrice.java:41)
```

Note: The client code that will invoke the method will handle the exception using `try-catch` block or rethrow it in case it does not want to handle it.

9.3.4 Exception Chaining

Exception chaining, or exception wrapping, is an object-oriented programming technique of handling exceptions by re-throwing a caught exception after wrapping it inside a new exception. The original exception is saved as a property (such as cause) of the new exception. The idea is that a method should throw exceptions defined at the same abstraction level as the method itself, but without discarding information from the lower levels.

The following code shows the common method to catch one exception and throw another in the Java code.

Code Snippet:

```
try {
    ...
} catch(YourException e) {
    throw new MyException();
}
```

If the information from the original exception is lost, debugging will become impossible. Hence, while wrapping exceptions, an accessor is generally provided to extract the contained exception. This allows developers to construct chains of exceptions consisting of wrapped exceptions.

The advantages of using the exception chaining facility are as follows:

➢ The fact that one exception caused another can be recorded, regardless of what the exceptions are.

➢ Since a common API is used to record the fact that one exception caused another, it encourages programmers to keep a record of the exception chain.

➢ The record that a particular exception caused another exception can be stored and referred to at a later time.

To keep a record of the exception chain, two methods of the `Throwable` class are used. The `getCause()` and `initCause(Throwable)`, and the two constructors, `Throwable(Throwable)` and `Throwable(String, Throwable)` are used to record the chain of exceptions. Other ‘general purpose’ exception classes (like `Exception`, `RunTimeException` and `Error`) have been similarly outfitted with `(Throwable)` and `(String, Throwable)` constructors. However, even exceptions without such constructors can be used as ‘wrapped exceptions’ by using the `initCause()` method.

The implementation of `Throwable.printStackTrace` has been modified to display back traces for the entire causal chain of exceptions. New method `getStackTrace()` provides programmatic access to the stack trace information provided by `printStackTrace`.

Knowledge Check 3

1. Which of these statements about user-defined exceptions are true and which statements are false?

(A)	A user defined exception must inherit from `RuntimeException` class.
(B)	A user defined exception must inherit from `Exception` class.
(C)	A user defined exception cannot be handled using `try-catch` block.
(D)	The keyword `throw` is used to throw user defined exceptions.
(E)	The keyword `throws` cannot to be used by methods that want to throw user defined exceptions.

2. Which one of the following declaration and invocation of user-defined exception is correct?

(A)	`public class MyException extends Exception {`
	`     public  MyException( String errorMessage )  {`
	`          super( errorMessage );`
	`     }`
	`}`
	`throw MyException();`

(B)	`public class MyException implements Exception  {`
	`     public  MyException()  {`
	`          super();`
	`     }`
	`}`
	
	`throw MyException;`

(C)	`public class MyException extends Error {`
	`     public MyException()  {`
	`          super();`
	`     }`
	`     public MyException( String errorMessage )  {`
	`          super( errorMessage );`
	`     }`
	`}`
	
	`throw new MyException;`

(D)
```
public class  MyException  extends  Exception  {
      public  MyException()  {
           super();
      }
      public  MyException( String  errorMessage )  {
           super(  errorMessage  );
      }
}
throw new MyException();
```

9.4 Assertions

In this last lesson, **Assertions**, you will learn to:

- State the use of `assert` statement.
- Explain implementation of assertion in code.
- Explain the use of internal invariants.
- State the use of control-flow invariants.
- Explain PreCondition, PostCondition, and Class Invariants.

9.4.1 Using the 'assert' Statement

An assertion allows testing the correctness of any assumptions that have been made about the program. Assertion is achieved using the `assert` statement in Java.

The `assert` statement is used with a boolean expression that is checked during runtime. The expression is believed to be true, but it generates an `AssertError`, if it is false.

There are two forms of the assert statement that are as follows:

- `assert booleanExpression;`
- `assert booleanExpression : expression;`

The first form, `assert booleanExpression;` evaluates the boolean expression and returns a `true` or `false` result.

Figure 9.5 shows the first form of declaring assertions.

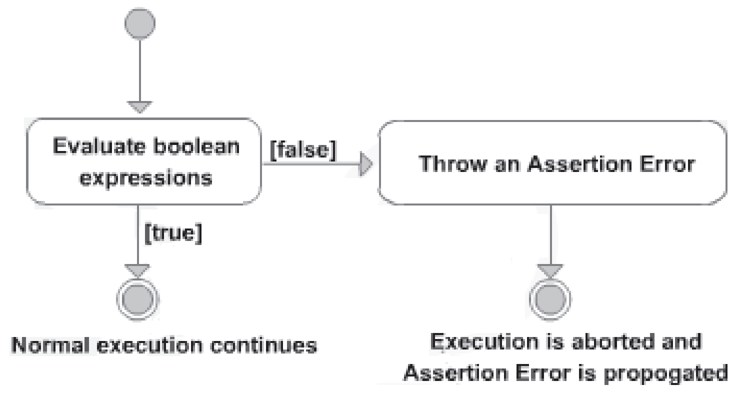

Evaluate boolean expressions
[false]
Throw an Assertion Error
[true]
Normal execution continues
Execution is aborted and Assertion Error is propogated

Figure 9.5: Using 'assert'

The following is the syntax for providing the first form of assertion.

Syntax:

```
assert expression1;
```

where, expression1 is a boolean expression that is evaluated. If the expression evaluates to a false value, then an `AssertionError` exception is thrown with no detailed message.

The following code demonstrates the first form of assertion with a boolean expression.

Code Snippet:

```
public class CalculateInterest{
public double calculateInterest(double principalAmount,double
numberOfYears,double rateOfInterest){
//First form of assertion with only boolean expression
      assert (principalAmount>0.0);
     assert (numberOfYears>0.0);
     assert (rateOfInterest>0.0);

return (principalAmount*numberOfYears*rateOfInterest)/100.0;
     }
```

If the method is invoked with the value in **`principalAmount`** less than 0.0, such as **`calculateInterest(-1200.5, 2.5, 3.45)`**, then it throws `AssertionError` exception as `java.lang.AssertionError`.

In this form, detailed information about the cause of assertion failure is not displayed.

Note: If the assumption is incorrect then using the `assert` statement will throw an error, otherwise the next statement of the program will be executed. When assertion fails, an `AssertionError` instance is created to print an exception with a textual message containing stack trace description. It is a subclass of `Error`. It has a default constructor and seven other parameterized constructors. In this form the default constructor is invoked, thus showing no detailed message about the exact cause of failure.

The second form, `assert booleanExpression : expression;` acts just like the first form. In addition, if the boolean expression evaluates to false, the second expression is returned as a detailed message for the `AssertionError`.

The message to be displayed is written in the second expression of the `assert` statement.

Figure 9.6 shows the second form of declaring assertions.

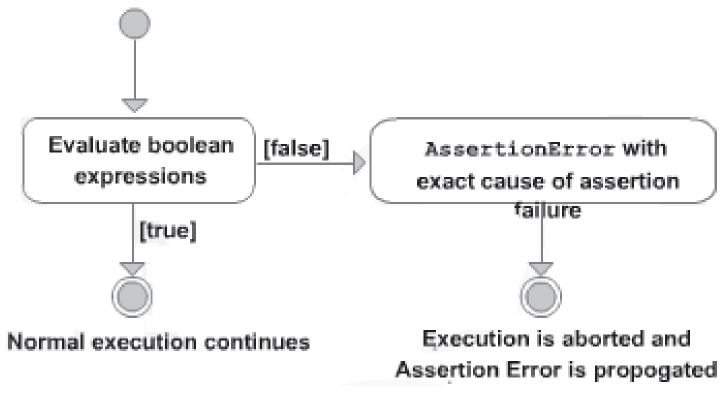

Evaluate boolean expressions
[false]
AssertionError with exact cause of assertion failure
[true]
Normal execution continues
Execution is aborted and Assertion Error is propogated

Figure 9.6: Using Assertions to Manipulate Variables

The following is the syntax for providing the second form of assertion.

Syntax:

```
assert expression1: expression2;
```

Where, expression1 is a boolean expression and expression2 is an expression having a value.

The following code demonstrates the second form of assertion with a boolean expression and a detailed message.

Code Snippet:

```
public class CalculateInterest{
public double calculateInterest(double principalAmount,double
numberOfYears,double rateOfInterest){
//Second form of assertion with boolean expression and value
assert (principalAmount>0.0):"Invalid Entry"+principalAmount;
assert (numberOfYears>0.0):"Invalid Entry"+numberOfYears);
assert (rateOfInterest>0.0):"Invalid Entry"+rateOfInterest;

return (principalAmount*numberOfYears*rateOfInterest)/100.0;
  }
}
```

If the method is invoked with the value in **`principalAmount`** less than 0.0, such as **`calculateInterest(-1200.5, 2.5, 3.45)`**, then it throws `AssertionError` exception as `java.lang.AssertionError: Invalid Entry -1200.5`.

The second form displays an error message with the exact value that caused the assertion failure.

Note: Using the second form of `assert` statement invokes other parameterized constructors that results in detailed description of error messages with exact reason of assertion failure. The value calculated in Expression2 is passed as an argument to the parameterized constructor of `AssertionError`.

9.4.2 Enabling Assertions

By default, assertions are disabled. The syntax for enabling assertion statement in Java source code is:

```
java -ea CalculateInterest
```

Or

```
java -enableassertions CalculateInterest
```

By default, assertions are turned off in NetBeans, whereas they are on by default when you compile source code with `javac`. Hence, in NetBeans the following steps need to be taken to turn runtime assertion checking on.

Follow the procedure to enable assertion in NetBeans:

1. Right-click the `Project` icon in the `Project panel` at the left.
2. Select `Properties` at the bottom of the drop-down menu.
3. Click the `Run` choice in the left panel.
4. In the `VM Options` text field at the right, enter `-ea`. `ea` stands for enable assertions. `VM` stands for Virtual Machine.

9.4.3 Disabling Assertions

The syntax for disabling assertion statement in Java source code is:

```
java -da CalculateInterest
```

Or

```
java -disableassertions CalculateInterest
```

Follow the procedure to disable assertion in NetBeans:

1. Right-click the `Project` icon in the `Project panel` at the left.
2. Select `Properties` at the bottom of the drop-down menu.
3. Click the `Run` choice in the left panel.
4. In the `VM Options` text field at the right, enter `-da`. `da` stands for disable assertions. `VM` stands for Virtual Machine.

9.4.4 Using Assertions

Assertion statements have advantages as well as disadvantages. Although, using assertion statements is recommended in most cases, in some cases, using these can be undesirable.

Appropriate uses of assertions are as follows:

- **Arguments to private methods**

  The following code snippet demonstrates the use of assertion on arguments to private methods.

  Code Snippet:

```
private void setPrincipalAmount(double pAmount){

// Appropriate use
assert pAmount > 0.0;

// Setting member variable
principalAmount=pAmount;
}
```

  `Assertion` is required in private method because it is not guaranteed that a private method's requirements are checked in advance.

- **Conditions at the beginning of any method**

  The following code snippet demonstrates the use of assertion using related conditions at the beginning of any method.

  Code Snippet:

```
private void calculateInterest(double pAmount, double nYears, double
rInterest){
   // appropriate use
   assert pAmount>0.0;
   assert nYears>0.0;
   assert rInterest>0.0;

   double interest=(pAmount*nYears*rInterest)/100.0;
}
```

➢ **Conditional cases**

The following code snippet demonstrates the use of assertions on arguments to public methods.

Code Snippet:

```
private int getVal() {
    if (x+y+z>10) {
        return 0;
    } else if (x+y>6) {
        return 1;
    } else {
        return 2;
    }
}
public void useGetVal() {
    int a = getValue(); // returns 0, 1, or 2
    if (a==0) {

    } else if (a==1) {

    } else if (a==2) {

    } else {
        //Will never reach else condition

        assert false : "Invalid value of a";
}
```

9.4.5 Inappropriate uses of Assertions

Inappropriate uses of assertions are as follows:

- **Using assertions on arguments to public methods**

  The following code shows the inappropriate use of assertions on arguments.

  Code Snippet:

```
public void setPrincipalAmount(double pAmount){
// Inappropriate use
assert pAmount > 0.0;

// Setting member variable
principalAmount=pAmount;
}
```

  This use is inappropriate because a `public` method writer is bound to check requirements of the function.

- **Using assertions on command line arguments**

  The following code shows the inappropriate use of assertions on command line arguments.

  Code Snippet:

```
public class CalculateInterest
{
static public void main( String args[] ) {
// Inappropriate use
assert args.length == 3;

int principal = Integer.parseInt( args[0] );
int noOfYears = Integer.parseInt( args[1] );
int rateOfInterest = Integer.parseInt( args[2] );
     }
}
```

  Use exceptions instead of assertion.

9.4.6 Reasons to use Assertions

Assertion statements can be used:

➢ **To replace comments**

The following code shows the use of assertions to replace comments to test the correctness of the assumptions.

Code Snippet:

```
for (i=0;i<10;i++)
 {
     if  (i%2 == 0)
       {
       }
     else if  (i%2== 1)
       {
       }

     else {
        // Assumption that i%2 will be 2
        assert i%2 == 2:i;
        // statements to be executed if i%2 == 2 evalutes to true.
        // It is an assumption that is generally placed in comments is
        // placed inside an assert statement to check the correctness
        //of the assumption
     }
 }
```

- **To replace code in default case**

The following code demonstrates the given `switch-case` statement in which it is assumed that **`numberOfYear`** will never reach 4. Assertion statement has been placed with default to check the correctness of assumption.

Code Snippet:

```
switch(numberOfYear){
case 1:
break;
case 2:
break;
case 3:
break;
default:
assert false: numberOfYear;
break;
}
```

- **To replace unreachable code**

The following code demonstrates the use of assertions to identify the unreachable statements in a code.

Code Snippet:

```
for (i=0;i<5;i++){
  if (i==3)
     return;
  // Unreachable code, an assertion statement has been placed below to
  // check correctness of assumption.
   assert false:i;
}
```

➢ **In the beginning of the method**

The following code demonstrates to use of assertions to check for some conditions that must be true, before the code in a method can be executed.

Code Snippet:

```
private    double    calculateInterest(double    principalAmount,double
numberOfYears,double rateOfInterest){
assert   principalAmount>0.0   &&   principalAmount   <20000.5   :”Invalid
Entry”+principalAmount;

return (principalAmount*numberOfYears*rateOfInterest)/100.0;
}
```

➢ **After method invocation**

Assertions can be used to check for conditions that must be true after the invoking a method. The following code checks the assumption that new value of length is greater than the old value after invoking a method.

Code Snippet:

```
newValue=changeLength(oldValue);
assert (newValue> oldValue);
```

➢ **To check object's state**

Assertions can be used to consistency of an object's state before and after invoking a method. The following code checks for inconsistency in object state after invoking `changeLength()` method.

Code Snippet:

```
newValue=changeLength(oldValue);
assert hasValidState():newValue;
```

9.4.7 Use of Internal Invariants

An invariant is an assumption that is always true. An internal invariant is believed to be true within some part of the program.

The following code snippet calculates the new value of principal amount depending on interest. The assumption in the program is that the interest value cannot be negative. This assumption can be tested for correctness using assertion.

Code Snippet:

```
calculatePrincipalAmount(double interest){
if (interest>5000.50)
    principalAmount = principalAmount + 500.5;
else if (interest>10000.50)
    principalAmount = principalAmount + 1000.5;
else
    // assumption: interest cannot be negative
    assert(interest>0.0):interest+”Invalid Interest entry”;
}
```

9.4.8 Using Control-flow Invariants

A control-flow invariant is an assumption that one cannot execute some statement in certain area of code, that is, some part of the code is unreachable. An assertion statement can be placed to test such codes.

The following code snippet shows an assertion statement that has been placed to test whether the code containing assertion is indeed unreachable.

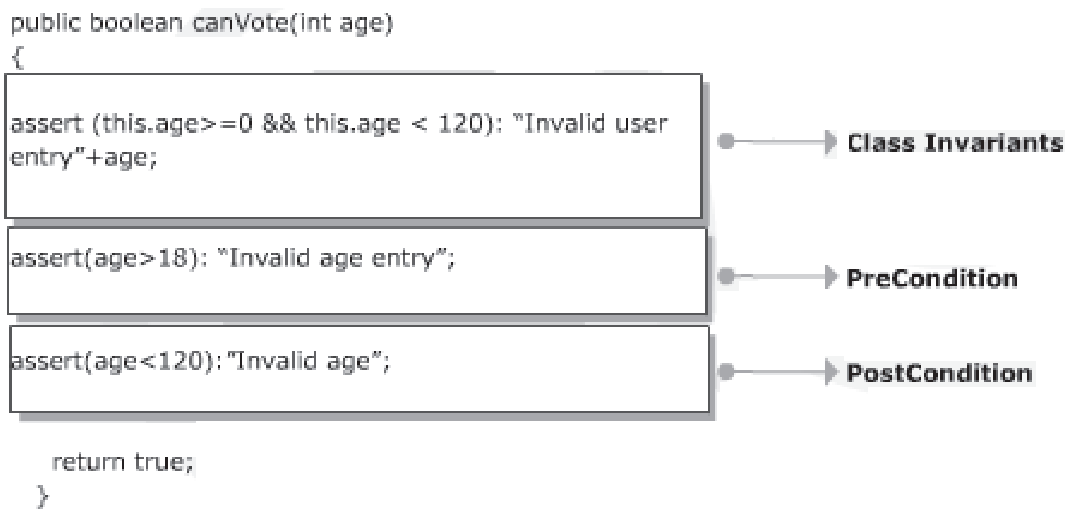
Code Snippet:

```
public double calculatePrincipal(double rateOfInterest){
  for (noOfYears=1.0; noOfYears<10.0; noOfYears++) {
                 if(interest=5000.0)
                       return interest;
                 }
//assumed to be unreachable. Assertion statement to test assumption
                 assert false;
     }
}
```

9.4.9 Design By Contract

The correctness of a program can be viewed as proof that the computation has been provided with correct input and has terminated with correct output. The method that invokes the computation has the responsibility of providing the correct input. This is known as precondition. If the computation is successful, then the computation is said to have satisfied the postcondition. Besides these two conditions, there is one more important invariant called class invariant.

Figure 9.7 shows the design by contract that comprises three constituents, also known as roles.



```
public boolean canVote(int age)
{
assert (this.age>=0 && this.age < 120): "Invalid user
entry"+age;
assert(age>18): "Invalid age entry";
assert(age<120):"Invalid age";
  return true;
 }
```

Figure 9.7: Design by Contract

The following code demonstrates class invariants, precondition, and postcondition assertions.

Code Snippet:

```
public class HumanBeing{
// method for testing class invariant
// checks if user entry of age falls within lifetime of human beings
validateAge(int age)
{
// class invariant
assert (person.age >= 0 && person.age < 120): "Invalid user entry"+age;
}
```

```
public class HumanBeing{
// method for testing class invariant
// checks if user entry of age falls within lifetime of human beings
validateAge(int age)
{
// class invariant
assert (person.age >= 0 && person.age < 120): "Invalid user entry"+age;
}

public boolean canVote(int age)
{
// validating for every instance of the class
validateAge(age);

// precondition
assert(age>18): "Invalid age entry for vote";
// postcondition;
assert(age<120):"Invalid age";
   return true;
   }
}
```

Note:

Precondition:

A condition that the invoker of an operation agrees to satisfy.

Postcondition:

A condition that the invoked method promises to achieve.

Class Invariant:

A class invariant is a type of internal invariant applicable to each instance of a class. This is not applicable when there is a change in state of an instance. A class invariant specifies the relationships among multiple attributes of a class that its instances must satisfy before and after each method call. For example, consider the assertion statement for person instance of `HumanBeing` class, `assert person.age >= 0 && person.age < 120;`

9.4.10 Class AssertionError

`AssertionError` contains a default constructor and seven single-parameter constructors. The assert statement's single-expression form uses the default constructor, whereas the two-expression form uses one of the seven single-parameter constructors.

To understand which `AssertionError` constructor is used, consider how assertions are processed when enabled.

Evaluate expression1

- **If true**
  - No further action
- **If false**
  - And if expression2 exists

    Evaluate expression2 and use the result in a single-parameter form of the `AssertionError` constructor
  - Else

    Use the default AssertionError constructor

Since, the assert statement in class `Excep` uses the single-expression form, the violation triggered in passing -1 to method **`m1(int)`** prompts the use of the default `AssertionError` constructor. The default constructor effectively uses `java.lang.Throwable` default constructor to print an exception message that includes a textual stack trace description.

The assertion error output from running class `Excep` is lacking. It is seen that an `AssertionError` occurs in method **`m1`** at line 6, but the output does not describe what went wrong. Fortunately, the assert statement's two-expression form provides this facility. As noted earlier, in the two-expression form, when expression1 evaluates as `false`, the assertion facility passes the result of evaluating expression2 to a single-parameter `AssertionError` constructor. Expression2 effectively acts as a `String` message carrier, meaning `AssertionError`'s single-parameter constructors must convert the result of expression2 to a `String`.

The following class **`Bar`**, uses the two-parameter form for a simple assertion in method **`m1(int)`**.

Code Snippet:

```
public class Bar {
    public void m1( int value ) {
        assert 0 <= value : "Value must be non-negative: value= " + value;
        System.out.println( "OK" );
    }

    public static void main( String[] args ) {
        Bar bar = new Bar();
        System.out.print( "bar.m1(  1 ): " );
        bar.m1( 1 );
        System.out.print( "bar.m1( -1 ): " );
        bar.m1( -1 );
    }
}
```

Bar's output with assertions enabled is as follows:

```
bar.m1(  1 ): OK
bar.m1( -1 ): Exception in thread "main" java.lang.AssertionError: Value
must be non-negative: value= -1
        at Bar.m1(Bar.java:6)
        at Bar.main(Bar.java:17)
```

The output shows the String conversion of the expression2 result concatenated to the end of the exception message, before the textual stack trace. The detailed message certainly improves the exception message's usability. Since, creating a reasonable error message is not difficult, developers should favor the assert statement's two-expression form.

The `java.lang.Throwable` class enables a cleaner formatting of stack trace information.

The following code class **FooBar**, uses these new capabilities to format the exception message produced by the assertion error.

Code Snippet:

```
public class FooBar {
    public void m1( int value ) {
        assert 0 <= value : "Value must be non-negative: value= " + value;
        System.out.println( "OK" );
    }
    public static void printAssertionError( AssertionError ae ) {
        StackTraceElement[] stackTraceElements = ae.getStackTrace();
        StackTraceElement stackTraceElement = stackTraceElements[ 0 ];
        System.err.println( "AssertionError" );
       System.err.println( "   class=   " + stackTraceElement.getClassName()
);
      System.err.println( "   method=  " + stackTraceElement.getMethodName()
);
        System.err.println( "   message= " + ae.getMessage() );
    }
    public static void main( String[] args ) {
        try {
            FooBar fooBar = new FooBar();
            System.out.print( "fooBar.m1(  1 ): " );
            fooBar.m1( 1 );
            System.out.print( "fooBar.m1( -1 ): " );
            fooBar.m1( -1 );
        } catch( AssertionError ae ) {
            printAssertionError( ae );
        }
    }
}
```

The following output of running **FooBar** with assertions enabled displays the cleaner `AssertionError` reporting:

```
fooBar.m1(  1 ): OK
fooBar.m1( -1 ): AssertionError
   class=   FooBar
   method=  m1
   message= Value must be non-negative: value= -1
```

9.4.11 Comparing Assertion and Exceptions

The frequent dilemma that new developers face is deciding when to use assertions and when to use exceptions. Both catch problems in the program, but the intended usage is very different. This is how exceptions differ in use from Assertions:

- An exception tells the user of the program that something went wrong.
- An assertion documents some assumption about the program.
- When an assertion fails, it points out a bug in the programming logic.
- Exceptions are created to deal with problems that might occur during the course of execution of the program.
- Assertions are written to state assumptions made by the program.

Knowledge Check 4

1. Can you match the type of assertions with their corresponding description?

	Description		Type
(A)	Assertions to test unreachable code using	1	Control-flow invariant
(B)	Using assertions to test control in a loop	2	Will invoke AssertError constructor with detailed assertion failure message
(C)	`assert (i<0):"Invalid value"+i`	3	Will print only brief assertion failure message
(D)	`assert (i<0)`	4	Internal invariant
(E)	An assumption is believed to be true within some part of the program	5	Control-flow invariant

2. Can you match the terms with their corresponding description?

	Description		Term
(A)	Design by contract consist of	1	Postcondition
(B)	Checks correct input before a method is invoked	2	Precondition, postcondition, and Class Invariant
(C)	Checks if a method has completed successfully	3	Precondition
(D)	Applicable to testing change in state of an object	4	Class Invariant
(E)	Defines relationship of attributes in a class that must be satisfied by its instances	5	Class Invariant

Module Summary

In this module, **Exceptions**, you have learnt about:

- **Introduction to Exceptions**

  Exceptions report abnormal conditions. Java supports exception mechanism. Exceptions can occur in APIs, in client code and be due to resource failures. Java supports two types of exceptions: `checked` and `unchecked`. `Checked`. Exceptions are derived from the `Exception` class, and need to be handled in a program, whereas `unchecked` exceptions are derived from the `RuntimeException` class, and need not be handled in a program.

- **Exception handling in Java**

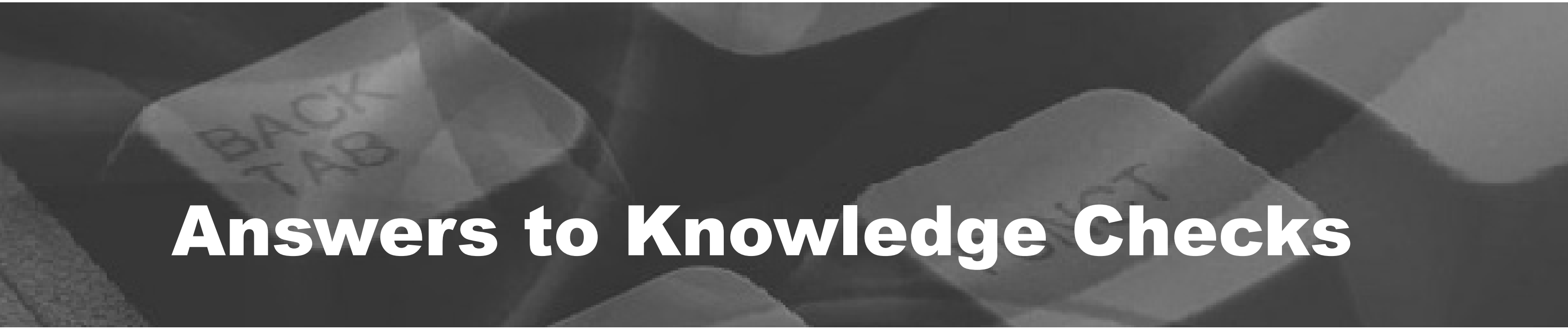
  Java supports exception handling mechanism by using `try-catch-finally` block. Statements that may raise exceptions are placed inside a `try` block. Exceptions are handled in a single or multiple `catch` blocks, and if any cleanup code is required that needs to be executed in the case of unexpected exceptions, then it is placed inside a `finally` block.

- **User-defined Exceptions**

  User-defined exceptions are derived from the `Throwable` base class. Any type of exceptions, predefined or user defined, can be raised in Java code. Exception handling is achieved using `try-catch` block.

- **Assertion**

  Assertions test the correctness of assumptions in a program. Assertions help in removing bugs from a program. Assertions can be effectively used in checking consistency in preconditions, postconditions, and class invariants.



Answers to Knowledge Checks

Module 1

Knowledge Check 1:

1. (A)–Method, (B)–Object, (C)–Field, (D)–Class, and (E)–Class

Knowledge Check 2:

1. (A)-True, (B)–True, (C)–True, (D)–False, and (E)–False
2. (A)–(4), (B)–(3), (C)–(2), (D)–(5), and (E)–(1)

Knowledge Check 4:

1. (A)-True, (B)–True, (C)-False, (D)–True, and (E)-False

Knowledge Check 5:

1. (A)–(4), (B)–(3), (C)–(1), (D)–(5), and (E)–(2)

Knowledge Check 6:

1. (A)-False, (B)–True, (C)-False, (D)–True, and (E)-True

Module 2

Knowledge Check 1:

1. (A)-(True), (B)-(True), (C)-(False), (D)-(True), and (E)-(True)

Knowledge Check 2:

1. (A)–(4), (B)-(5), (C)-(2), (D)-(1), and (E)-(3)

Knowledge Check 3:

1. (B)
2. (A)-True, (B)–False, (C)–False, (D)–False, and (E)-True

Knowledge Check 4:

1. (A)–(3), (B)-(5), (C)-(4), (D)-(1), and (E)-(2)
2. (B)

Knowledge Check 5:

1. (3), (5), (4), (1), (2)

Module 3

Knowledge Check 1:

1. (D)
2. (C)

Knowledge Check 2:

1. (C), (E), (D), (A), (B)
2. (A)-True, (B)-True, (C)-False, (D)-True, and (E)-False
3. (C)
4. (A)-(2), (B)-(3), (C)–(1), and (D)-(2)

Knowledge Check 3:

1. (C)

Module 4

Knowledge Check 1:

1. (A)-True, (B)-True, (C)-False, (D)-False, and (E)-True
2. (A)-False, (B)-False, (C)-True, (D)-True, and (E)-False

Knowledge Check 2:

1. (E), (B), (A), (D), (C)

Knowledge Check 3:

1. (B)

Knowledge Check 4:

1. (A)-True, (B)-True, (C)-False, (D)-False, and (E)-True

Module 5

Knowledge Check 1:

1. (B)

2. (C)

Knowledge Check 2:

1. (A)–(3), (B)-(1), (C)–(2), (D)-(5), and (E)-(4)

2. (A), (C)

Knowledge Check 3:

1. (A)-(2), (B)-(4), (C)–(5), (D)-(1), and (E)–(3)

Module 6

Knowledge Check 1:

1. (A)–True, (B)-False, (C)-True, (D)-False, and (E)-True

2. (A)-(True), (B)-(False), (C)-(True), (D)-(False), and (E)-(False)

Knowledge Check 2:

1. (A)-(default), (B)-(protected), (C)-(private), (D)-(public), and (E)-(protected)

2. (B)

Knowledge Check 3:

1. (A)-True, (B)-False, (C)-True, (D)-False, and (E)-True

2. (A)-True, (B)-True, (C)-True, (D)-False, and (E)-False

Module 7

Knowledge Check 1:

1. (A)-False, (B)-True, (C)-True, (D)-True, and (E)-False

2. (A)-True, (B)-True, (C)-False, (D)-False, and (E)- False.

Knowledge Check 2:

1. (B)

Knowledge Check 3:

1. 4, 2, 3, 1, 5

Knowledge Check 4:

1. (B)

Knowledge Check 5:

1. (C)

Module 8

Knowledge Check 1:

1. (A)
2. (B)

Knowledge Check 2:

1. (A)–(True), (B)–(False), (C)-(True), (D)–(True), and (E)–(False)

Knowledge Check 3:

1. (A)–(3), (B)–(1), (C)–(2), (D)–(5), and (E)–(4)
2. (A)-(True), (B)-(True), (C)–(False), (D)–(True), and (E)–(False)

Module 9

Knowledge Check 1:

1. (A)–False, (B)-True, (C)-True, (D)-True, and (E)-False
2. (A)-(2), (B)-(4), (C)-(1), (D)-(5), and (E)-(3)

Knowledge Check 2:

1. (A)
2. 4, 1, 5, 2, 3
3. (A)-False, (B)-True, (C)-False, (D)-False, and (E)-True

Knowledge Check 3:

1. (A)-False, (B)-True, (C)-False, (D)-True, and (E)-False
2. (D)

Knowledge Check 4:

1. (A)-5, (B)-1, (C)-2, (D)–3, and (E)-4
2. (A)-(2), (B)-(3), (C)-(1), (D)–(5), and (E)–(4)