

Java Programming - The Complete Guide for Beginners



Java Programming - The Complete Guide for Beginners

© 2023 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

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

First Edition - 2023



Preface

The book, **Java Programming - The Complete Guide for Beginners**, aims to teach the basics of the 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.

The Java programming language has undergone several reformations since its inception. This Learner's Guide intends to familiarize the reader with the latest version of Java, that is, Java SE 20. The book begins with an introduction to the basic concepts of Java programming language and the NetBeans IDE. The book proceeds with the explanation of various Java features and constructs such as classes, methods, objects, loops, inheritance, interfaces, and so on. It also covers various enhancements that have been made to existing features.

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.

Design Team



Onlinevarsity App for Android devices

Download from Google Play Store

Table of Contents

Sessions

Session 01: Introduction to Java

Session 02: Variables, Data Types, and Operators

Session 03: Decision-Making Constructs and Loops

Session 04: Classes, Objects, and Methods

Session 05: Arrays and Strings

Session 06: Modifiers and Packages

Session 07: Inheritance and Polymorphism

Session 08: Interfaces and Nested Classes

Session 09: Exceptions

Session 10: Date and Time API

Session 11: Additional Features of Java

Session 12: JDK 20 New Features and Functionalities

Appendix

Onlinevarsity

INDUSTRY BEST PRACTICES

SYNC WITH THE INDUSTRY



Session - 1

Introduction to Java

Welcome to the Session, **Introduction to Java**.

This session explains various methodologies that have been adopted over a period of time for solving problems and developing applications. It proceeds to introduce object-oriented paradigm as a solution to develop applications that are modeled to real-world entities, that is, objects. Further, the session explains concept of an object and a class in Object-Oriented Programming (OOP). It introduces Java as an OOP language and as a platform to develop platform independent applications. Finally, the session explains various components of Java platform.

In this Session, you will learn to:

- Explain structured programming paradigm
- Explain object-oriented programming paradigm
- Explain features of Java as a OOP language
- Describe Java platform and its components
- List different editions of Java
- Explain evolution of Java Standard Edition (Java SE)
- Describe steps for downloading and installing Java Development Kit (JDK)



1.1 Structured Programming Paradigm

Earlier programming languages, such as C, Pascal, Cobol, and so forth followed structured programming paradigm. In structured programming paradigm, application development is decomposed into a hierarchy of subprograms. Figure 1.1 displays an application developed for a bank with some of the bank activities broken down into subprograms.

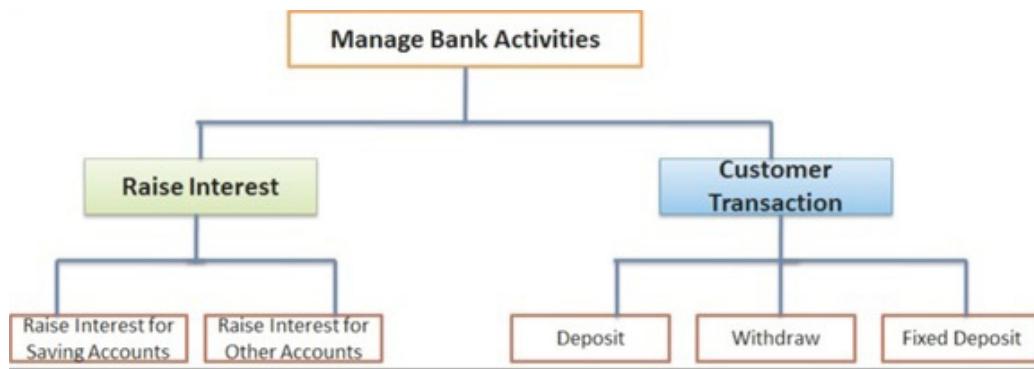


Figure 1.1: Decomposing of Problem into Subprograms

Subprograms are referred to as procedures, functions, or modules in different structured programming languages. Each subprogram is defined to perform a specific task. Thus, to manage various bank processes, the software application is broken down into subprograms that will interact with each other to solve business requirements.

The main emphasis of structured programming languages was to decompose applications into procedures to solve complex problems, whereas, data was given less importance. In other words, efficiency of programs was measured on time and memory occupancy, rather than on how to control data shared globally between procedures. Hence, most of the development efforts in structured programming languages were spent on accomplishing the solution rather than focusing on the problem domain.

This also often led to software crisis, as the maintenance cost of complex problems became high and the availability of reliable software was reduced.

Figure 1.2 shows handling of data in structured programming languages.

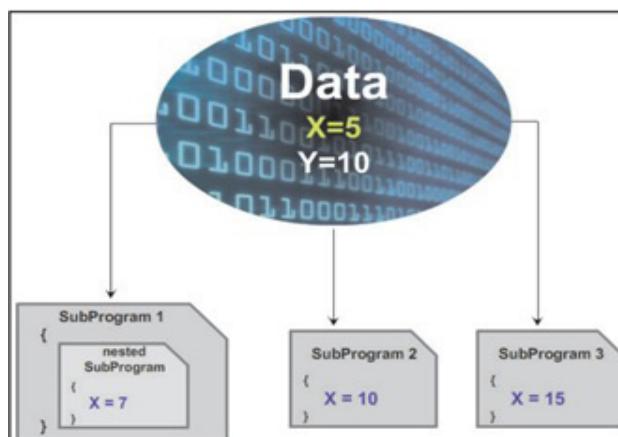


Figure 1.2: Data Shared Between Subprograms

1.2 New Paradigm: Object-oriented Programming Paradigm

The growing complexity of software required a change in the programming style. Some of the features that were aimed for are as follows:

- Development of reliable software at reduced cost.
- Reduction in the maintenance cost.
- Development of reusable software components.
- Completion of the software development within the specified time interval.

These features resulted in the evolution of object-oriented programming paradigm.

Object-oriented programming paradigm enables you to develop complex software applications for different domain problems with reduced cost and high maintenance. Software applications developed using object-oriented programming paradigm is designed around data, rather than focusing only on functionalities.

Object-oriented programming paradigm basically divides the application development process into three distinct activities. These are as follows:

Object-oriented Analysis (OOA) – This process determines functionality of the system. In other words, it determines purpose of developing an application.

Object-oriented Design (OOD) – This is the process of planning a system in which objects interact with each other to solve a software problem.

Object-oriented Programming (OOP) – This is the process that deals with actual implementation of the application.

Figure 1.3 shows different activities involved in the object-oriented software development.

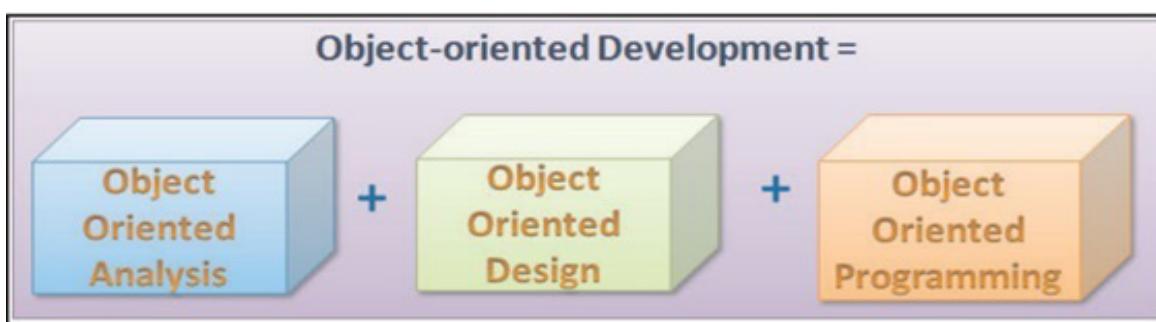


Figure 1.3: Object-oriented Software Development

As shown in Figure 1.3, the OOA and OOD phases deal with the analysis and designing of the software application. These activities are carried out manually using a modeling language, Unified Modeling Language (UML). The UML contains graphic notations that help to create visual models in the system. The actual implementation of these visual models is done using an OOP language.

An OOP language is based on certain principles that are as follows:

- **Object** – Represents an entity which possesses certain features and behaviors.
- **Class** – Is a template that is used to create objects.
- **Abstraction** – Is a design technique that focuses only on the essential features of an entity for a specific problem domain.
- **Encapsulation** – Is a mechanism that combines data and implementation details into a single unit called class. It also secures the data through a process called data hiding where a user cannot manipulate the data directly.
- **Inheritance** – Enables the developer to extend and reuse features of existing classes and create new classes. The new classes are referred to as derived classes.
- **Polymorphism** – Is the ability of an object to respond to same message in different ways.

The main building blocks of an OOP language are classes and objects. They allow you to view the problem from user's perspective and model the solution around them. Hence, a clear understanding of classes and objects is important for application development using OOP languages.

1.2.1 Concept of an Object

An object represents a real-world entity. Any tangible or touchable entity in the real-world can be described as an object.

Figure 1.4 shows some real-world entities that exist around everyone.

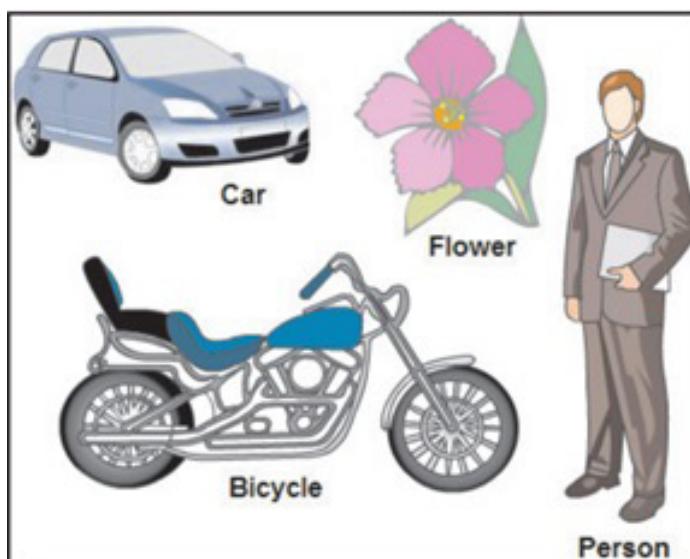


Figure 1.4: Real-world Entities

As shown in Figure 1.4, the real-world entities, such as a car, flower, bike, or person are treated as objects. Each object has some characteristics and is capable of performing certain actions. Characteristics are defined as attributes, properties, or features describing the object, while actions are defined as activities or operations performed by the object.

For example, the properties of an object, **Dog**, are as follows:

- Breed
- Color
- Age

An object also executes actions. Thus, the actions that a **Dog** can perform are as follows:

- Barking
- Eating
- Running

1.2.2 Defining a Class

In the real-world, several objects have common state and behavior. For example, all car objects have attributes, such as color, make, or model. These objects can be grouped under a single class. Thus, it can be defined that a class is a template or blueprint which defines the state and behavior for all objects belonging to that class.

Figure 1.5 shows a car (any car in general) as a class and a Toyota car (a specific car) as an object or instance created for that class.

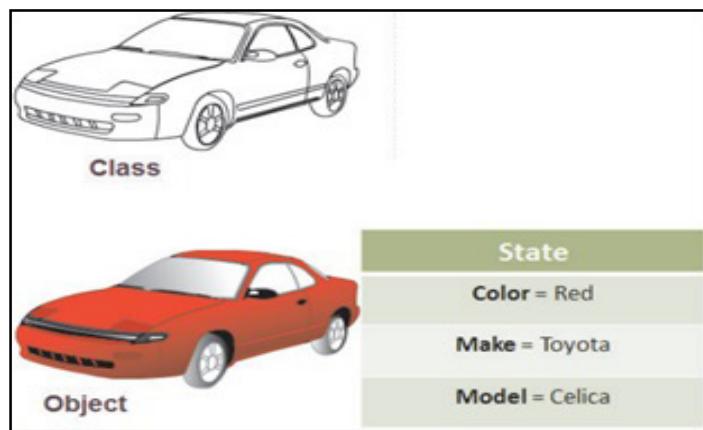


Figure 1.5: Class and an Object

Typically, in an OOP language, a class comprises fields and methods, collectively called as members. The fields are known as variables and depict state of objects. Methods are known as functions and depict behavior of objects.

Figure 1.6 shows the design of a class with fields and methods in an OOP language.

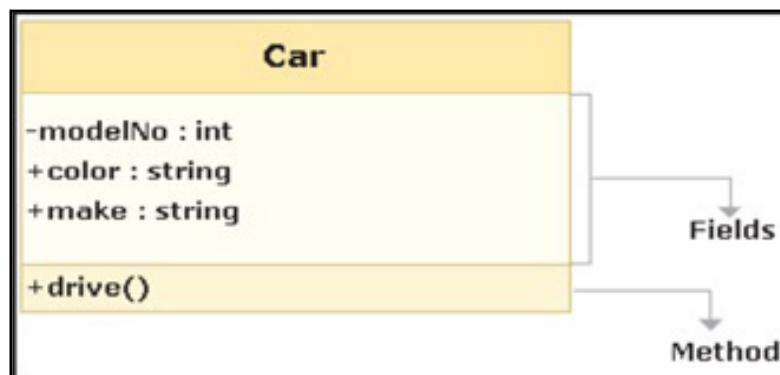


Figure 1.6: Class Design

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

1.3 Introduction to Java

Java is a programming language and a platform. Java is a high level, robust, object-oriented, and secure programming language. It is also one of the most popular OOP languages. It helps programmers to develop a wide range of applications that can run on various hardware and Operating System (OS). Java is also a platform that creates an environment for executing Java application.

Java was initially developed to cater to small-scale problems, but was later found capable of addressing large-scale problems across the Internet. Very soon, it gained in popularity and was adopted by millions of programmers all across the world. Today, Java applications are built for a variety of platforms that range from embedded devices to desktop applications, Web applications to mobile phones, and from large business applications to supercomputers.

Where is Java Used?

Java is a very versatile and popular programming language that can be used for various purposes. Some of the common uses of Java are:

- **Android mobile apps:** Java is the official language for Android mobile app development. In fact, the Android operating system itself is written in Java. Most of the Android applications are built using Java, such as X (formerly called Twitter) and Instagram.
- **Desktop GUI applications:** Java can also be used to create Graphical User Interface (GUI)

applications for desktop computers. Java provides several libraries, such as Abstract Window Toolkit (AWT), Swing, and JavaFX, that offer pre-built components such as buttons, menus, and form fields that you can use to design your GUI.

- **Web-based applications:** Java was one of the first languages to support Web development by providing applets that could run in a Web browser. Although applets are no longer used, Java is still widely used for creating back-end Web applications that run on a Web server. Java offers various technologies, such as Servlets, Java Server Pages (JSP) and Struts that enable you to create dynamic Web pages and handle user requests.
- **Game development:** Java is also a popular choice for game development because it supports the open-source 3D engine called jMonkeyEngine. This engine provides a powerful framework for creating 3D games in Java that can run on multiple platforms.
- **Big Data technologies:** Java is also widely used for processing and analyzing large amounts of data using Big Data technologies. One of the most popular tools for Big Data is Hadoop, which is a platform that allows distributed storage and processing of data using clusters of computers. Hadoop is written in Java and uses the MapReduce paradigm to perform parallel computations on data.
- **Distributed applications:** Java can also be used to create distributed applications that run on multiple machines and communicate with each other over a network. Java provides the Remote Method Invocation (RMI) mechanism that allows objects on different machines to invoke methods on each other. Java also supports the Enterprise JavaBeans (EJB) technology that enables developers to create scalable and secure business components that can be deployed on a server.
- **Cloud-based applications:** Java is also a suitable language for developing cloud-based applications that run on remote servers and provide services over the Internet. Java offers various frameworks and libraries that simplify the development and deployment of cloud applications, such as Spring Boot, Micronaut, and Quarkus. Some of the popular cloud platforms that support Java are Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure.

1.4 History of Java

The brief timeline of Java is shown in Table 1.2.

Version	Release Date	Features and Changes
JDK Beta	1995	The very first version of Java, designed for interactive television and digital devices.
JDK 1.0	January 23, 1996	The first stable version of Java, also known as Java 1. It introduced the AWT, applets, and the Java Virtual Machine (JVM).

Version	Release Date	Features and Changes
Java Standard Edition (SE) 5	September 30, 2004	It added generics, annotations, autoboxing/unboxing, enums, varargs, enhanced <code>for</code> loop, static import, and concurrency utilities.
Java SE 6	December 11, 2006	It added scripting language support, Java Database Connectivity (JDBC) 4.0, compiler Application Programming Interface (API), pluggable annotations, and improved Web services support.
Java SE 7	July 28, 2011	It added <code>switch</code> expressions with strings, <code>try-with-resources</code> statement, diamond operator, binary literals, underscores in numeric literals, and invokedynamic bytecode instruction.
Java SE 8	March 18, 2014	It added lambda expressions, streams API, default methods in interfaces, functional interfaces, method references, Nashorn JavaScript engine, and date/time API.
Java SE 9	September 21, 2017	It added modules system (Project Jigsaw), JShell (REPL), private methods in interfaces, Hypertext Transfer Protocol (HTTP)/2 client API, and multi-release JARs.
Java SE 10	March 20, 2018	It added local variable type inference (<code>var</code> keyword), application Class-Data Sharing (CDS), thread-local handshakes, and experimental Graal JIT compiler.
Java SE 11	September 25, 2018	It added lambda parameters with <code>var</code> keyword, dynamic class-file constants (<code>condy</code>), nest-based access control (<code>nestmates</code>), HTTP client API (standard), Epsilon garbage collector (experimental), and Z garbage collector (experimental). It also removed Java Enterprise Edition (EE) modules and deprecated Nashorn JavaScript engine.
Java SE 12	March 19, 2019	It added switch expressions (preview), raw string literals (preview), default CDS archives, JVM constants API, low-pause-time garbage collector Shenandoah, microbenchmark suite, one AArch64 port, and deprecated the pack200 tools.
Java SE 13	September 17, 2019	It added text blocks (preview, which means it was not stable yet), dynamic CDS archives, Z Garbage Collector (ZGC) improvements, re-implemented the legacy Socket API, and deprecated the Remote Method Invocation (RMI) activation mechanism.
Java SE 14	March 17, 2020	The update included pattern matching for <code>instanceof</code> , records, text blocks, switch expressions, helpful <code>NullPointerExceptions</code> , packaging tool, and Non Uniform Memory Access aware memory allocation for G1. It also introduced support for ZGC on macOS and Windows, deprecated Solaris and Scalable Process Architecture (SPARC) ports, and removed the Concurrent Mark Sweep garbage collector.

Version	Release Date	Features and Changes
Java SE 15	September 15, 2020	The update introduced sealed classes in preview, hidden classes, text blocks as a standard feature, and pattern matching for instanceof in standard mode. It also included the Edwards-Curve Digital Signature Algorithm (EdDSA) and marked ZGC as a production-ready feature. Additionally, the update removed the Nashorn JavaScript engine.
Java SE 16	March 16, 2021	The update included records as standard, sealed classes in their second preview, and introduced pattern matching for switch in preview. It includes vector API, foreign linker API, and foreign-memory access API as incubator features, along with Unix-domain socket channels, and extended ZGC support for Alpine Linux and 64-bit ARM. Furthermore, it introduced a Windows/AArch64 port and deprecated the Applet API.
Java SE 17	September 14, 2021	The update introduced sealed classes as standard and pattern matching for switch in the second preview. It also added the foreign-memory access API and the vector API in their respective incubator stages, along with context-specific deserialization filters. Additionally, the update featured a new macOS rendering pipeline, improved pseudo-random number generators, and removed the Security Manager and the RMI activation mechanism.
Java SE 18	March 15, 2022	It added pattern matching for switch (standard), foreign-memory access API (standard), vector API (third incubator), foreign linker API (third incubator), and deprecated the Swing Application Framework.
Java SE 19	September 20, 2022	Key features in this version encompassed virtual threads, structured concurrency, and Record patterns in preview. It also introduced pattern matching for switch in the third preview and the foreign function and memory API. Furthermore, the update brought the vector API in its fourth incubator stage and expanded platform support with the LINUX/Reduced Instruction Set Computer (RISC)-V port.
Java SE 20	March 21, 2023	It has scoped values (incubator), foreign function and memory API (preview), virtual threads (preview), and structured concurrency (incubator).

Table 1.2: History of Java

Since James Gosling was the key member of the team that developed the language, he is known as the father of Java.

1.5 Java Platform and Its Components

Java platform is a combination of hardware and software which creates an environment for the execution of an application. The Java platform provides an environment for developing applications that can be executed on various hardware and OS.

Various components of Java platform are as follows:

- **Java Virtual Machine (JVM)** – Programming languages, such as C and C++, translate the compiled code into an executable binary code, which is machine dependent. In other words, the resulting executable code is for a specific microprocessor that executes it. In Java, the compiled code is not translated into an executable code.

It is instead compiled and converted into a bytecode that is an intermediate form closer to machine representation. Java bytecode is an optimized set of instructions executed by the Java runtime environment. This environment is known as JVM.

JVM is an execution engine that creates a platform independent execution environment for executing Java compiled code. There are different implementations of JVM available for different platforms, such as Windows, Unix, and Solaris. Thus, the execution of same bytecode by different implementations of JVM on various platform results in code portability and platform independence.

Figure 1.7 shows the execution of same bytecode with different implementations of JVM on various platforms.

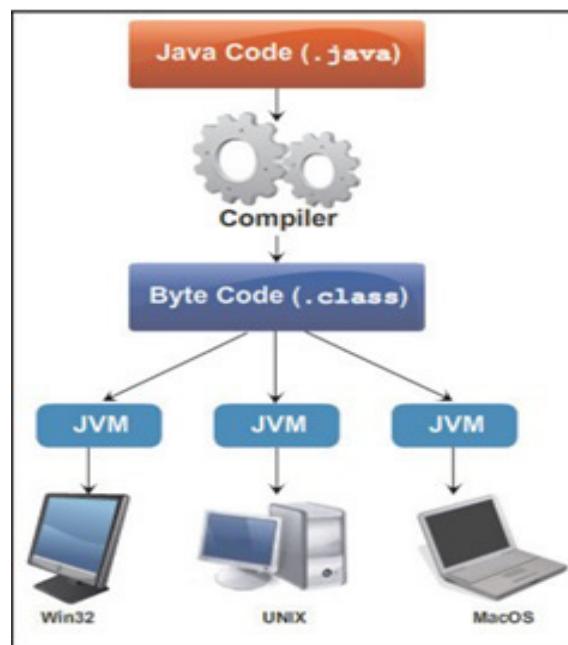


Figure 1.7: Bytecode Execution on Different JVMs

- **Java Runtime Environment (JRE)** - Java Runtime Environment is also written as Java Runtime Environment (RTE). The JRE is a set of software tools that are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime. The implementation of JVM is also actively released by other companies besides Sun Micro Systems.
- **Java Development Kit (JDK)** – JDK is a binary software development kit released by Oracle Corporation as part of the Java platform. It is an implementation of Java and distributed for various platforms, such as Windows, Linux, Mac OS X, and so on. It contains a comprehensive set of tools, such as compilers and debuggers that are used to develop Java applications.

JDK contains a private JVM and a few other resources such as an interpreter, a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and so on.

1.5.1 Differences between Java SE, JRE, and JDK

Table 1.3 lists differences between JRE and Java SE.

	JRE	Java SE
Who requires it?	Computer users who run applications written using Java technology	Software developers who write applications using Java technology
What is it?	An environment required to run applications written using Java programming language	A software development kit used to write applications using Java programming language
How do you get it?	Distributed freely and is available from: java.com	Distributed freely and is available from: oracle.com/javase

Table 1.3: Differences Between JRE and Java SE

Table 1.4 lists differences between JRE and JDK.

JRE	JDK
Implementation of the JVM which actually executes Java programs.	A bundle of software that you can use to develop Java based applications.
JRE is a plug-in required for running Java programs.	Java Development Kit is required for developing Java applications.
JRE is smaller than the JDK so it requires less disk space.	JDK requires more disk space as it contains JRE along with various development tools.
JRE can be downloaded/supported freely from: java.com	JDK can be downloaded/supported freely from oracle.com/technetwork/java/javase/downloads/
It includes the JVM, Core libraries, and other additional components to run applications written in Java.	It includes the JRE, set of API classes, Java compiler, Webstart, and additional files required to write Java applications.

Table 1.4: Differences Between JRE and JDK

1.5.2 OpenJDK and Oracle JDK

OpenJDK refers to a free and open-source implementation of Java. The initial release of OpenJDK was in 2007. Oracle JDK has had better performance than OpenJDK. However, the performance of OpenJDK is growing. Contributions of the OpenJDK community often outperform Oracle JDK. Both OpenJDK and Oracle JDK are created and maintained currently by Oracle only.

1.6 Class Data Sharing

Class Data Sharing (CDS) feature helps reduce the startup time and memory footprint between multiple JVMs.

When you use the installer to install the Oracle JRE, the installer loads a default set of classes from the system Java Archive (JAR) file into a private internal representation. It then dumps that representation to a file called a shared archive. If JRE installer is not being used, then, you can generate the shared archive manually.

When the JVM starts, the shared archive is memory-mapped to allow sharing of read-only JVM metadata for these classes among multiple JVM processes. Since, accessing shared archive is faster than loading classes, startup time is reduced.

The memory for applications comprises two components namely, stack and heap. The stack is an area in the memory that stores object references and method information. The heap area of memory deals with dynamic memory allocations. The heap memory grows as and when the physical allocation is done for objects. Hence, JVM provides a garbage collection routine that frees the memory by destroying objects that are no longer required in Java program. Java provides several different types of garbage collectors, such as G1, serial, parallel, and parallelOldGC garbage collectors.

1.7 Downloading and Installing the JDK

To download the JDK, follow the link <https://www.oracle.com/java/technologies/javase/jdk20-archive-downloads.html>.

→ Step 1: Click the required version of JDK as shown in Figure 1.8.

Linux x64 Compressed Archive	183.11 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_linux-x64_bin.tar.gz (sha256)
Linux x64 Debian Package	155.91 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_linux-x64_bin.deb (sha256)
Linux x64 RPM Package	182.82 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_linux-x64_bin.rpm (sha256)
macOS Arm 64 Compressed Archive	177.21 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_macos-aarch64_bin.tar.gz (sha256)
macOS Arm 64 DMG Installer	176.54 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_macos-aarch64_bin.dmg (sha256)
macOS x64 Compressed Archive	179.54 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_macos-x64_bin.tar.gz (sha256)
macOS x64 DMG Installer	178.87 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_macos-x64_bin.dmg (sha256)
Windows x64 Compressed Archive	180.99 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_windows-x64_bin.zip (sha256)
Windows x64 Installer	160.12 MB	https://download.oracle.com/java/20/archive/jdk-20.0.2_windows-x64_bin.exe (sha256)

Figure 1.8: JDK Versions

→ **Step 2:** Run the JDK Installer.

User requires administrator privilege to install the JDK on Microsoft Windows.

To run the JDK installer:

- Start the JDK 20 installer by double-clicking the installer's icon or file name in the download location.
- Follow the instructions provided by the Installation wizard.
- The JDK includes the JavaFX Software Development Kit (SDK), a private JRE, and the Java Mission Control tools suite. The installer integrates the JavaFX SDK into the JDK installation directory.
- After the installation is complete, delete the downloaded file to recover the disk space.

1.7.1 Setting the PATH Environment Variable

It is useful to set the PATH variable permanently for JDK 20 so that it is persistent after rebooting.

If you do not set the PATH variable, then, you must specify the full path to the executable file every time that you run it.

For example,

```
C:\> "C:\Program Files\Java\jdk-20\bin\javac" MyClass.java
```

To set the PATH variable permanently, add the full path of the jdk-20\bin directory to the PATH variable. Typically, the full path is:

```
C:\Program Files\Java\jdk-20\bin
```

To set the PATH variable on Microsoft Windows:

- Select Control Panel and then, System.
- Click Advanced and then, Environment Variables.
- Add the location of the bin folder of the JDK installation to the PATH variable in System Variables.

Note: The PATH environment variable is a series of directories separated by semicolons (;) and is not case-sensitive. Microsoft Windows looks for programs in the PATH directories in order, from left to right.

You should only have one bin directory for a JDK in the path at a time. Those following the first instance are ignored. If you are not sure where to add the JDK path, append it. The new path takes effect in each new command window that you open after setting the PATH variable.

Following is a typical value for the PATH variable:

```
C:\WINDOWS\system32;C:\WINDOWS;"C:\Program Files\Java\jdk-20\bin"
```

Note: Windows 7 and Windows 10 have a Start menu; however, the menu is not available in Windows 8 and Windows 8.1. The JDK and Java information in Windows 8 and Windows 8.1 is available in following Start directory: %ALLUSERSPROFILE%\Microsoft\Windows\Start Menu\Programs.

1.8 New Features of Java

Table 1.5 outlines various new features introduced between Java 9 to Java 20.

Java Version	Features
Java 9	Private and Static methods in interfaces
	Try with Resources Improvements
	Immutable Collections
	Optional Class Improvements
	Enhanced @Deprecated annotation
	Stream API Improvements
	JShell
	Create and Use Modules
Java 10	Time-Based Release Versioning
	Local-Variable Type Inference
	Experimental Java-Based JIT Compiler
	Application Class-Data Sharing
	Parallel Full GC for G1
	Garbage-Collector Interface
	Additional Unicode Language-Tag Extensions
	Root Certificates
	Thread-Local Handshakes
	Heap Allocation on Alternative Memory Devices
	Remove the Native-Header Generation Tool – javah
	Consolidate the JDK Forest into a Single Repository
Java 11	Running Java File with single command
	New utility methods in String class
	Local-Variable Syntax for Lambda Parameters
	Nested Based Access Control
	Flight Recorder
	Reading/Writing Strings to and from the Files
Java 12	String API Updates
	Compact Number Formatting
	File mismatch() Method
	Teeing Collectors in Stream API

Java Version	Features
Java 13	Reimplement the Legacy Socket API
	Dynamic CDS Archive
	FileSystems.newFileSystem() Method
	Support for Unicode
	Document Object Model (DOM) and Simple API for XML (SAX) Factories with Namespace
Java 14	Switch Expressions
	Pattern Matching for instanceof
	Helpful NullPointerExceptions
	Records
	Text Blocks
Java 15	Hidden classes
	Z Garbage Collector
	Sealed classes
	Records
	Improved security with Edwards-Curve Digital Signature algorithm
Java 16	Elastic Metaspace
	Packaging Tool
Java 17	Enhanced Pseudo-Random number generators
	Restore Always- Strict-Floating-Point-Semantics
	Strongly encapsulate JDK internals
Java 18	Simple Web server
	Vector API
	Reimplement Core Reflection
	Unicode Transformation Format (UTF)-8 by Default
Java 19	Virtual Threads
	Structured Concurrency
	Foreign Function and Memory API
Java 20	Scoped Values
	Linux/RISC- V Port
	Deprecations and Removals - The method java.lang.System.getSecurityManager(), java.lang.String.getBytes(int, int, byte[], int, and java.lang.String(byte[], int, int, int) has been deprecated. The java.applet package as well as the java.beans.AppletInitializer interface, are no longer available.

Table 1.5: New Features of Java

1.9 Structure of a Java Class

The Java programming language is designed around object-oriented features and begins with a class design. The class represents a template for the objects created or instantiated by JRE. Thus, the development of a Java program starts with a class definition. The definition of the class is written in a file and is saved with a .java extension.

Figure 1.9 shows the basic structure of a Java class.

```
package <package_name>;  
  
import <other_packages>;  
  
public class ClassName {  
    <variables(also known as fields)>;  
  
    <constructor method(s)>;  
  
    <other methods>;  
}
```

Figure 1.9: Structure of a Java Class

As shown in Figure 1.9, the bold letters in the class structure are the Java keywords. The brief description of these Java keywords is as follows:

- **package** – A package defines a namespace that stores classes with similar functionalities in them. The **package** keyword identifies name of the package to which the class belongs. It also identifies visibility of the class within the package and outside the package.

The concept of package is similar to folder in the OS. As folders are created to store related files in the system, similarly packages are used to group similar classes and interfaces in Java. In Java, all classes belongs to a package. If the package statement is not specified, then, the class belongs to the default package.,

For example, all the user interface classes are grouped in the `java.awt` or `java.swing` packages. Similarly, all the classes related to Input/Output functionalities are found in `java.io` or `java.nio` packages.

- **import** – The **import** keyword identifies the classes and packages that are used in a Java class. They help to narrow down the search performed by the Java compiler by informing it about the classes and packages used in the class. In Java, it is mandatory to import the required classes, before they are used in the Java program.

There are some exceptions wherein the use of `import` statement is not required. They are as follows:

- The `import` statement is not required for classes present in the `java.lang` package. This is because `java.lang` package is the default package included in the entire Java program.
 - You do not require to import classes, if they are located in the same package. For example, if the current package is `com.java.company`, any class present in this package can access the other classes without using the `import` statement.
 - Classes that are declared and used along with their package name do not require the `import` statement. For example, `java.text.NumberFormat nf = new java.text.NumberFormat();`.
- **class** – The `class` keyword identifies a Java class. It precedes the name of the class in the declaration. Also, the `public` keyword indicates the access modifier that decides the visibility of the class. The `public` modifier means that the class is visible outside the package. Class name and file name should match.
- Besides these, there are some essential building blocks of a Java program.
- **Variables** – Variables are also referred to as instance fields or instance variables. They represent the state of objects. They are known as instance variables because each instance or object of the class contains its own copy of instance variables.
- **Methods** – Methods are functions that represent some action to be performed on an object. Code is written within methods. They are also referred to as instance methods.
- **Constructors** – Constructors are also methods or functions that are invoked during the creation of an object. They are basically used to initialize the objects.

1.10 Developing a Java Program on Windows Platform

The basic requirements to write a Java program are as follows:

1. JDK 20 installed and configured on the system
2. A text editor

The text editor can be any simple editor included with the platforms. For example, the Windows platform provides a simple text editor named **Notepad**.

To create, compile, and execute a Java program, perform following steps:

- Create a Java program
- Compile `.java` file
- Build and execute Java program

1.10.1 Create a Java Program

Code Snippet 1 demonstrates a simple Java program that displays a message, 'Welcome to the world of Java' to the user. The program is written in a simple text editor, **Notepad** available on the Windows platform.

Code Snippet 1:

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

The step by step explanation of different parts of Java program is as follows:

Step 1: Class Declaration

The syntax to declare a class is as follows:

Syntax:

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

where,

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

The entire class definition and its members must be written within the opening and closing curly braces, { }. In other words, the class declaration is enclosed within a code block. Code blocks are defined within braces. The braces inform the compiler about the beginning and end of a class.

The area between the braces is known as the class body and contains the code for that class.

In Code Snippet 1, class Helloworld { }, HelloWorld is the name of the class.

Step 2: Write the main method

The main() method is the entry point for an application. In other words, all Java applications begin execution by invoking the main() method.

The syntax to write the main() method is as follows:

Syntax:

```
public static void main(String[] args) {  
    // block of statements  
}
```

where,

`public`: Is a keyword that enables the JVM to access the `main()` method.

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

`void`: Is a keyword that represents the data type of the value returned by the `main()` method. It informs the compiler that the method will not return any value.

`args`: Is an array of type `String` and stores command line arguments. `String` is a class in Java and stores group of characters.

Step 3: Write desired functionality

In this step, actionable statements are written within the methods. Code Snippet 1 illustrates the use of a built-in method, `println()` that is used to display a string as an output. The string is passed as a parameter to the method.

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

`System.out.println()` statement displays the string that is passed as an argument. `System` is a predefined class and provides access to the system resources, such as console and `out` is the output stream connected to the console.

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 should be same in most of the cases.

To save the `HelloWorld.java` program, click **File→Save As** in **Notepad** and enter "`HelloWorld.java`" in the **File name** box. The quotation marks are provided to avoid saving the file with extension `HelloWorld.java.txt`.

Figure 1.10 shows the **Save As** dialog box to save the `HelloWorld.java` file.

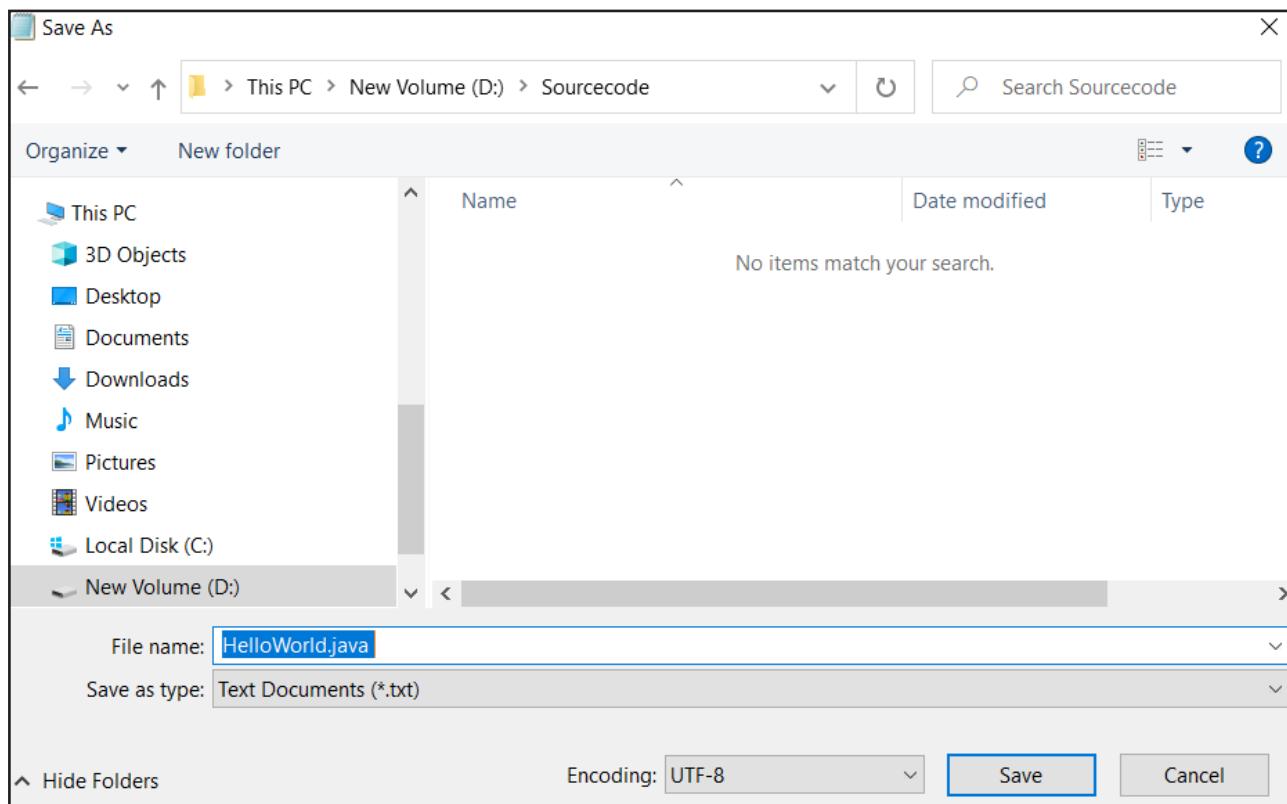


Figure 1.10: Notepad Editor - Save As Dialog Box

Click **Save** to save the file on the system.

1.10.2 Compile .java File

The `HelloWorld.java` file is known as source code file. It is compiled by invoking tool named `javac.exe`, which compiles the source code into a `.class` file. The `.class` file contains the bytecode that is interpreted and converted into machine code before execution. To run the program, the Java interpreter, `java.exe` is required which will interpret and run the program.

Figure 1.11 shows the compilation process of the Java program.

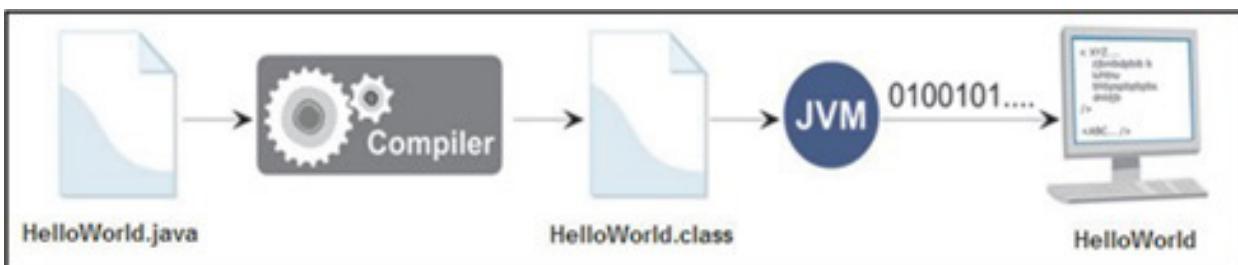
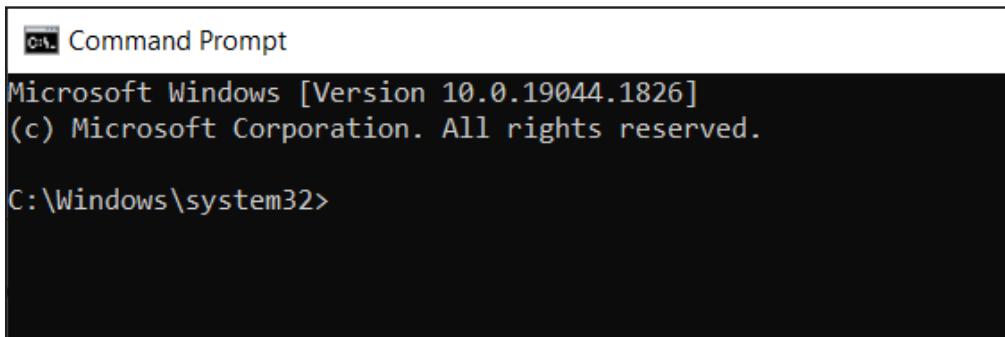


Figure 1.11: Compilation of Java Program

To compile the `HelloWorld.java` program from the Windows platform, the user can click **Start** menu, choose **Run**, and enter the `cmd` command to display the **Command Prompt** window.

Figure 1.12 shows the **Command Prompt** window with the current directory. Normally, the current directory is the home directory of the user.



```
Command Prompt
Microsoft Windows [Version 10.0.19044.1826]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

Figure 1.12: Command Prompt

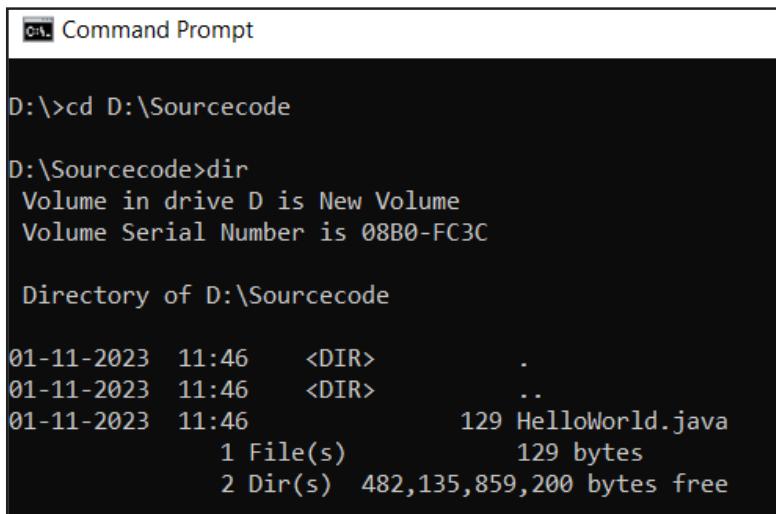
To compile the Java application, developer should change the current directory to the directory in which the `.java` file is located. In this case, the directory is `D:\Sourcecode`.

To change the directory path, perform following steps:

- Type `cd D:\Sourcecode` and press **Enter**.
- Type `dir` command to view the source file, `HelloWorld.java`.

These commands set the drive and directory path to the directory containing `.java` file.

Figure 1.13 shows the commands on the Command Prompt window for changing the directory path to `D:\Sourcecode`.



```
Command Prompt
D:\>cd D:\Sourcecode

D:\Sourcecode>dir
 Volume in drive D is New Volume
 Volume Serial Number is 08B0-FC3C

 Directory of D:\Sourcecode

01-11-2023 11:46    <DIR>      .
01-11-2023 11:46    <DIR>      ..
01-11-2023 11:46            129 HelloWorld.java
                           1 File(s)       129 bytes
                           2 Dir(s) 482,135,859,200 bytes free
```

Figure 1.13: Command Prompt – Change Drive and Directory

As shown in Figure 1.13, the `dir` command displays the `HelloWorld.java` file in the current directory.

Next, the program requires to be compiled using the `javac.exe` command.

The syntax to use the `javac.exe` command is as follows:

Syntax:

`javac [option] source`

where,

`source`: Is one or more file names that end with a `.java` extension.

Type the command, `javac HelloWorld.java` and press **Enter**.

This command will generate a file named `HelloWorld.class` in the current directory. This means that the `HelloWorld` class is the compiled class with the `main()` method in it. This class is assigned to the Java runtime environment, that is, JVM for execution.

Table 1.6 lists some of the options that can be used with the `javac` command.

Option	Description
<code>-classpath</code>	Specifies the location for the imported classes (overrides the CLASSPATH environment variable)
<code>-d</code>	Specifies the destination directory for the generated class files
<code>-g</code>	Prints all debugging information instead of the default line number and file name
<code>-verbose</code>	Generates message while the class is being compiled
<code>-version</code>	Displays version information
<code>sourcepath</code>	Specifies the location of the input source file
<code>-help</code>	Prints a synopsis of standard options

Table 1.6: Options for `javac` Command

For example, `javac -d c:\ HelloWorld.java` will create and save `HelloWorld.class` file in the `C:\` drive.

Note - If the source file defines more than one classes, then, after compilation, each class is compiled into the separate `.class` file. The file that can be given for execution to JVM is the class with the `main()` method defined in it.

1.10.3 Build and Execute Java Program

The JVM is at the heart of the Java programming language and is responsible for executing the `.class` file or bytecode file. The portability feature of the `.class` file has made the principle of Java '**write once, run anywhere**' possible. The `.class` file can be executed on any computer or device, that has the JVM implemented on it.

Figure 1.14 shows the components of JVM involved in the execution of the compiled bytecode.

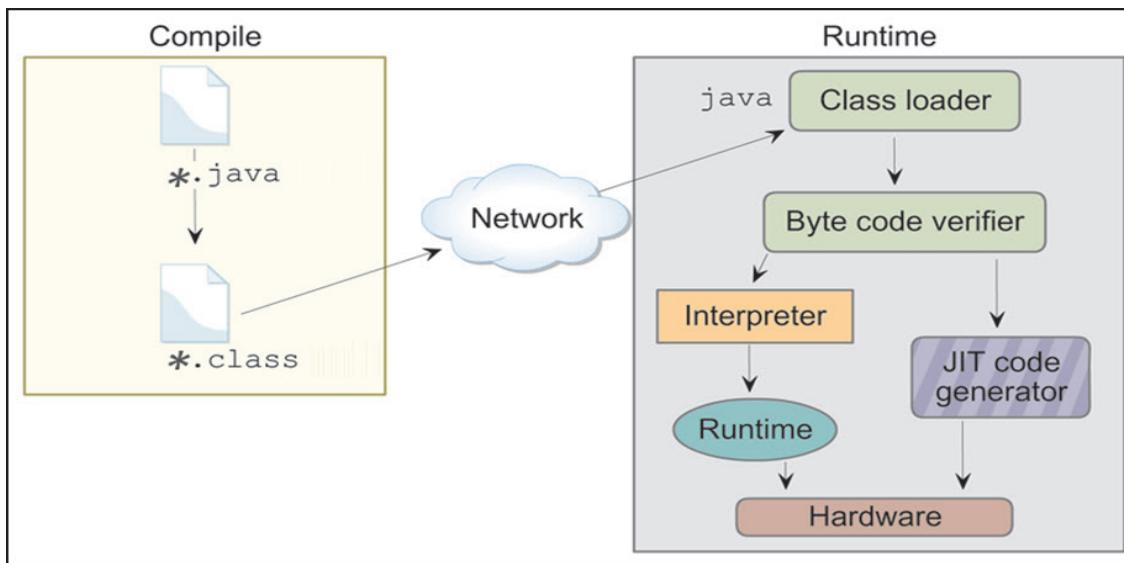


Figure 1.14: Components of JVM

As shown in Figure 1.14, the class loader component of JVM loads all the necessary classes from the runtime libraries required for execution of the compiled bytecode. The bytecode verifier then checks the code to ensure that it adheres to the JVM specification. This ensures that the bytecode does not contain any untrusted code or illegal instructions. Next, the bytecode is executed by the interpreter. To boost the speed of execution, in Java version 2.0, a Hot Spot Just-in-Time (JIT) compiler was included at runtime. During execution, the JIT compiler compiles some of the code into native code or platform-specific code to boost the performance. The Java interpreter command, `java` is used to interpret and run the Java bytecode. It takes the name of a class file as an argument for execution.

The syntax to use the `java.exe` command is as follows:

Syntax:

`java [option] classname [arguments]`

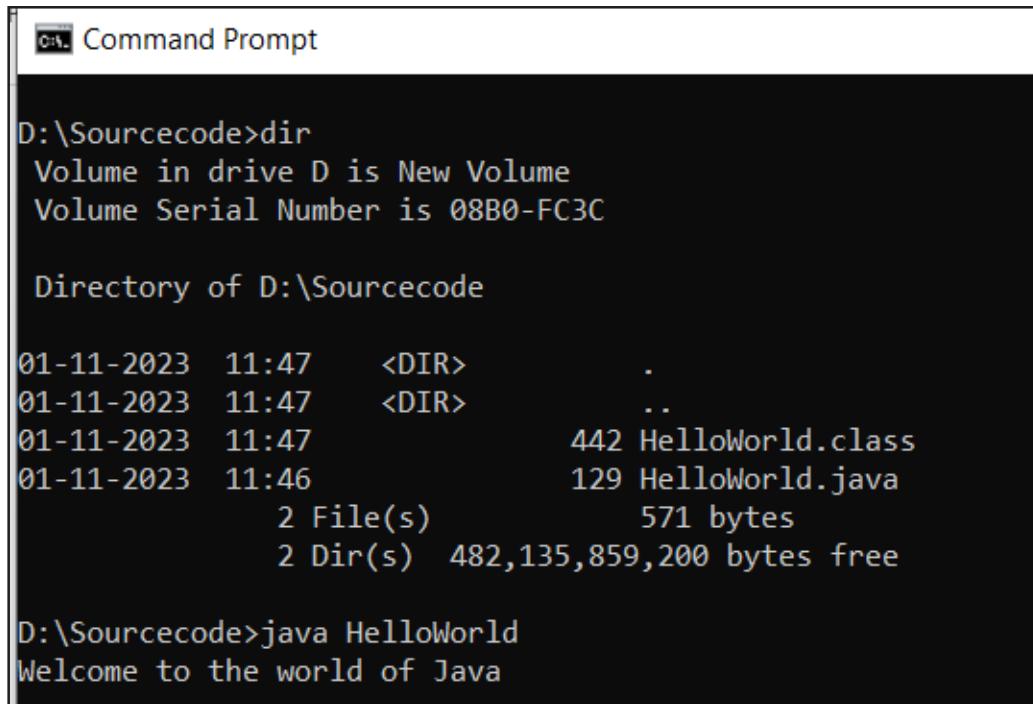
where,

`classname`: Is the name of the class file.

`arguments`: Is the arguments passed to the main function.

To execute the `HelloWorld` class, type the command, `java HelloWorld` and press **Enter**.

Figure 1.15 shows the output of the Java program from Code Snippet 1 on the **Command Prompt** window.



```

Command Prompt

D:\Sourcecode>dir
Volume in drive D is New Volume
Volume Serial Number is 08B0-FC3C

Directory of D:\Sourcecode

01-11-2023 11:47    <DIR>      .
01-11-2023 11:47    <DIR>      ..
01-11-2023 11:47                442 HelloWorld.class
01-11-2023 11:46                129 HelloWorld.java
                           2 File(s)       571 bytes
                           2 Dir(s)  482,135,859,200 bytes free

D:\Sourcecode>java HelloWorld
Welcome to the world of Java

```

Figure 1.15: Output of HelloWorld Program

Table 1.7 lists some of the options that can be used with the `java` command.

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

Table 1.7: Options for `java` Command

1.11 Using NetBeans Integrated Development Environment (IDE)

NetBeans is an open source 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. As compared to an editor, such as **Notepad**, the coding in NetBeans IDE is completed faster because of its features, such as code completions, code template, and fix import. It also supports debugging of an application in the source editor.

Some of the benefits of using NetBeans IDE for Java application development are as follows:

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

1.11.1 Elements of NetBeans IDE

The NetBeans IDE has following elements and views:

- Menu Bar
- Folders View
- Components View
- Coding and Design View
- Output View

NetBeans 17.x is the latest release of the Apache NetBeans IDE, which is a development environment, tooling platform and application framework for Java and other languages. NetBeans 17.x has many improvements and new features, such as:

- Enhanced support for Java SE 19, including records, sealed classes, pattern matching, and text blocks.
- Improved support for Java EE 9, Jakarta EE 9, and MicroProfile 4.0.
- New support for Hypertext Processor (PHP) 8.0, including attributes, union types, match expressions, and named arguments.
- New support for Maven projects, including improved indexing, dependency management, and code generation.
- New support for Web development, including Hypertext Markup Language5 (HTML), Cascading Style Sheet3 (CSS3), JavaScript, TypeScript, and Angular.
- New support for native development, including C/C++, Rust, and Python.
- It enhances Apache Maven tooling, brings multiple enhancements for users of Gradle, and includes built-in features for Payara (an open-source application server) and WildFly. WildFly, formerly known as JBoss, is an application server. WildFly is written in Java and implements Java Platform, EE.

Figure 1.16 shows various elements in the NetBeans IDE 17.0.

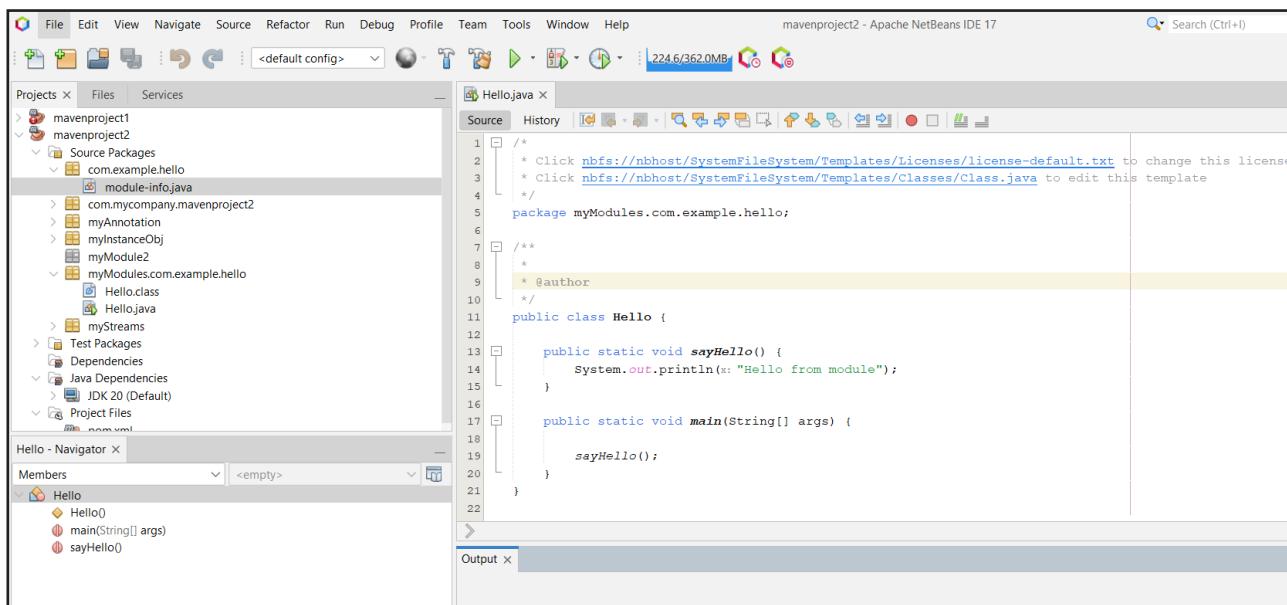


Figure 1.16: Elements in NetBeans IDE

As shown in Figure 1.16, the brief explanation for these elements is as follows:

→ **Menu Bar**

The menu bar of NetBeans IDE contains menus that have several sub-menu items, each providing a unique functionality.

Figure 1.17 shows the menu bar in the NetBeans IDE.

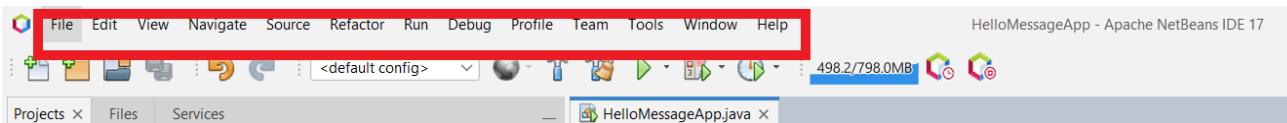


Figure 1.17: Menu Bar in NetBeans IDE

Table 1.8 lists and explains different menus in the NetBeans IDE.

Menu	Description
File	Displays commands for using the project and the NetBeans IDE
Edit	Provides commands for editing
View	Provides options to display the document in different views
Navigate	Provides options for navigation in the project
Source	Provides options for overriding methods, fixing imports, and inserting try-catch blocks
Refactor	Allows to change the structure of the code and reflect changes made to the code

Menu	Description
Run	Compiles source files, executes the application, and generate packaged build output, such as JAR or Web Application Archive (WAR) file
Debug	Inserts breakpoints, watches, and attaches the debugger
Profile	Used for profiling an application with the help of a profiler which examines the time and memory usage of the application
Team	Used for versioning application that helps to keep the backup of the files and tracing the modified source code files. It also allows to work with the shared source files present in the common repository
Tools	Provides tools for creating JUnit test cases, applying internationalization, and creating libraries. Through internationalization, applications can adopt to different languages without changing the source codes
Window	Provides options to select or close any window
Help	Provides guidance on how to use NetBeans IDE effectively

Table 1.8: Menus in NetBeans IDE

→ Folder View

The folder view shows the structure of files associated with Java applications. This view contains Projects window, Files window, and Services window.

Figure 1.18 shows the folder view in the NetBeans IDE.

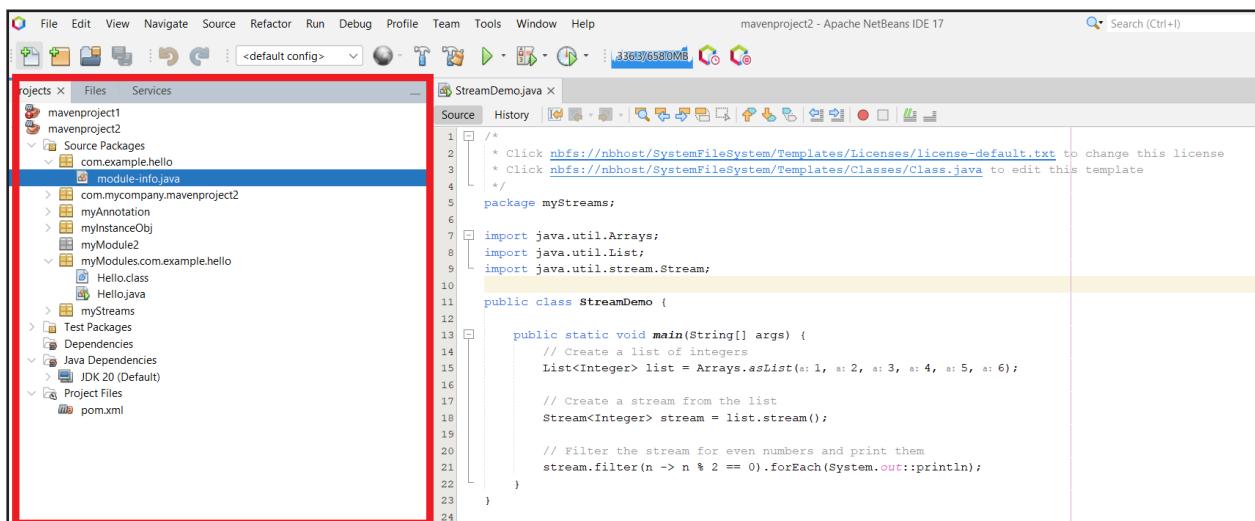


Figure 1.18: Folder View in NetBeans IDE

Table 1.9 lists and describes different elements in the folder view.

Element	Description
Projects Window	Displays the project content, such as Source Packages and Libraries <ul style="list-style-type: none"> Source Packages folder contains the Java source code for the project Libraries comprise resources required by the project, such as source files and Javadoc files
Files Window	Shows the directory structure of all files and folders present in the project
Services Window	Displays information about database drivers, registered servers, and Web services

Table 1.9: Elements in Folder View

→ Component View

The component window is used for viewing components in the NetBeans IDE. The **Navigator** window serves as a tool that displays details of the source files of the currently opened project. Elements used in the source are displayed here in the form of a list or an inheritance tree.

Figure 1.19 shows the **Navigator** window in the NetBeans IDE.

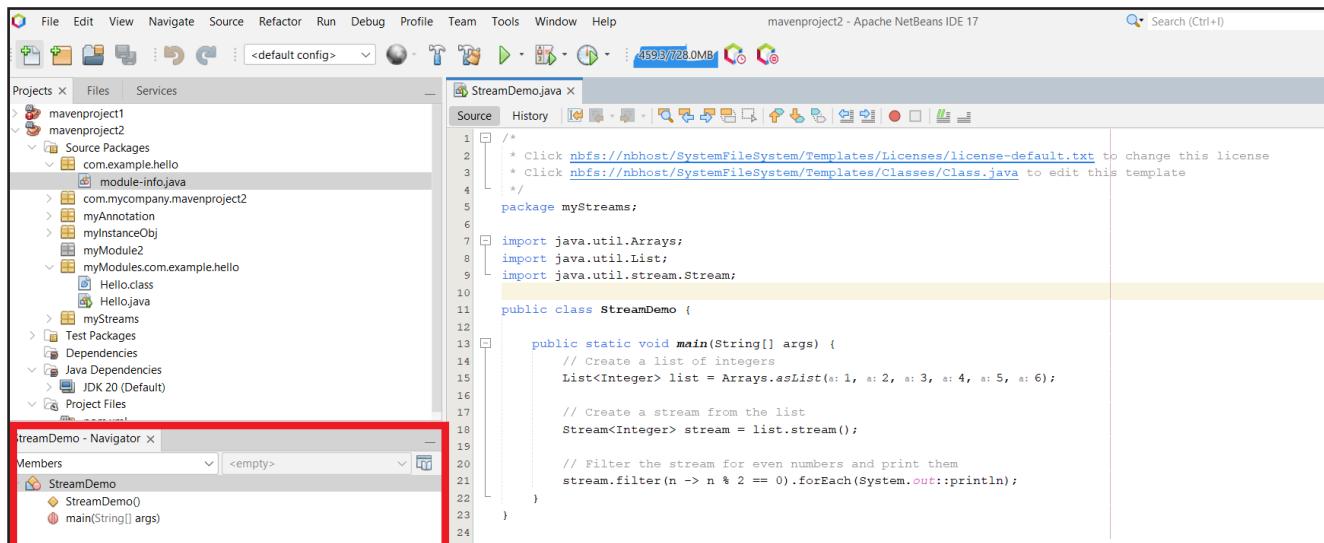
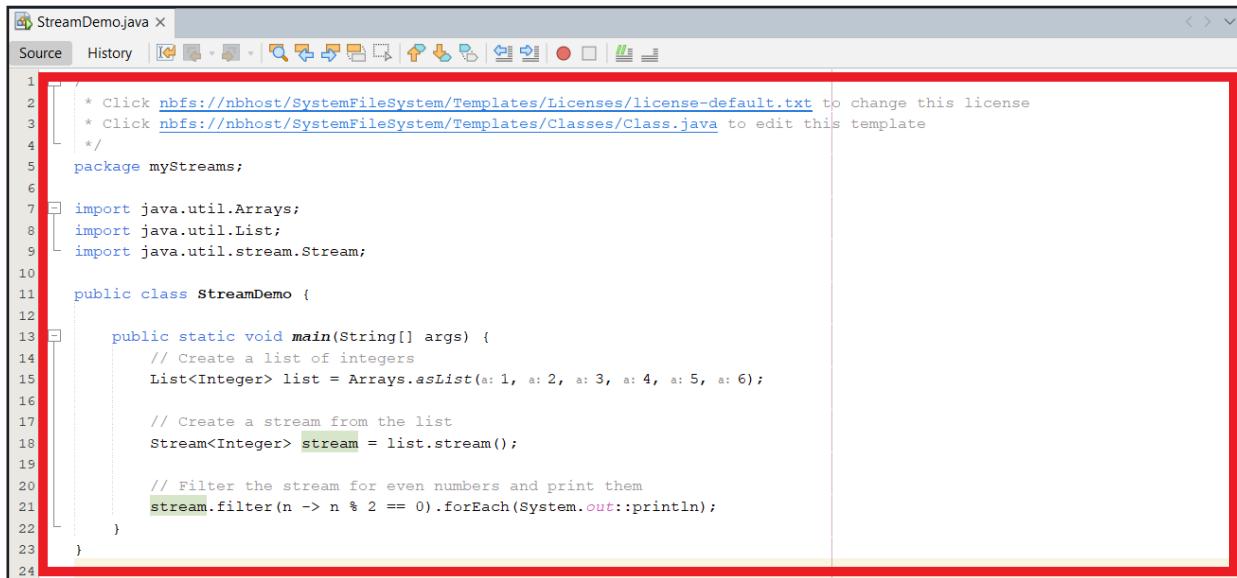


Figure 1.19: Navigator Window

→ Code View

Key element of the coding area in NetBeans IDE is **Source Editor**.

Figure 1.20 shows the **Source Editor** used for viewing the code.



The screenshot shows the NetBeans IDE interface with the title bar "StreamDemo.java X". The menu bar includes "Source", "History", and various tool icons. The code editor displays the following Java code:

```

1  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
3  */
4
5 package myStreams;
6
7 import java.util.Arrays;
8 import java.util.List;
9 import java.util.stream.Stream;
10
11 public class StreamDemo {
12
13     public static void main(String[] args) {
14         // Create a list of integers
15         List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
16
17         // Create a stream from the list
18         Stream<Integer> stream = list.stream();
19
20         // Filter the stream for even numbers and print them
21         stream.filter(n -> n % 2 == 0).forEach(System.out::println);
22     }
23 }

```

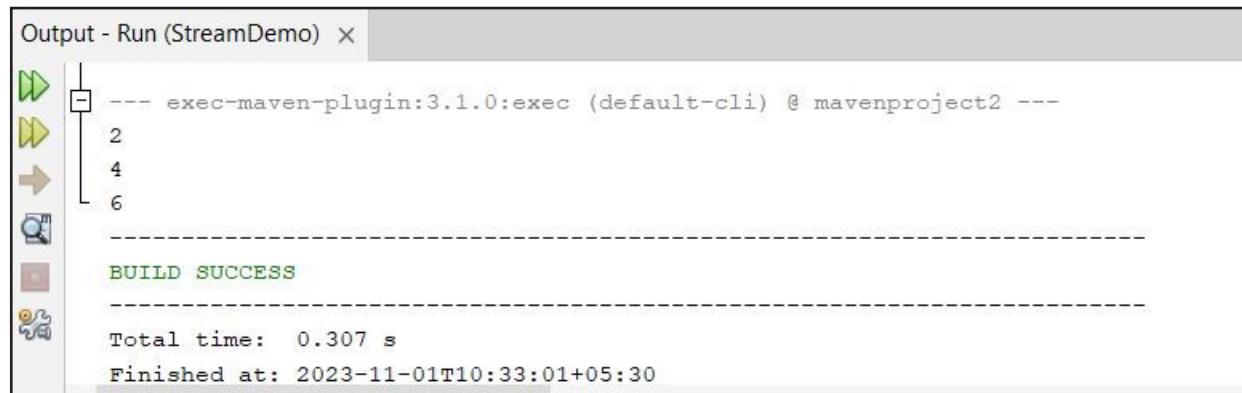
Figure 1.20: Source Editor

Source Editor is a text editor of NetBeans IDE. With Source Editor, user can create, edit, and view source code.

→ Output View

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

Figure 1.21 shows the **Output** window.



The screenshot shows the NetBeans IDE Output window titled "Output - Run (StreamDemo)". The window displays the following output:

```

--- exec-maven-plugin:3.1.0:exec (default-cli) @ mavenproject2 ---
2
4
6
-----
BUILD SUCCESS
-----
Total time: 0.307 s
Finished at: 2023-11-01T10:33:01+05:30

```

Figure 1.21: Output Window

To download NetBeans IDE, visit the link <https://netbeans.apache.org/download/index.html> and perform following steps:

- Step 1: Select **Download** option as shown in Figure 1.22.

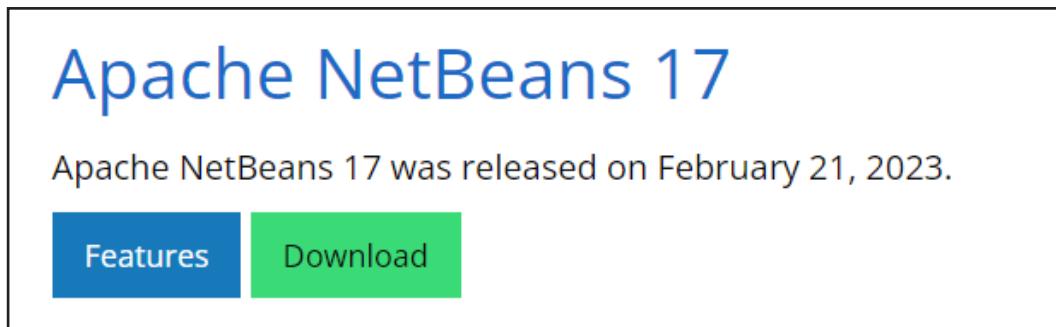


Figure 1.22: Download Option

- Step 2: Select appropriate binary distribution as shown in Figure 1.23.

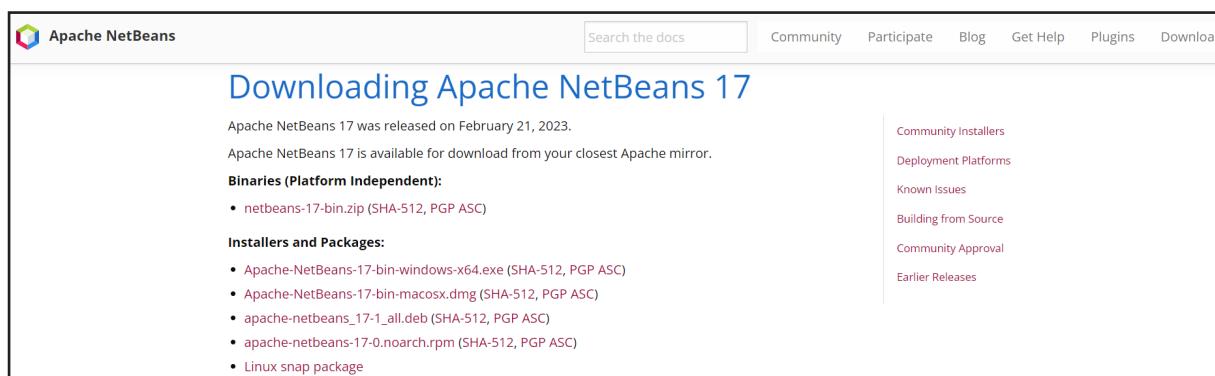


Figure 1.23: Selecting Binary Distribution

- Step 3: Select and install downloaded file by following instructions provided by the installer, one by one. Refer to Figure 1.24 for the downloaded file.

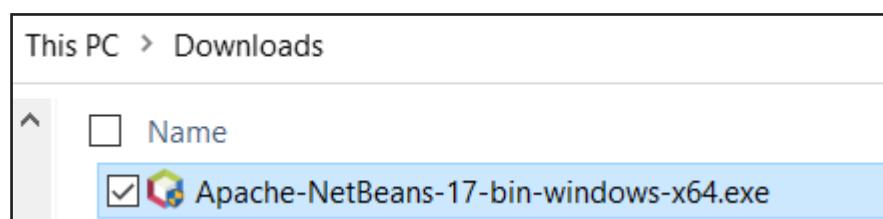


Figure 1.24: Downloaded File

1.11.2 Creating a New Project

To create a project in IDE, perform following steps:

1. To create a new project, click **File → New Project**. This opens the **New Project** wizard.
 2. Under **Categories**, expand **Java with Maven** and then, select **Java Application** under **Projects**.
- Maven is a build automation tool for Java projects.

Figure 1.25 shows the **Choose Project** page in the wizard.

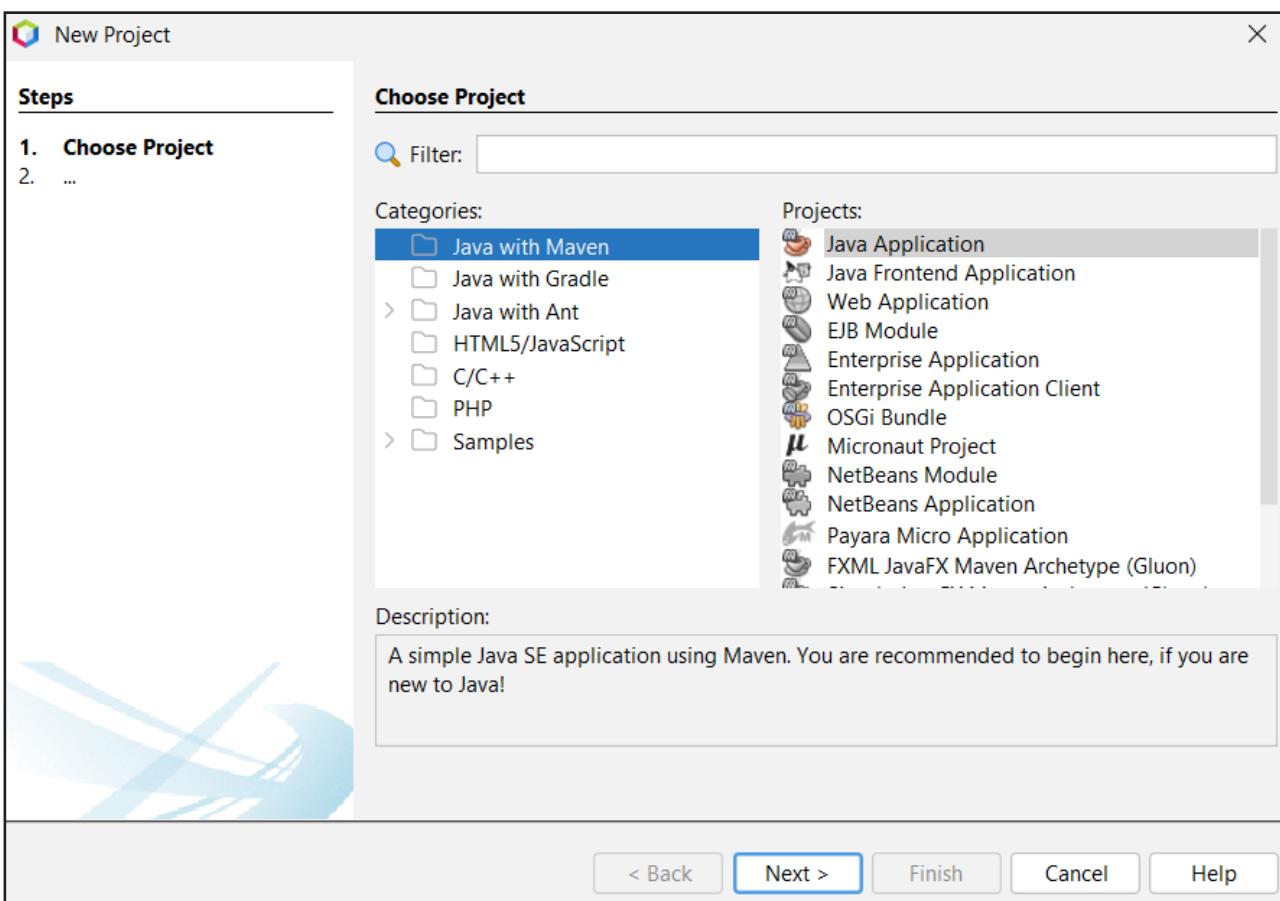


Figure 1.25: New Project Wizard – Choose Project Page

3. Click **Next**. This displays the **Name and Location** page in the wizard.
4. Type **HelloMessageApp** in the **Project Name** box.

5. Click **Browse** and select the appropriate location on the system to set the project location as shown in Figure 1.26.
6. Click **Finish**.

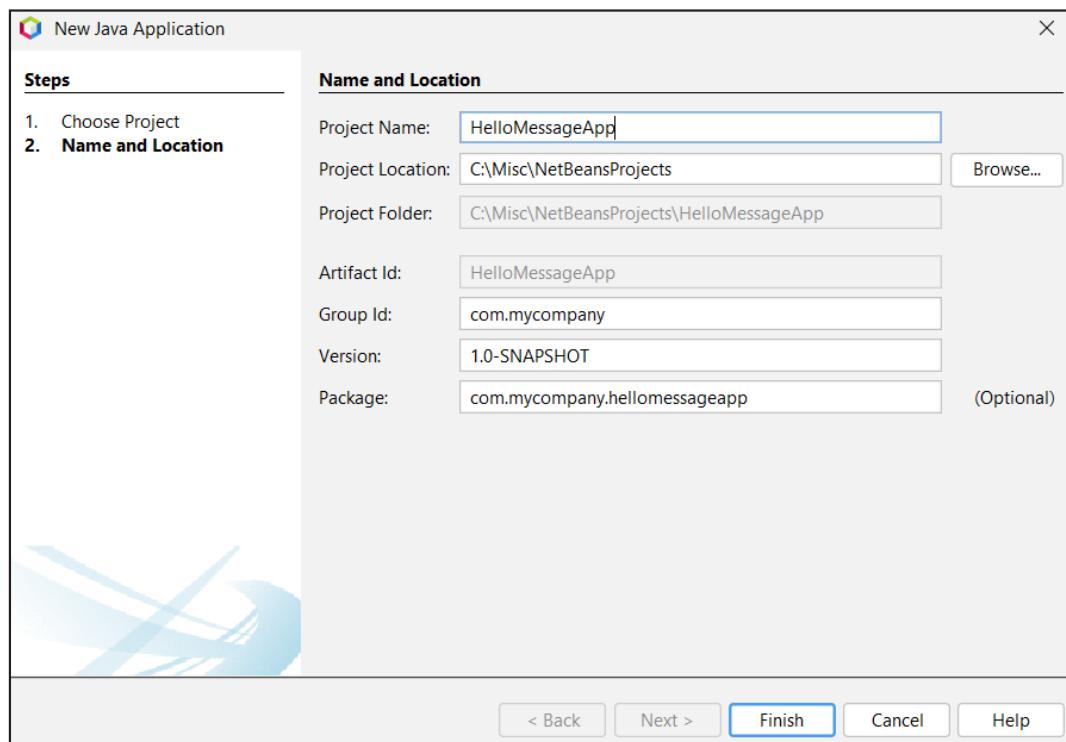


Figure 1.26: New Project Wizard – Name and Location Page

Figure 1.27 shows the project **HelloMessageApp** with class **HelloMessageApp** and auto-generated code.

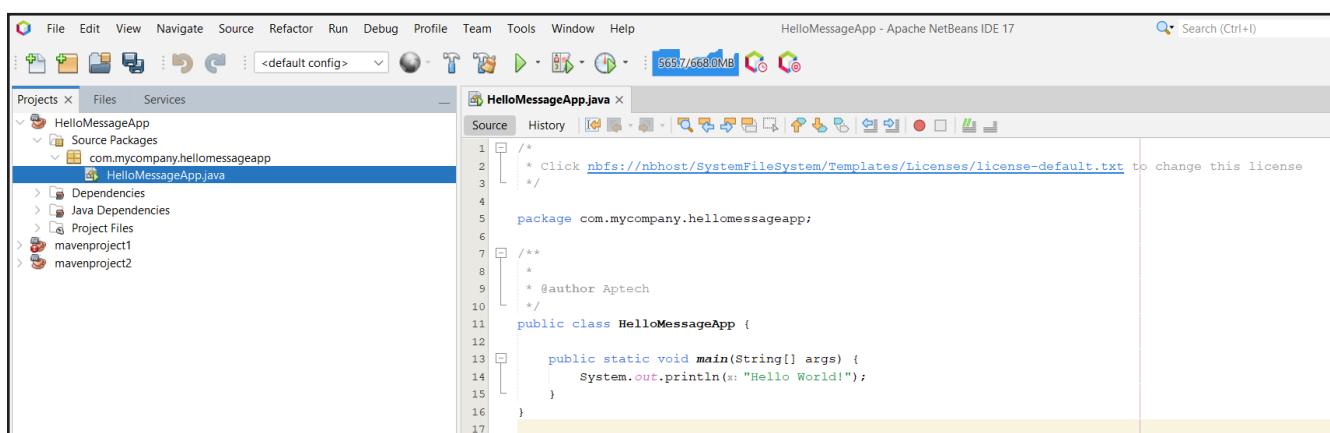
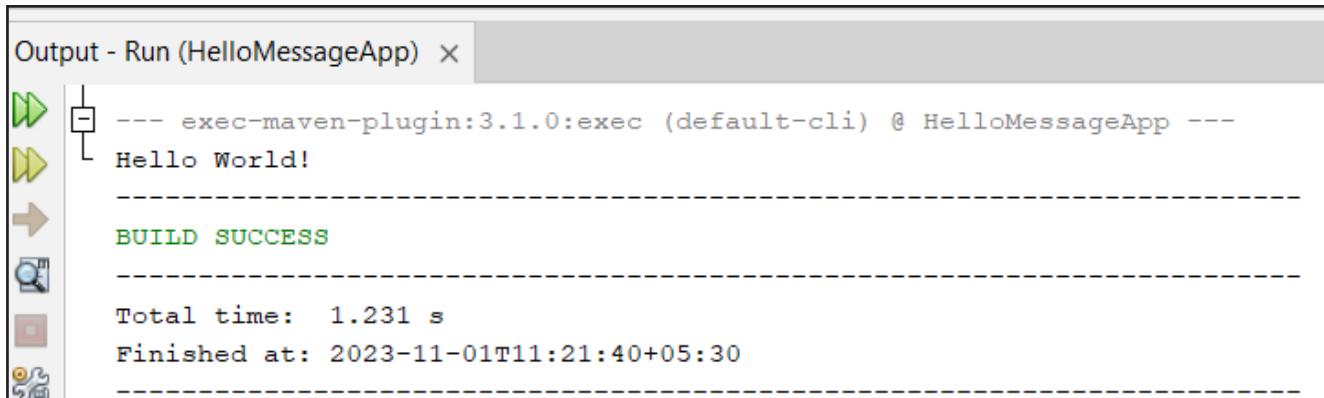


Figure 1.27: NetBeans IDE – HelloMessageApp

Next click **Build** and **Run** the project to see the output as shown in Figure 1.28.



The screenshot shows the 'Output - Run (HelloMessageApp)' window. It displays the following text:

```

--- exec-maven-plugin:3.1.0:exec (default-cli) @ HelloMessageApp ---
Hello World!
-----
BUILD SUCCESS
-----
Total time: 1.231 s
Finished at: 2023-11-01T11:21:40+05:30
-----
```

Figure 1.28: Output Window - HelloMessageApp

1.12 Comments in Java

Comments are placed in a Java program source file. They are used to document the Java program and are not compiled by the compiler. They are added as remarks to make the program more readable for the user. A comment usually describes the operations for better understanding of the code.

There are three styles of comments supported by Java namely, single-line, multi-line, and Javadoc.

1.12.1 Single-line Comments

A single-line comment is used to document the functionality of a single line of code.

Figure 1.29 shows a **HelloWorld** program with single-line comments.

```

class HelloWorld {
    public static void main(String[] args) {
        //The println() method is used to display a message on the screen
        System.out.println("Welcome to the world of Java");
    }
}
```

Figure 1.29: Single-line Comments

There are two ways of using single-line comments that are as follows:

→ **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:

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

The syntax for applying comments is as follows:

Syntax:

```
// Comment text
```

Code Snippet 2 shows different ways of using single-line comments in a Java program.

Code Snippet 2:

```
...
// Declare a variable
int a = 32;
int b // Declare a variable
...
```

1.12.2 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.

Code Snippet 3 shows a Java program that uses multi-line comments.

Code Snippet 3:

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

1.12.3 Javadoc Comments

A Javadoc comment is used to document public or protected classes, attributes, and methods. It starts with `/**` 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.

Code Snippet 4 demonstrates the use of Javadoc comments in the Java program.

Code Snippet 4:

```

/*
 * The program prints the welcome message using the println() method.
 */

package hellomessageapp;

/**
 *
 * @author vincent
 */

public class HelloMessageApp {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // The println() method displays a message on the screen
        System.out.println("Welcome to the world of Java");
    }
}

```

When you execute the `javadoc` command as shown in Figure 1.30, an HTML file will be created showing documentation for the class.

```

C:\Misc\HelloMessage>javadoc HelloMessageApp.java
Loading source file HelloMessageApp.java...
Constructing Javadoc information...
Standard Doclet version 1.8.0_251
Building tree for all the packages and classes...
Generating .\hellomessageapp\HelloMessageApp.html...
Generating .\hellomessageapp\package-frame.html...
Generating .\hellomessageapp\package-summary.html...
Generating .\hellomessageapp\package-tree.html...
Generating .\constant-values.html...
Building index for all the packages and classes...
Generating .\overview-tree.html...
Generating .\index-all.html...
Generating .\deprecated-list.html...
Building index for all classes...
Generating .\allclasses-frame.html...
Generating .\allclasses-noframe.html...
Generating .\index.html...
Generating .\help-doc.html...

C:\Misc\HelloMessage>_

```

Figure 1.30: javadoc Command

1.13 Check Your Progress

1. Which of the following features of Java programming languages allows to execute multiple tasks concurrently?

(A)	Portability	(C)	Garbage Collection
(B)	Multithreading	(D)	Exception Handling

2. Match the following terms against their corresponding description.

Term		Description	
a.	Object	1.	Represents behavior of an object
b.	Class	2.	Represents state of an object
c.	Field	3.	Template or blueprint
d.	Method	4.	Instance of class

(A)	a-1, b-2, c-3, d-4	(C)	a-2, b-1, c-4, d-3
(B)	a-4, b-3, c-2, d-1	(D)	a-3, b-4, c-1, d-2

3. Which of the following statements are true regarding Java platform and its components?

a.	Java platform provides an environment for developing applications that can be executed only on Java hardware and OS
b.	Java API is a large collection of ready-made software components
c.	JVM is a comprehensive set of development tools used for developing applications
d.	There are different implementations of JVM available for different platforms
e.	Java APIs are used to run a Java program

(A)	c and d	(C)	b, c, and d
(B)	a and e	(D)	a, b, and c

4. Which of these statements about compiling and executing a Java program are true?

a.	The <code>javac</code> tool invokes the Java compiler
b.	The <code>java</code> tool invokes the Java interpreter
c.	The Java interpreter checks for the syntax, grammar, and data types of the program
d.	The <code>.class</code> files contain bytecodes
e.	The Java interpreter compiles the code

(A)	c and d	(C)	b, c, and d
(B)	a, b, and d	(D)	c, d, and e

5. _____ defines a namespace that stores classes with similar functionalities in them.

(A)	Package	(C)	Constructor
(B)	Object	(D)	Method

1.13.1 Answers

1.	A
2.	D
3.	B
4.	A
5.	B

```
g package;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

Summary

- The development of application software is performed using a programming language that enforces a particular style of programming, also referred to as programming paradigm.
- In structured programming paradigm, the application development is decomposed into a hierarchy of subprograms.
- In object-oriented programming paradigm, applications are designed around data, rather than focusing only on the functionalities.
- The main building blocks of an OOP language are classes and objects. An object represents a real-world entity and a class is a conceptual model.
- Java is an OOP language as well as a platform used for developing applications that can be executed on different platforms.
- Hidden classes, Z Garbage Collector, Sealed classes, Records, and improved security with Edwards-Curve Digital Signature algorithm are some of the new features in Java 20.
- Apache NetBeans IDE version 17.0 and higher provide an integrated development environment to create, compile, and execute Java programs.
- The components of Java SE platform are JDK and JRE. JRE provides JVM and Java libraries that are used to run a Java program. JDK includes necessary development tools, runtime environment, and APIs for creating Java programs.

```
import java.util.*;  
public class Main {  
    public static void main(  
        String args[]) {  
        System.out.println("Hello World");  
    }  
}
```

Try It Yourself

1. Check the directory structure of JDK 20 on your local system. Also, identify and list various tools provided by JDK 20. Prepare a Table displaying the list of tools with their appropriate description and commands to invoke the tools.
2. Find all the possible errors that might be generated due to incorrect use of PATH and CLASSPATH variables. Also, identify the appropriate steps that can be taken to fix those errors. Note these errors and solutions in a **Notepad** file.

Session - 2

Variables, Data Types, and Operators

Welcome to the Session, **Variables, Data Types, and Operators**.

This session focuses on the usage of variables, literals, data types, and operators in Java programs. It provides a clear understanding of different data types available in Java. Further, the session explains different types of operators and their implementations in Java. Finally, it explains implicit and explicit conversion techniques.

In this Session, you will learn to:

- Explain variables and their purpose
- Explain the syntax of variable declaration
- Explain the rules and conventions for naming variables
- Explain data types
- Explain primitive and reference data types
- Explain escape sequence
- Explain format specifiers
- Identify and explain different type of operators
- Explain the concept of casting
- Explain implicit and explicit conversion



2.1 Introduction

The core of any programming language is the way it stores and manipulates the data. The Java programming language can work with different types of data, such as number, character, boolean, and so on. To work with these types of data, Java programming language supports the concept of variables. A variable is like a container in the memory that holds the data used by the Java program. It is an identifier whose value can be changed.

A variable is associated with a data type that defines the type of data that will be stored in the variable.

Java is a strongly-typed language which means that any variable or an object created from a class must belong to its type and should store the same type of data. The compiler checks all expressions variables and parameters to ensure that they are compatible with their data types. In case, if any error or mismatch is found, then they must be corrected during compile time. This reduces the runtime errors that occur in other languages due to data type mismatch.

2.2 Variables

A variable is a location in the computer's memory which stores the data that is used in a Java program.

Figure 2.1 depicts a variable that acts as a container and holds the data in it.

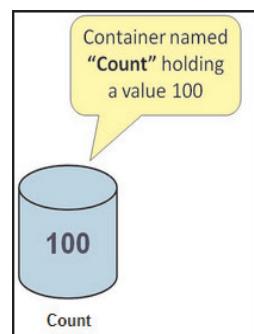


Figure 2.1: Variable

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. They must be declared before they can be used in the program.

A variable declaration begins with data type and is followed by variable name and a semicolon. The data type can be a primitive data type or a class.

The syntax to declare a variable in a Java program is as follows:

Syntax:

```
datatype variableName;
```

where,

datatype: Is a valid data type in Java.

variableName: Is a valid variable name.

Code Snippet 1 demonstrates how to declare variables in a Java program.

Code Snippet 1:

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

In the code, the statements declare an integer variable named `rollNumber` and a character variable called `gender`. These statements instruct the Java runtime environment to allocate the required amount of memory for each of the variable. These variables will hold the type of data specified for each of them.

Additionally, each statement also provides a name that can be used within the program to access the values stored in each variable.

2.2.1 Rules for Naming Variables

Java programming language provides set of rules and conventions that must be followed for naming variables. For example, variable names should be short and meaningful. The use of naming conventions ensures good programming practices and results in less syntax errors.

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`).

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

Variable Name	Valid/Invalid
<code>rollNumber</code>	Valid
<code>a2x5_w7t3</code>	Valid
<code>\$yearly_salary</code>	Valid
<code>_2010_tax</code>	Valid

Variable Name	Valid/Invalid
\$_\$	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 Variable Names

2.2.2 Assigning Value to a Variable

Values can be assigned to variables by using the assignment operator (=). There are two ways to assign value to variables. These are as follows:

→ **At the time of declaring a variable**

Code Snippet 2 demonstrates the initialization of variables at the time of declaration.

Code Snippet 2:

```
...
int rollNumber = 101;
char gender = 'M';
...
```

In the code, variable **rollNumber** is an integer variable, so it has been initialized with a numeric value **101**. Similarly, variable **gender** is a character variable and is initialized with a character '**M**'. The values assigned to the variables are called as literals. Literals are constant values assigned to variables directly in the code without any computation.

→ **After the variable declaration**

Code Snippet 3 demonstrates the initialization of variables after they are declared.

Code Snippet 3:

```
int rollNumber; // Variable is declared
...
rollNumber = 101; //variable is initialized
...
```

Here, the variable **rollNumber** is declared first and then, according to the requirement of the variable in the code, it has been initialized with the numeric literal **101**.

Code Snippet 4 shows different ways of declaring and initializing variables in Java.

Code Snippet 4:

```
// Declares three integer variables x, y, and z
int x, y, z;

// Declares three integer variables, initializes a and c
int a = 5, b, c = 10;

// Declares a byte variable num and initializes its value to 20
byte num = 20;

// Declares the character variable n with value 'c'
char n = 'c';

// Stores value 10 in num1 and num2
int num2;

int num1 = num2 = 10; //
```

In the code, the declarations, `int x, y, z;` and `int a=5, b, c=10;` are examples of comma separated list of variables. The declaration `int num1 = num2 = 10;` assigns same value to more than one variable at the time of declaration.

Note - In Java, a variable must be declared before it can be used in the program. Thus, Java can be referred to as strongly-typed programming language.

2.2.3 Different Types of Variables

Java programming language allows you to define different kind of variables. These variables are categorized as follows:

- Instance variables
- Static variables
- Local variables

They are described as follows:

- **Instance variables** – The state of an object is represented as fields or attributes or instance variables in the class definition. Each object created from a class contains its own individual instance variables. In other words, each object will have its own copy of instance variables.

Figure 2.2 shows the instance variables declared in a class template. All objects from the class contain their own instance variables which are non-static fields.

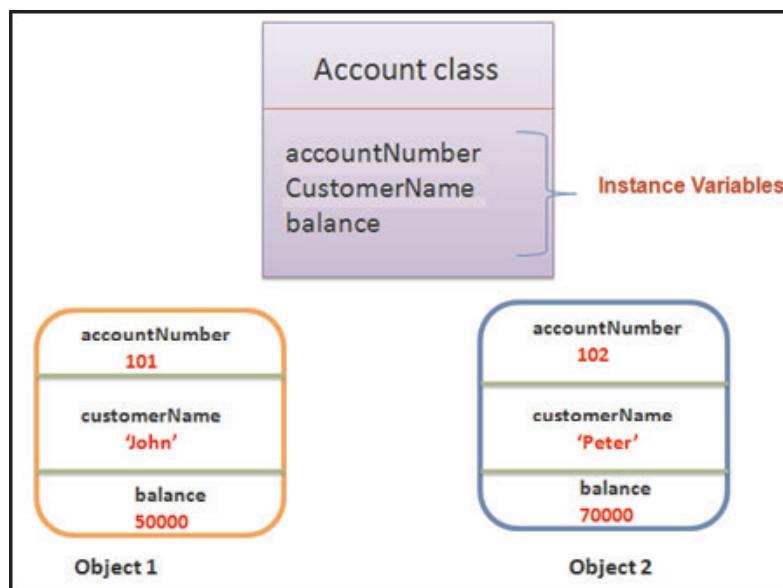


Figure 2.2: Instance Variables

- **Static variables** – These are also known as class variables. Only one copy of static variable is maintained in the memory that is shared by all the objects belonging to that class. These fields are declared using the `static` keyword and inform the compiler that only one copy of this variable exists irrespective of number of times the class has been instantiated.

Figure 2.3 shows static variables in a Java program.

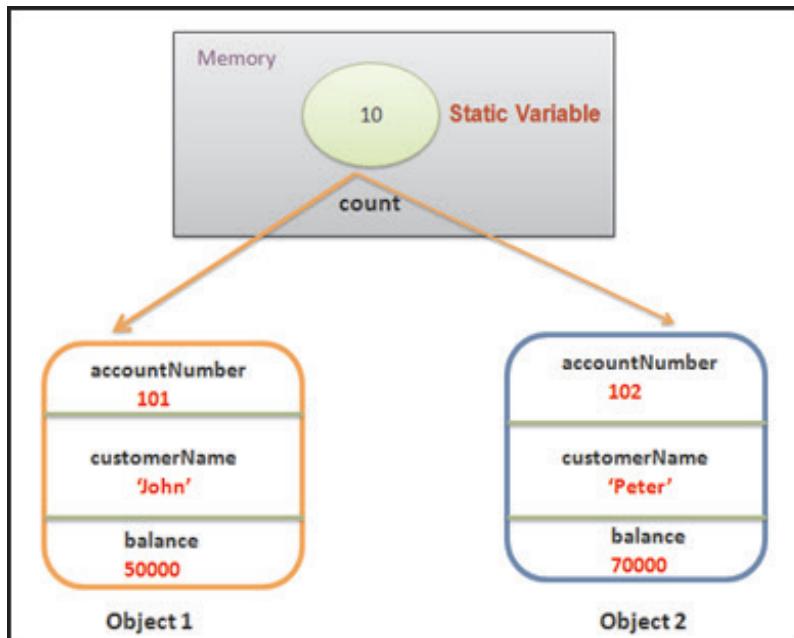


Figure 2.3: Static Variables

- **Local variables** – The variables declared within the blocks or methods of a class are called local variables. A method represents the behavior of an object. Local variables are visible within those methods and are not accessible outside them. A method stores its temporary state in local variables. There is no special keyword available for declaring a local variable, hence, the position of declaration of the variable makes it local.

Note - A block begins with an opening curly brace ({) and ends with a closing curly brace (}).

2.2.4 Scope of Variables

In Java, variables can be declared within a class, method, or within any block. Boundaries of the block, that is, opening and closing curly braces define the scope of variables in Java. A scope determines the visibility of variables to other part of the program. Everytime a block is defined, it creates a new scope. Similarly, the lifetime of a variable defines the time period for which the variable exists in a program.

Other Languages, such as C and C++ defines the visibility or scope of an variable in two categories. These are global and local. In Java, the two major scopes of a variable are defined either within a class or within a method.

The variables declared within the class can be instance variables or static variables. The instance variables are owned by the objects of the class. Thus, their existence or scope depends upon the object creation. Similarly, static variables are shared between the objects and exists for the lifetime of a class.

2.2.5 Local Variable Type Inference

The variables defined within the methods of a class are local variables. The lifetime of these variables depends on the execution of methods. This means memory is allocated for the variables when the method is invoked and destroyed when the method returns. After the variables are destroyed, they are no longer in existence.

Methods also have parameters. Parameters are the variables declared with the method. They hold values passed to them during method invocation. The parameter variables are also treated as local variables which means their existence is till the method execution is completed.

Figure 2.4 shows the scope and lifetime of variables x and y defined within the Java program.

```
public class ScopeOfVariables {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Known to code within main() method
        int x; ← Variable x is accessible within the
        x = 10;

        { // Starts a block with new scope
            int y = 20; ← Variable y is visible only within the
            System.out.println("x and y: " + x + " " + y);

            // Calculates value for variable x
            x = y * 2;
        } // End of the block

        // y = 100; // Error! y not known here

        // x is accessible
        System.out.println("x is: " + x);
    }
}
```

Figure 2.4: Scope of Variables

As shown in Figure 2.4, variables declared in the outer block are visible to the inner blocks. Here, the outer block is the `main()` method and the block defined within the `main()` method is the inner block. Thus, variable `x` is visible throughout the method, whereas variable `y` is visible only within the inner block.

2.3 Data Types

When you define a variable in Java, you must inform 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 a variable.

Thus, the data types determine the type of data that can be stored in variables and the operation that can be performed on them.

In Java, data types fall under two categories that are as follows:

- Primitive data types
- Reference data types

2.3.1 Primitive Data Types

The Java programming language provides eight primitive data types to store data in Java programs. A primitive data type, also called built-in data types, stores a single value at a time, such as a number or a character. The size of each data type will be same on all machines while executing a Java program.

The primitive data types are predefined in the Java language and are identified as reserved words.

Figure 2.5 shows the primitive data types that are broadly grouped into four groups.

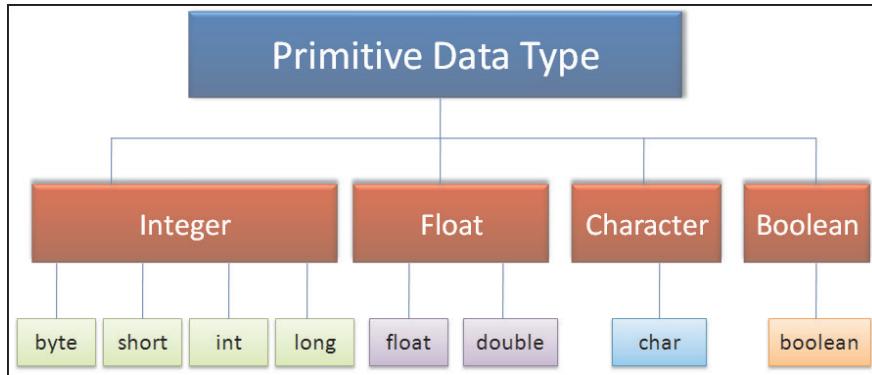


Figure 2.5: Primitive Data Types

Table 2.2 shows the list of primitive data types.

Data Type	Description	Example	Size	Range	Default Value
boolean	This data type represents true and false values	boolean b = true;	1 bit	true or false	false
byte	An 8-bit signed two's complement integer	byte a = 100;	1 byte	-128 to 127 (inclusive)	0
char	A single 16-bit Unicode character	char c = 'A';	2 bytes	'\u0000' to '\uffff'	'\u0000'
short	A 16-bit signed two's complement integer	short s = 200;	2 bytes	-32,768 to 32,767 (inclusive)	0
int	A 32-bit signed two's complement integer	int x = 10;	4 bytes	-2,147,483,648 to 2,147,483,647 (inclusive)	0
long	A 64-bit signed two's complement integer	long y = 100L;	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (inclusive)	0L
float	A single-precision 32-bit Institute of Electrical and Electronics Engineers (IEEE) 754 floating point number	float f = 3.14f;	4 bytes	Approximately ±3.40282347E+38F (6-7 significant decimal digits)	0.0f

Data Type	Description	Example	Size	Range	Default Value
double	A double-precision 64-bit IEEE 754 floating point number	double d = 2.71828;	8 bytes	Approximately ±1.79769313486231570E+308 (15 significant decimal digits)	0.0d

Table 2.2: List of Primitive Data Types

Apart from primitive data types, Java programming language also supports strings. A string is a sequence of characters. Java does not provide any primitive data type for storing strings, instead provides a class `String` to create string variables. The `String` class is defined within the `java.lang` package in Java SE API.

Code Snippet 5 demonstrates the use of `String` class as primitive data type.

Code Snippet 5:

```
...
String str = "A String Data";
...
```

The statement, `String str` creates a `String` object and is not of a primitive data type. When you enclose a string value within double quotes, the Java runtime environment automatically creates an object of `String` type. Also, once the `String` variable is created with a value '`A String Data`', it will remain constant and you cannot change the value of the variable within the program. However, initializing string variable with new value creates a new `String` object. This behavior of strings makes them as immutable objects.

Code Snippet 6 demonstrates the use of different data types in Java.

Code Snippet 6:

```
public class EmployeeData {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declares a variable of type integer
        int empNumber;
        // Declares a variable of type decimal
        float salary;
        // Declare and initialize a decimal variable
        double shareBalance = 456790.897;
        // Declare a variable of type character
    }
}
```

```

char gender = 'M';
// Declare and initialize a variable of type boolean
boolean ownVehicle = false;
// Variables, empNumber and salary are initialized
empNumber = 101;
salary = 6789.50f;
// Prints the value of the variables on the console
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);
}
}

```

Here, variables of type int, float, char, double, and boolean are declared. A float value requires to have the letter f appended at its end. Otherwise, by default, all the decimal values are treated as double in Java. Values are assigned to each of these variables and are displayed using the System.out.println() method.

The output of the code is shown in Figure 2.6.

```

run:
Employee Number: 101
Salary: 6789.5
Gender: M
Share Balance: 456790.897
Owns vehicle: false
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 2.6: Output of Different Data Types

2.3.2 Reference Data Types

In Java, objects and arrays are referred to as reference variables. When an object or an array is created, a certain amount of memory is assigned to it and the address of this memory block is stored in the reference variable. In other words, reference data type is an address of an object or an array created in memory.

Figure 2.7 shows the reference data types supported in Java.

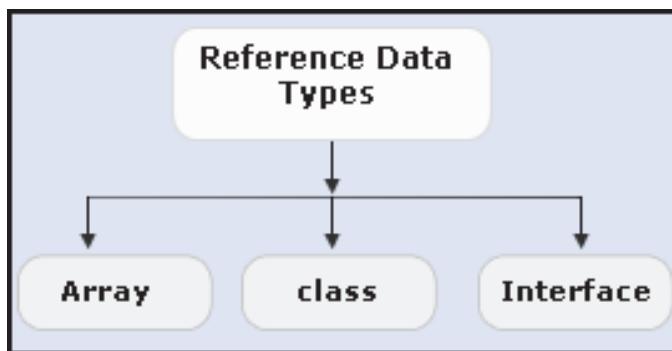


Figure 2.7: Reference Data Types

Table 2.3 lists and describes the three reference data types.

Data Type	Description
Array	It is a collection of several items of the same data type. For example, names of students in a class can be stored in an array
Class	It is encapsulation of instance variables and instance methods
Interface	It is a type of class in Java used to implement inheritance

Table 2.3: Reference Data Types

2.4 Literals

A literal represents a fixed value assigned to a variable. It is represented directly in the code and does not require computation.

Figure 2.8 shows some literals for primitive data types.

Integer	Float	Character	Boolean
50	35.7F	'C'	true

Figure 2.8: Literals

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 considered as an integer literal.

Integers literals can be expressed as:

- Decimal values have a base of 10 and consist of numbers from 0 through 9. For example,
`int decNum = 56;`

- Hexadecimal values have a base of 16 and consist of numbers 0 through 9 and letters A through F. For example, `int hexNum = 0X1c;`
- Binary values have a base of 2 0 and 1. Java supports binary literals. For example, `int binNum = 0b0010;`
- 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 `boolean` variables or used in expressions with `boolean` operators.

→ Character Literals

Character literals are enclosed in single quotes. All the visible American Standard Code for Information Interchange (ASCII) characters can be directly enclosed within quotes, such as '`g`', '`$`', and '`z`'. Single characters that cannot be enclosed within single quotes are used with escape sequence.

→ Null Literals

When an object is created, a certain amount of memory is allocated for that object. The starting address of the allocated memory is stored in an object variable, 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`. For example, `Car toyota = null;`

→ String Literals

String literals consist of sequence of characters enclosed in double quotes. For example, "`Welcome to Java`", "`Hello\nWorld`".

2.4.1 Underscore Character in Numeric Literals

Java allows you to add underscore characters (_) between the digits of a numeric literal. The underscore character can be used only between the digits.

In integral literals, underscore characters can be provided for telephone numbers, identification numbers, or part numbers, and so on. Similarly, for floating-point literals, underscores are used between large decimal values.

The use of underscore character in literals improves the readability of a Java program.

Restrictions for using underscores in numeric literals are as follows:

- A number cannot begin or end with an underscore.
- In the floating-point literal, underscore cannot be placed adjacent to a decimal point.
- Underscore cannot be placed before a suffix, L or F.
- Underscore cannot be placed before or after the binary or hexadecimal identifiers, such as b or x.

Table 2.4 shows valid and invalid placement of underscore character.

Numeric Literal	Valid/Invalid
1234_9876_5012_5454L	Valid
_8976	Invalid, as underscore placed at the beginning
3.14_15F	Valid
0b11010000_11110000_00001111	Valid
3_.14_15F	Invalid, as underscore is adjacent to a decimal point
0x_78	Invalid, an underscore is placed after the hexadecimal

Table 2.4: Placement of Underscore Character

2.5 Escape Sequences

The escape sequences can be used for character and string literals. 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 notation in a Java program (\ 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 2.5 lists escape sequence characters in Java.

Escape Sequence	Character Value
\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
\uxxxx	Unicode character with encoding xxxx, where xxxx is one to four hexadecimal digits. Unicode escapes are distinct from the other escape types

Table 2.5: Escape Sequences

Code Snippet 7 demonstrates the use of escape sequence characters.

Code Snippet 7:

```
public class EscapeSequence {
    /*
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Uses tab and new line escape sequences
        System.out.println("Java \t Programming \n Language");
        // Prints Tom "Dick" Harry string
        System.out.println("Tom \"Dick\" Harry");
    }
}
```

The output of the code is shown in Figure 2.9.

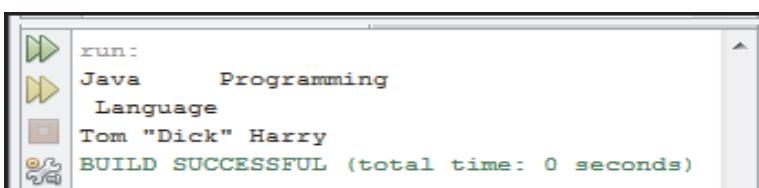


Figure 2.9: Output of Escape Sequences

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

Code Snippet 8 demonstrates the Unicode characters in a Java program.

Code Snippet 8:

```
public class UnicodeSequence {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Prints 'Hello' using hexadecimal escape sequence characters
        System.out.println("\u0048\u0065\u006C\u006C\u006F" + "!\\n");
        // Prints 'Blake' using octal escape sequence character for 'a'
        System.out.println("Bl\\141ke\"2007\" ");
    }
}
```

The output of the code is shown in Figure 2.10.

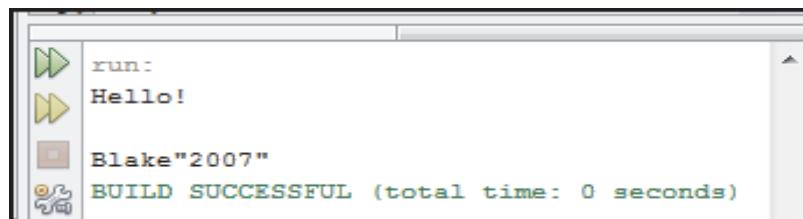


Figure 2.10: Output of Unicode Sequence

The two types of escape sequences can have different semantics because the Unicode, \u escape sequences are processed, before the other escape sequences.

Note - The hexadecimal escape sequence starts with \u followed by four hexadecimal digits. The octal escape sequence comprises three digits after back slash. For example,

\xxyy

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

2.6 Constants and Enumerations

Consider a code that calculates the area of a circle. To calculate the area of a circle, the value of PI and radius must be provided in the formula. The value of PI is a constant value. This value will remain unchanged irrespective of the value provided to the radius.

Similarly, constants in Java are fixed values assigned to identifiers that are not modified throughout the execution of the code. They are defined when you want to preserve values to reuse them later or to prevent any modification to the values. In Java, the declaration of constant variables is prefixed with the final keyword. Once the constant variable is initialized with a value, any attempt to change the value

within the program will generate a compilation error.

The syntax to initialize a constant variable is as follows:

Syntax:

```
final data-type variable-name = value;
```

where,

final: Is a keyword and denotes that the variable is declared as a constant.

Code Snippet 9 demonstrates the code that declares constant variables.

Code Snippet 9:

```
public class AreaOfCircle {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declares constant variable
        final double PI = 3.14159;
        double radius = 5.87;
        double area;
        // Calculates the value for the area variable
        area = PI * radius * radius;
        System.out.println("Area of the circle is: " + area);
    }
}
```

In the code, a constant variable **PI** is assigned the value 3.14159, which is a fixed value. The variable **radius** stores the radius of the circle. The code calculates area of the circle and displays the output.

The output of the code is shown in Figure 2.11.

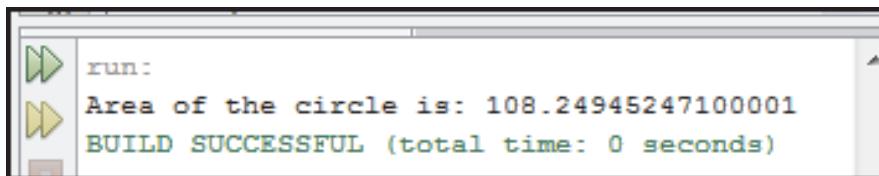


Figure 2.11: Output of Constant Variable

Note - The **final** keyword can also be applied at method or class level. When applied to a method, then it cannot be overridden. When applied to a class declaration, then that class cannot be extended.

An enumeration is defined as a list that contains constants. In previous languages, such as C++, enumeration was a list of named integer constants, but in Java, enumeration is a class type. This means

it can contain instance variables, methods, and constructors. As a result, the concept of enumeration has been expanded in Java. The enumeration is created using the `enum` keyword.

The syntax to create an enumeration is as follows:

Syntax:

```
enum enum-name {
    constant1, constant2, . . . , constantN
}
```

Though, enumeration is a class in Java, you do not use `new` operator to instantiate it. Instead, declare a variable of type enumeration to use it in the Java program. This is similar to using primitive data types. The enumeration is mostly used with decision-making constructs, such as `switch-case` statement.

Code Snippet 10 demonstrates the declaration of enumeration in a Java program.

Code Snippet 10:

```
public class EnumDirection {
    /**
     * Declares an enumeration
     */
    enum Direction {
        East, West, North, South
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Declares a variable of type Direction
        Direction direction;
        // Instantiate the enum Direction
        direction = Direction.East;
        // Prints the value of enum
        System.out.println("Value: " + direction);
    }
}
```

The output of the code is shown in Figure 2.12.

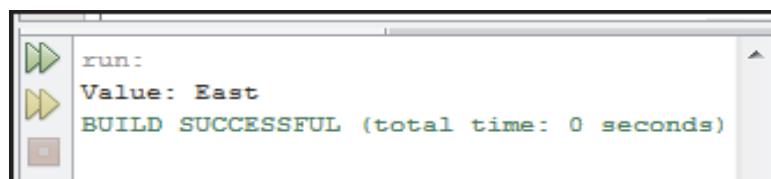


Figure 2.12: Output of Enumeration Variable

2.7 Formatted Output and Input

Whenever an output is to be displayed on the screen, it requires to be formatted. Formatting can be done using three ways that are as follows:

- `print()` and `println()`
- `printf()`
- `format()`

These methods behave in a similar manner. The `format()` method uses the `java.util.Formatter` class to do the formatting work.

2.7.1 `print()` and `println()` Methods

These methods convert all the data to strings and display it as a single value. Methods use appropriate `toString()` method for conversion of the values. These methods can also be used to print mixture combination of strings and numeric values as strings on the standard output.

Code Snippet 11 demonstrates the use of `print()` and `println()` methods.

Code Snippet 11:

```
public class DisplaySum {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int num1 = 5;
        int num2 = 10;
        int sum = num1 + num2;
        System.out.print("The sum of ");
        System.out.print(num1);
        System.out.print(" and ");
        System.out.print(num2);
        System.out.print(" is ");
        System.out.print(sum);
        System.out.println(".");
        int num3 = 2;
        sum = num1 + num2 + num3;
        System.out.println("The sum of " + num1 + ", " + num2 + " and " +
        num3 + " is " + sum + ".");
    }
}
```

The `sum` variable is formatted twice. In the first case, the `print()` method is used for each instruction which prints the result on the same line. In the second case, the `println()` method is used to convert each data type to string and concatenate them to display as a single result.

The output of the code is shown in Figure 2.13.

```
run:
The sum of 5 and 10 is 15.
The sum of 5, 10 and 2 is 17.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 2.13: Output of `print()` and `println()` Methods

2.7.2 `printf()` Method

The `printf()` method can be used to format the numerical output to the console.

Table 2.6 lists some of the format specifiers in Java.

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

Table 2.6: Format Specifiers in Java

Code Snippet 12 demonstrates the use of `printf()` methods to format the output.

Code Snippet 12:

```
public class FormatSpecifier {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        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);
}
}
```

The output of the code is shown in Figure 2.14.

Figure 2.14: Output of Format Specifiers

2.7.3 *format () Method*

This method formats multiple arguments based on a format string. The format string consists of the normal string literal information associated with format specifiers and an argument list.

The syntax of a format specifier is as follows:

Syntax:

`%[arg_index$] [flags] [width] [.precision] conversion character`

where,

`arg_index`: Is an integer followed by a \$ symbol. The integer indicates that the argument should be printed in the mentioned position.

`flags`: Is a set of characters that format the output result. There are different flags available in Java.

Table 2.7 lists some of the flags available in Java.

Flag	Description
"_"	Left justify the argument
"+"	Include a sign (+ or -) with this argument
"0"	Pad this argument with zeros
";"	Use locale-specific grouping separators
"("	Enclose negative numbers in parenthesis

Table 2.7: Types of Flags in Java

width: Indicates the minimum number of characters to be printed and cannot be negative.

precision: Indicates the number of digits to be printed after a decimal point. Used with floating-point numbers.

conversion character: Specifies the type of argument to be formatted. For example, **b** for boolean, **c** for char, **d** for integer, **f** for floating-point, and **s** for string.

The values within '[]' are optional. The only required elements of format specifier are the % and a conversion character.

Code Snippet 13 demonstrates the `format()` method.

Code Snippet 13:

```
public class VariableScope {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int num = 2;
        double result = num * num;
        System.out.format("The square root of %d is %f.%n", num,
result);
    }
}
```

The output of the code is shown in Figure 2.15.

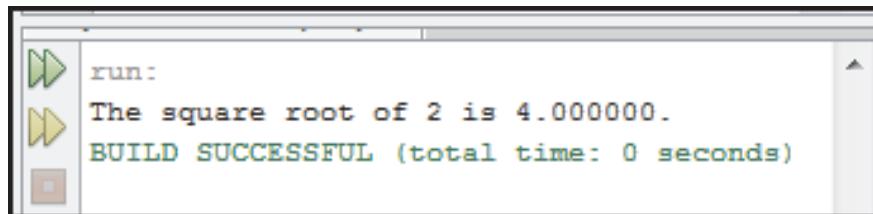


Figure 2.15: Output of `format()` Method

2.7.4 Formatted Input

The `Scanner` class allows the user to read or accept values of various data types from the keyboard. It breaks the input into tokens and translates individual tokens depending on their data type.

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

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

Table 2.8: Methods of Scanner Class

Code Snippet 14 demonstrates the `Scanner` class methods and how they can be used to accept values from the user.

Code Snippet 14:

```
import java.util.*;
public class FormattedInput {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Creates an object and passes the inputstream object
        Scanner s = new Scanner(System.in);

        System.out.println("Enter a number:");
        // Accepts integer value from the user
        int intValue = s.nextInt();

        System.out.println("Enter a decimal number:");
        // Accepts integer value from the user
        float floatValue = s.nextFloat();
        System.out.println("Enter a String value");
        // Accepts String value from the user
        String strValue = s.next();

        System.out.println("Values entered are: ");
        System.out.println(intValue + " " + floatValue + " " + strValue);
    }
}
```

The output of the code is shown in Figure 2.16.

```

run:
Enter a number:
23456
Enter a decimal number:
9876.76545
Enter a String value
John
Values entered are:
23456 9876.766 John
BUILD SUCCESSFUL (total time: 19 seconds)

```

Figure 2.16: Output of Scanner Class

2.8 Operators

All programming languages provide some mechanism for performing various operations on the data stored in variables. The simplest form of operations involves arithmetic (such as adding, dividing, and so on) 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 as **Operands**. The combination of both the operator and the operands, `z = x + y`, is known as an **Expression**.

Java provides several categories of operators as shown in Table 2.9.

Operator Type	Category	Example
Arithmetic	Used to perform basic mathematical operations such as addition, subtraction, multiplication, and division	<code>a + b, a - b, a * b, a / b</code>
Unary	Used to perform operations on a single operand such as incrementing, decrementing, negating, or inverting a value	<code>a++, a--, -a, !a</code>
Assignment	Used to assign a value to a variable	<code>a = b, a += b, a -= b</code>
Relational	Used to compare two values and return a boolean result	<code>a < b, a > b, a <= b, a >= b</code>
Equality	Used to check if two values are equal or not and return a boolean result	<code>a == b, a != b</code>
Logical	Used to perform logical operations such as AND, OR, and NOT on boolean values	<code>a && b, `a</code>

Operator Type	Category	Example
Ternary	Used to evaluate a condition and return one of two values depending on whether the condition is true or false	(a > b) ? a : b
Bitwise	Used to perform operations on individual bits of integer values such as AND, OR, XOR, complement, left shift, and right shift	a & b, `a
instanceof	Used to check if an object is an instance of a specific class or interface and return a boolean result	obj instanceof String
Conditional	It test the relationship between two operands and displays the result in Boolean values either true or false	=, !=, >, <, >=, <=

Table 2.9: List of Operators

2.8.1 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 2.10 lists order of precedence of operators from highest to lowest in which operators are evaluated in Java.

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

Table 2.10: 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.

For example, consider following expression:

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

The evaluation of the expression based on its operators precedence is as follows:

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`

2.8.2 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 2.11 shows Java operators and their associativity.

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

Table 2.11: Java Operators and their Associativity

Consider following expression:

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

- 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.

$$6 - 30$$

The result is -14.

2.9 Type Casting

In any application, there may be situations where one data type may require 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. 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 OOP languages, type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

There are two types of conversion: **implicit** and **explicit**. The term for implicit type conversion is coercion. The most common form of explicit type conversion is known as casting. Explicit type conversion can also be achieved with separately defined conversion routines such as an overloaded object constructor.

2.9.1 Implicit Type Casting

When a data of a particular type is assigned to a variable of another type, then automatic type conversion takes place. It is 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

Figure 2.17 shows the implicit type casting.

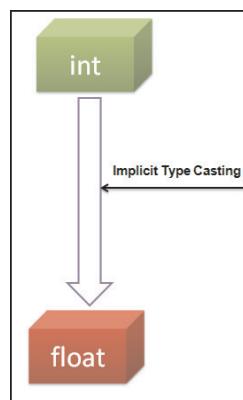


Figure 2.17: Implicit Type Casting

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. 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.

Code Snippet 15 demonstrates implicit type conversion.

Code Snippet 15:

```
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
...
```

2.9.2 Explicit 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 syntax for explicit casting is as follows:

Syntax:

```
(target data type) value;
```

Figure 2.18 shows the explicit type casting of data types.

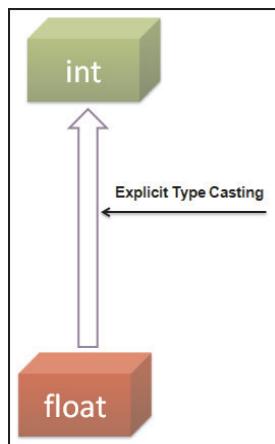


Figure 2.18: Explicit Type Casting

Code Snippet 16 adds a `float` value to an `int` and stores the result as an integer.

Code Snippet 16:

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

The `float` value in `a` 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.

2.10 Check Your Progress

1. Match the following operators in Java against their corresponding description.

Operator		Description	
a.	Conditional	1.	Requires two operands to operate
b.	Arithmetic	2.	Tests the relationship between two operands
c.	Logical	3.	Requires only one operand
c.	Unary	4.	Works on boolean expressions

(A)	a-1, b-2, c-3, d-4	(C)	a-2, b-1, c-4, d-3
(B)	a-4, b-3, c-2, d-1	(D)	a-3, b-4, c-1, d-2

2. Which of the following operators is used to change the order of evaluation while evaluating an expression?

(A)	Unary	(C)	Arithmetic
(B)	Parentheses	(D)	Relational

3. Which of these statements about escape sequences are true?

a.	The escape sequence character \n represents a new line character
b.	The \u escape sequence represents a Unicode character
c.	The escape sequence character \r represents a backslash character
d.	The escape sequence character \\ represents a new line character

(A)	a, b, and c	(C)	b, c, and d
(B)	c and d	(D)	a and b

4. Match the following data types with the values that can optimally fit in that data type.

Data Type		Value	
a.	byte	1.	5000
b.	char	2.	48999.988
c.	double	3.	'F'
c.	int	4.	256

(A)	a-1, b-2, c-3, d-4	(C)	a-2, b-1, c-4, d-3
(B)	a-4, b-3, c-2, d-1	(D)	a-3, b-4, c-1, d-2

5. 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?

a.	System.out.println("dblTest=" + dblTest); int sum = (int) (dblTest + floatTest);
b.	System.out.println("sum=" + sum);
c.	boolean boolTest = true; float floatTest = 3.14f;
d.	System.out.println("floatTest=" + floatTest); dblTest = (double) (boolTest?1:0);
e.	double dblTest = 0.000000000000053; floatTest = (float) (boolTest?1:0);

(A)	c, e, d, a, b	(C)	c, d, e, b, a
(B)	e, c, d, a, b	(D)	e, c, b, a, d

2.10.1 Answers

1.	C
2.	B
3.	D
4.	B
5.	A

```
g package;
import java.util.*;
public class Main {
    public static void main() {
        String str = "Hello Java";
        System.out.println(str);
    }
}
```

Summary

- Variables store values required in the program and should be declared before they are used. In Java, variables can be declared within a class, method, or within any block.
- 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.
- A literal signifies a value assigned to a variable in the Java program.
- The output of a Java program can be formatted using these: print() and println(), printf(), format().
- Operators are symbols that help to manipulate or perform some sort of function on data.
- Parentheses are used to change the order in which an expression is evaluated.
- The type casting feature helps in converting a certain data type to another data type.

Try It Yourself

1. Write a program that declares variables of different data types. During declaration, try to use some invalid variable names in the program and observe the compiler errors generated for the same.

2. Execute the following code and check the output:

```
class TestOperator {
    public static void main(String[] args) {
        int i = 5;
        i++;
        System.out.println(i);
        ++i;
        System.out.println(i);
        System.out.println(++i);
        System.out.println(i++);
        System.out.println(i);
    }
}
```

3. Write a program that uses the arithmetic operators to calculate the area and circumference of a circle. Use bitwise operators to perform some bitwise operations on two integers, and the conditional operator to compare the results of the bitwise operations.

The program should take three inputs from the user: the radius of the circle, and the two integers for the bitwise operations.

The program should display the area and circumference of the circle, the results of the bitwise operations, and the comparison of the results using the conditional operator.

Onlinevarsity

**24 x 7
Access To
Learning**



Session - 3

Decision-Making Constructs and Loops

Welcome to the Session, **Decision-Making Constructs and Loops**.

This session explains different types of decision-making statements present in Java programming language. It focuses on two main aspects of making decisions: first is the comparison of data and second is the sequence of execution of statements. This session also explains different types of looping statements available in Java programming language. The session also covers various jump statements, also called as branching statements used in Java.

In this Session, you will learn to:

- List different types of decision-making statements
- Explain the if statement and various forms of if statement
- Explain switch-case statement
- Compare the if-else and switch-case statement
- List different types of loops
- Explain the while statement and the associated rules
- Identify the purpose of the do-while statement
- Identify the necessity of the for statement
- Describe nested loops
- Compare different types of loops
- Illustrate the purpose of jump statements
- Describe break statement
- Describe continue statement



3.1 Introduction

A Java program consists of 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 change in the flow of statements is achieved by using different **control flow statements**. There are three categories of control flow statements supported by Java programming language. These are as follows:

- **Conditional Statements** - These types of statements are also referred to as decision-making statements. They allow the program to execute a particular set of statements depending on the result of evaluation of a conditional expression or the state of a variable.
- **Iteration Statements** - These types of statements are also referred to as looping constructs. They allow the program to repeat a particular set of statements for certain number of times.
- **Branching Statements** - These types of statements are referred to as jump statements. They either allow the program to skip or continue execution of the looping statements. The program is executed in a non-linear fashion.

3.1.1 Decision-making Statements

The Java programming language possesses different decision-making capabilities. Decision-making statements enable us to change the flow of execution of a Java program. Based on the result of evaluation of a condition during program execution, a statement or a sequence of statements is executed. Different types of decision-making statements supported by Java are as follows:

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

3.2 *if* Statement

The `if` statement is the most basic form of decision-making statement. The `if` statement evaluates a given condition and based on the result of evaluation executes a certain section of code. If the condition evaluates to `true`, then the statements present within the `if` block gets executed. If the condition evaluates to `false`, the control is transferred directly to the statement outside the `if` block.

Figure 3.1 shows the flow of execution for the `if` statement.

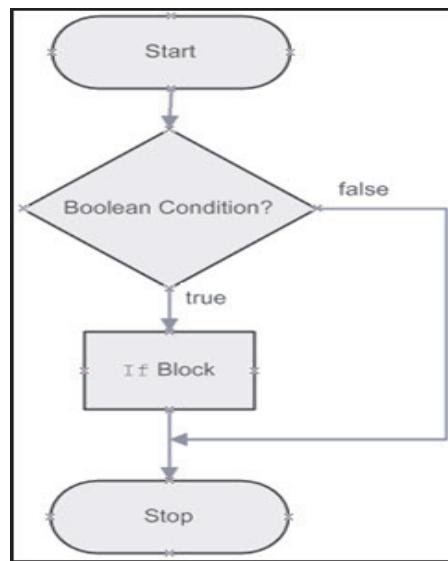


Figure 3.1: Flow of Execution – `if` Statement

The syntax for using the `if` statement is as follows:

Syntax:

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

where,

condition: Is the boolean expression.

statements: Are instructions/statements enclosed in curly braces. These statements are executed when the boolean expression evaluates to `true`.

Code Snippet 1 demonstrates the code that performs conditional check on the value of a variable using the `if` statement.

Code Snippet 1:

```
public class CheckNumberValue {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int first = 400, second = 700, result;
        result = first + second;
        // Evaluates the value of result variable
    }
}
```

```

if (result > 1000) {
    second = second + 100;
}
System.out.println ("The value of second is " + second);
}
}

```

The program tests the value of the variable, **result** and accordingly calculates value for the variable, **second**. If the value of **result** is greater than 1000, then the value of the variable **second** is incremented by 100. If the evaluation of condition is false, the value of the variable **second** is not incremented. Finally, the value of the variable **second** gets printed on the console.

The output of the code is shown in Figure 3.2.

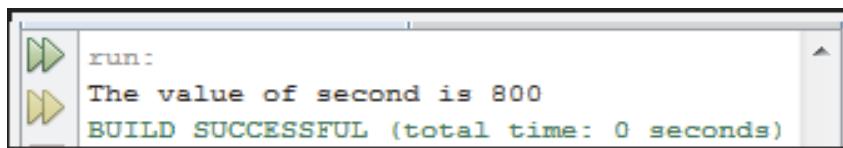


Figure 3.2: Output of Code With Simple if Statement

If there is only a single action statement within the body of the **if** block, then use of opening and closing curly braces is optional.

For example, Code Snippet 1 is rewritten in Code Snippet 2 to illustrate this concept.

Code Snippet 2:

```

public class ModifiedIf {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int first = 400, second = 700, result;
        result = first + second;
        // Evaluates the value of result variable
        if (result > 1000)
            second = second + 100;
        System.out.println ("The value of second is " + second);
    }
}

```

The main disadvantage of omitting braces is that if another statement is added to the body of the **if** statement, without enclosing the statements in curly braces, it would result in wrong output.

3.2.1 if-else Statement

The `if` statement specifies a block of statement to be executed when the condition is evaluated to `true`.

However, sometimes it is required to define a block of statements to be executed when a condition evaluates to `false`. This is done by using the `if-else` statement.

The `if-else` statement begins with the `if` block followed by the `else` block. The `else` block specifies a block of statements that are to be executed when a condition evaluates to `false`.

The syntax for using the `if-else` statement is as follows:

Syntax:

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

Code Snippet 3 demonstrates the code that checks whether a number is even or odd.

Code Snippet 3:

```
public class Number_Division {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int number = 11, remainder;
        // % operator to return the remainder of the division
        remainder = number % 2;
        if (remainder == 0) {
            System.out.println("Number is even");
        } else {
            System.out.println("Number is odd");
        }
    }
}
```

In Code Snippet 3, the variable, `number` is divided by 2 to obtain the remainder of the division. This is done by using the `%` (modulus) operator which returns the remainder after performing the division. If the remainder is 0, the message 'Number is even' is printed. Otherwise, the message 'Number is odd' is printed.

The output of the code is shown in Figure 3.3.

```
run:
Number is odd
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.3: Output of Code With if-else Statement

3.2.2 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 within another if statement. This is known as nested-if. A nested-if statement is an if statement that is the target of another if or else statement.

Important points to remember about nested-if statements are as follows:

- An else statement should always refer to the nearest if statement.
- The if statement must be within the same block as the else and it should not be already associated with some other else statement.

The syntax to use the nested-if statements is as follows:

Syntax:

```
if(condition) {
    if(condition)
        true-block statement(s);
    else
        false-block statement(s);
}
else {
    false-block statement(s);
}
```

Code Snippet 4 demonstrates the use of nested-if statement and checks whether a number is divisible by 3 as well as 5.

Code Snippet 4:

```
import java.util.Scanner;
public class NumberDivisibility {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Scanner class is used to accept values from the user
```

```

Scanner input = new Scanner(System.in);
System.out.println("Enter a Number: ");
int num = input.nextInt();
// Checks whether a number is divisible by 3
if (num % 3 == 0) {
    System.out.println("Inside Outer if Block");
    // Inner if statement checks if number is divisible by 5
    if (num % 5 == 0) {
        System.out.println("Number is divisible by 3 and 5");
    } else {
        System.out.println("Number is divisible by 3, but not by 5");
    } // End of inner if-else statement
}
else {
    System.out.println("Number is not divisible by 3");
} // End of outer if-else statement
}
}

```

Code Snippet 4 declares a variable **num** and stores an integer value accepted from the user. The code contains an outer **if** statement, `if(num % 3 == 0)` and an inner **if** statement, `if (num % 5 == 0)`. Initially, the outer **if** statement is evaluated. If it evaluates to **false**, then the inner **if-else** statement is skipped and the final **else** block is executed. If the outer **if** statement evaluates to **true**, then its body containing the inner **if-else** statement is evaluated. In other words, evaluation of the inner **if** statement depends on the result of the outer **if** statement.

The output of the code is shown in Figure 3.4.

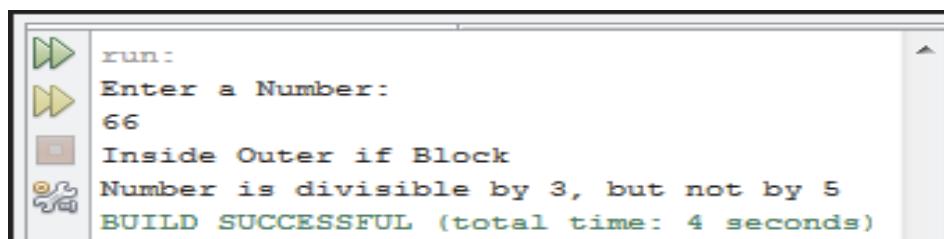


Figure 3.4: Output of Code With Nested-if Statement

3.2.3 *if-else-if Ladder*

The multiple **if** construct is known as the **if-else-if ladder**. Conditions are evaluated sequentially starting from the top of the ladder and moving downwards.

When a condition controlling the `if` statement is evaluated as `true`, the statements associated with the `if` condition are executed and all the other `if` conditions are bypassed. If none of the condition is `true`, then, the final `else` statement is executed. The final `else` statement also acts as a default statement.

In case, if all the `if` constructs are `false` and no final `else` statement is specified, then no action is performed by the program.

The syntax for using the `if-else-if` statement is as follows:

Syntax:

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

Figure 3.5 shows the flow of execution for the `if-else-if` ladder.

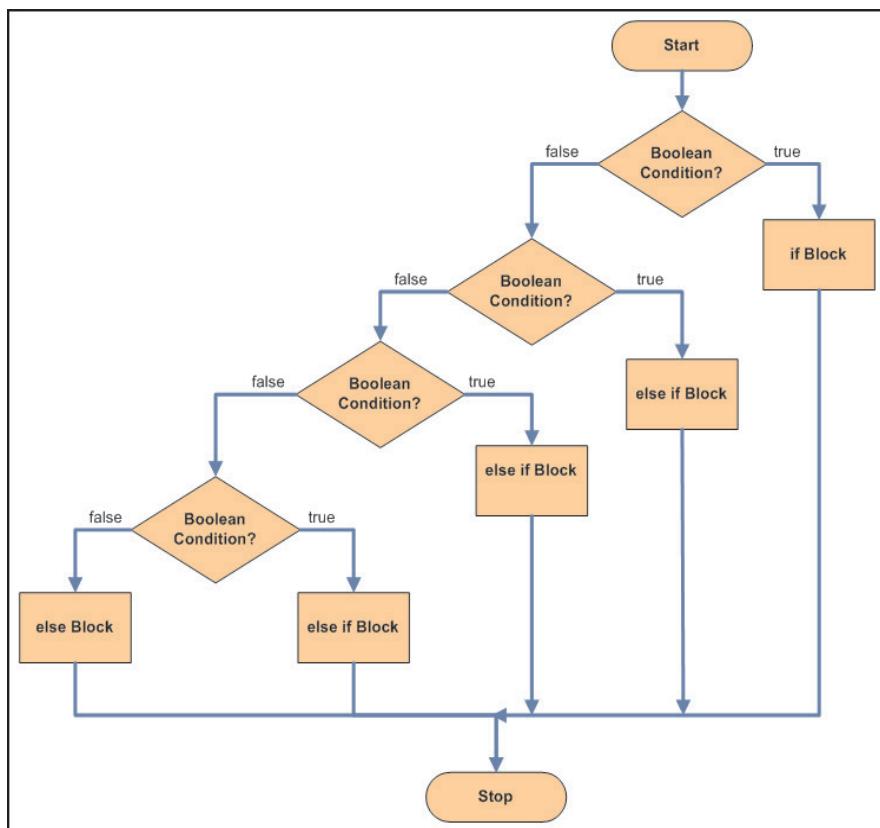


Figure 3.5: Flow of Execution - `if-else-if` Ladder

Code Snippet 5 checks the total marks and prints the appropriate grade.

Code Snippet 5:

```
public class CheckMarks {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int totalMarks = 59;
        /* Tests the value of totalMarks and accordingly transfers control to the
        else if statement
        */
        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 = C");
        } else {
            System.out.println("Fail");
        }
    }
}
```

If the code satisfies a given condition, then the statements within that `else if` condition are executed. After execution of the statements, the control breaks and remaining `if` conditions are bypassed for evaluation. If none of the condition is satisfied, then the final `else` statement, also known as the default `else` statement is executed.

The output of the code is shown in Figure 3.6.

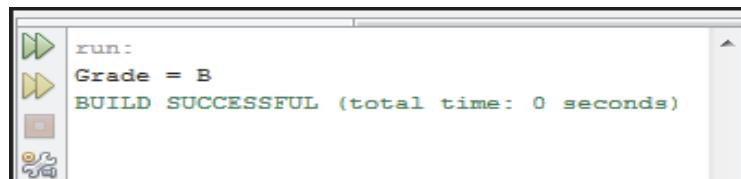


Figure 3.6: Output of Code With `if-else-if` Ladder

3.3 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` statements can be used as an alternative for multiple selections. The use of the `switch-case` statement results in better performance.

The `switch-case` statement contains a variable as an expression whose value is compared against different values. It can have a number of possible execution paths depending on the value of expression provided with the `switch` statement. The expression to be evaluated can contain different primitive data types, such as `byte`, `short`, `int`, and `char`.

Apart from strings, it also supports objects of few classes present in the Java API. The classes are `Character`, `Byte`, `Short`, and `Integer` and are referred to as wrapper classes. A wrapper class encloses or wraps the primitive data type into an object of that type. For example, the wrapper class, `Integer` allows you to use `int` value as object, that is, `Integer y = new Integer(52);`. It also supports the use of enumerated types as expression.

The syntax for using the `switch-case` statement is as follows:

Syntax:

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

where,

`switch`: The `switch` keyword is followed by an expression enclosed in parentheses.

`case`: The `case` keyword is followed by a constant and a colon. Each case value is a unique literal. The `case` statement might be followed by a code sequence that is 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 to the `else` of the `if-else-if` statement.

break: The `break` statement is used inside the `switch-case` statement to terminate the execution of the statement sequence. The `break` statement is optional. If there is no `break` statement, execution flows sequentially into the next cases. Sometimes, multiple cases can be present without `break` statements between them. The use of `break` statement makes the modification in code easier and with less error.

Figure 3.7 shows the flow of execution for the `switch-case` statement.

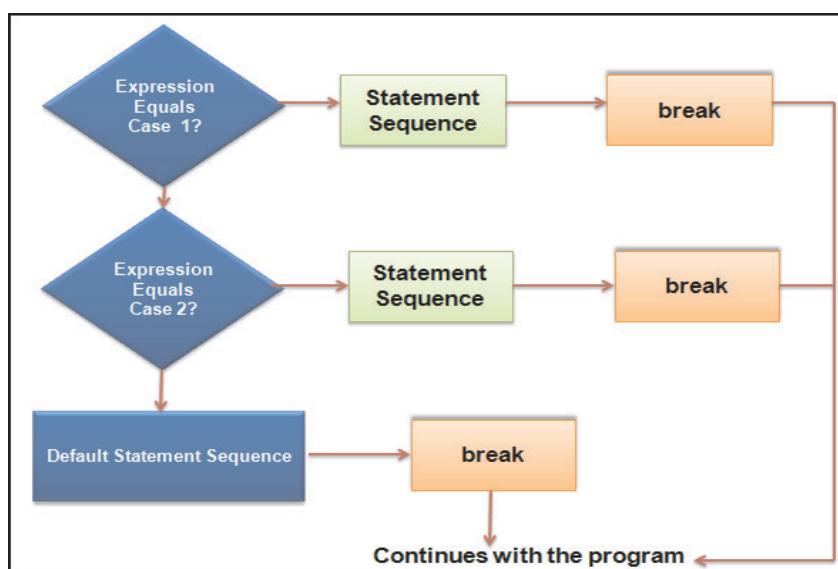


Figure 3.7: switch-case Statement

The value of the expression specified with the `switch` statement is compared with each case constant value. If any case value matches, the corresponding statements in that case are executed. If there is no matching case, then the `default` case is executed.

When the `break` statement is encountered, it terminates the `switch-case` block and control switches to the statements following the block. The `break` statement must be provided with each case, as without them `switch` blocks fall through. This means even after the matching case is executed; all other cases following the matching case are also executed, until a `break` statement is encountered.

Code Snippet 6 demonstrates the use of the `switch-case` statement.

Code Snippet 6:

```

public class TestNumericOperation {
    /**
     * @param args the command line arguments
     */
    
```

```

public static void main(String[] args) {
    // Declares and initializes the variable
    int choice = 3;
    // switch expression value is matched with each case
    switch (choice) {
        case 1:
            System.out.println("Addition");
            break;
        case 2:
            System.out.println("Subtraction");
            break;
        case 3:
            System.out.println("Multiplication");
            break;
        case 4:
            System.out.println("Division");
            break;
        default:
            System.out.println("Invalid Choice");
    } // End of switch-case statement
}
}

```

In Code Snippet 6, value of the expression, `choice` is compared with the literal value in each of the `case` statement. Here, `case 3` is executed, as its value is matching with the expression. Finally, the control moves out of the `switch-case`, due to the presence of the `break` statement. The program will not perform any action, if no matching case is found or the `default` statement is not present.

The output of the code is shown in Figure 3.8.

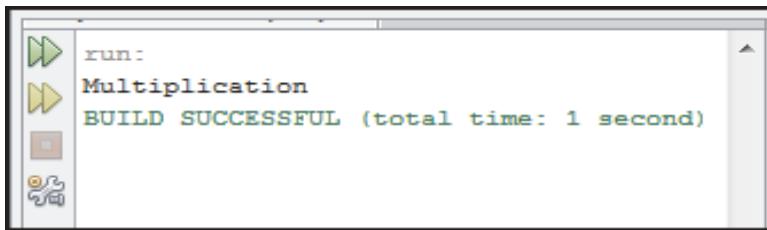


Figure 3.8: Output of Code With Simple switch-case Statement

As discussed, the `break` statement is optional. If it is omitted, then execution continues with the other cases, even after the matching case is found. However, sometimes it is required to have multiple `case` statements without a `break` statement. This is commonly used when same set of statements are required

to be executed for the multiple cases.

Code Snippet 7 demonstrates the use of multiple case statements with no break statement.

Code Snippet 7:

```
public class NumberOfDays {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        int month = 5;  
        int year = 2001;  
        int numDays = 0;  
        // Cases are executed until a break statement is encountered  
        switch (month) {  
            case 1:  
            case 3:  
            case 5:  
            case 7:  
            case 8:  
            case 10:  
            case 12:  
                numDays = 31;  
                break;  
            case 4:  
            case 6:  
            case 9:  
            case 11:  
                numDays = 30;  
                break;  
            case 2:  
                if (year % 4 == 0) {  
                    numDays = 29;  
                } else {  
                    numDays = 28;  
                }  
                break;  
            default:        }
```

```

        System.out.println("InvalidMonth");
    } // End of switch-case statement
    System.out.println("Month: " + month);
    System.out.println("Number of Days: " + numDays);
}
}

```

In Code Snippet 7, the value of expression, `month` is compared through each case, till a `break` statement or end of the `switch-case` block is encountered. The output of the code is shown in Figure 3.9.

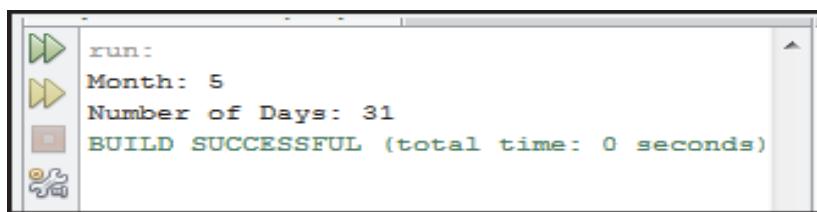


Figure 3.9: Output of Code With Multiple Cases Without `break` Statement

3.3.1 String-based `switch-case` Statement

Allowing the use of strings in `switch-case` statement enables the program to incorporate a readable code. A string is not a primitive data type, but an object in Java. Thus, to use strings for comparison, a `String` object is passed as an expression in the `switch-case` statement.

Code Snippet 8 demonstrates the use of strings in the `switch-case` statement.

Code Snippet 8:

```

public class DayofWeek {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String day = "Monday";
        // switch statement contains an expression of type String
        switch (day) {
            case "Sunday":
                System.out.println("First day of the Week");
                break;
            case "Monday":
                System.out.println("Second Day of the Week");
                break;
        }
    }
}

```

```

case "Tuesday":
    System.out.println("Third Day of the Week");
    break;
case "Wednesday":
    System.out.println("Fourth Day of the Week");
    break;
case "Thursday":
    System.out.println("Fifth Day of the Week");
    break;
case "Friday":
    System.out.println("Sixth Day of the Week");
    break;
case "Saturday":
    System.out.println("Seventh Day of the Week");
    break;
default:
    System.out.println("Invalid Day");
} // End of switch-case statement
}
}

```

In Code Snippet 8, the statement `String day="Monday"` creates an object named `day` of type `String` and initializes it. Then, the object is passed as an expression to the `switch` statement. The value of this expression, that is "Monday", is compared with the value of each `case` statement. If a matching case is not found, then the `default` statement is executed.

The output of the code is shown in Figure 3.10.

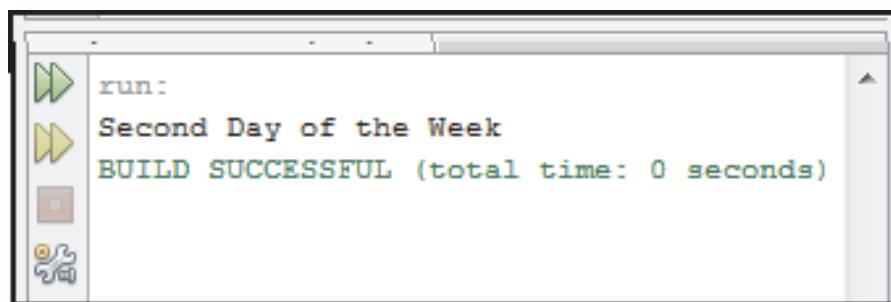


Figure 3.10: Output of Code With `switch-case` Statement Having String Expression

There are some points that must be considered while using strings with the `switch-case` statement. These are as follows:

- **Null values** – A runtime exception is generated when a string variable is assigned a `null` value and is passed as an expression to the `switch` statement. The runtime exception thrown by JVM is `java.lang.NullPointerException`.
- **Case-sensitive values** – The value of `String` variable that is matched with the case literals is case sensitive. This means, if a `String` value "`Monday`" is matched with the case labeled "`MONDAY`":, then it will not be treated as a matched value. Hence, in that case, the `default` statement will be executed.

3.3.2 Enumeration-based `switch-case` Statement

The `switch-case` statement also supports the use of an enumeration (`enum`) value in the expression. The only constraint with an `enum` expression is that all case constants must belong to the same `enum` variable used with the `switch` statement.

Code Snippet 9 demonstrates the use of enumerations in the `switch-case` statement.

Code Snippet 9:

```
public class TestSwitchEnumeration {
    /**
     * An enumeration of Cards Suite
     */
    enum Cards {
        Spade, Heart, Diamond, Club
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Cards card = Cards.Diamond;
        // enum variable is used to control a switch statement
        switch (card) {
            case Spade:
                System.out.println("SPADE");
                break;
            case Heart:
                System.out.println("HEART");
                break;
            case Diamond:
                System.out.println("DIAMOND");
                break;
        }
    }
}
```

```

        System.out.println("DIAMOND");
        break;
    case Club:
        System.out.println("CLUB");
        break;
    } // End of switch-case statement
}
}

```

In Code Snippet 9, the `enum`, `card` is passed as an expression to the `switch` statement. Each `case` statement has an enumeration constant associated with it and does not require it to be qualified by the enumeration name.

This is because during compilation, the `switch` statement with an `enum`, implicitly understands the type of constants used with the `case` statements.

The output of the code is shown in Figure 3.11.

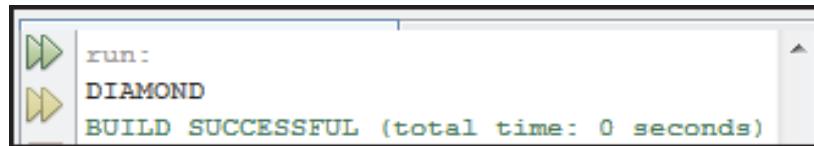


Figure 3.11: Output of Code With `switch-case` Statement Having Enumeration

3.3.3 Nested `switch-case` Statement

The `switch-case` statement can also be used as a part of another `switch-case` statement. This is called as `nested switch-case` statements.

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

Code Snippet 10 demonstrates the use of nested `switch-case` statements.

Code Snippet 10:

```

public class Greeting {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // String declaration
        String day = "Monday";
        String hour = "am";
    }
}

```

```

// Outer switch statement
switch (day) {
    case "Sunday":
        System.out.println("Sunday is a Holiday...");
        // Inner switch statement
        switch (hour) {
            case "am":
                System.out.println("Good Morning");
                break;
            case "pm":
                System.out.println("Good Evening");
                break;
        } // End of inner switch-case statement
        break; // Terminates the outer case statement
    case "Monday":
        System.out.println("Monday is a Working Day...");
        switch (hour) {
            case "am":
                System.out.println("Good Morning");
                break;
            case "pm":
                System.out.println("Good Evening");
                break;
        } // End of inner switch-case statement
        break;
    default:
        System.out.println("Invalid Day");
}
} // End of the outer switch-case statement
}

```

In Code Snippet 10, the variable, `day` is used as an expression with the outer `switch` statement. It is compared with the list of cases provided with the outer `switch-case` statements. If the value of `day`

variable matches with "Sunday" or "Monday", then the inner switch-case statement is executed. The inner switch statement compares the value of `hour` variable with case constants "am" or "pm".

The output of the code is shown in Figure 3.12.

```
run:
Monday is a Working Day...
Good Morning
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.12: Output of Code With Nested switch-case Statements

The three important features of switch-case statements are as follows:

- The switch-case statement differs from the if statement, as it can only test for equality. The if statement can test any type of boolean expression. In other words, the switch-case statement 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 statement can have identical values, except the nested switch-case statements.
- A switch statement is more efficient and executes faster than a set of nested-if statements.

3.4 Comparison Between if and switch-case Statement

Though, the 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-case statement.

if	switch-case
Each if statement 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, int, or String
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: Difference Between if and switch-case Statement

3.5 Loops

A computer program consists of 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. Then, 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 3.13 shows a pseudocode that displays the multiples of 10.

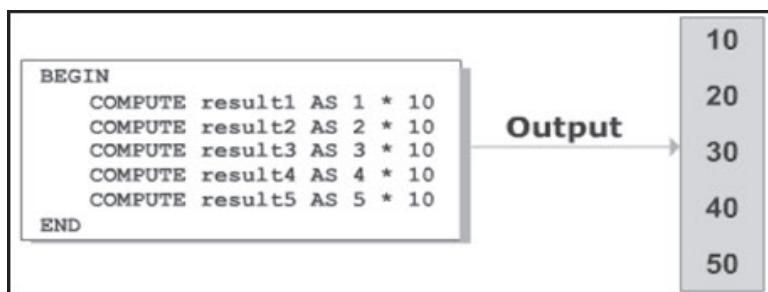


Figure 3.13: Pseudocode – Multiples of 10

As shown in Figure 3.13, the same statement is repeating five times to display the multiple of 10 with 1, 2, 3, 4, and 5. Thus, a loop can be used in this situation.

3.5.1 Looping Statements

Loops enable programmers to develop concise programs, which otherwise would require thousands of program statements. A loop consists of statement or a block of statements that are repeatedly executed, until a condition evaluates to `true` or `false`. The loop statements supported by Java programming language are as follows:

- ➔ `while` statement
- ➔ `do-while` statement
- ➔ `for` statement
- ➔ `for-each` statement

3.6 while Statement

The `while` statement is the most fundamental looping statement in Java. It is used to execute a statement or a block of statements until the specified condition is `true`. Normally, it is used when the number of times the block has to be executed is not known.

The syntax to use the `while` statement is as follows:

Syntax:

```
while (expression) {
```

```
// one or more statements
```

```
}
```

where,

`expression`: Is a conditional expression which must return a boolean value, that is, `true` or `false`.

The use of curly braces (`{ }`) is optional. They can be avoided, if there is only a single statement within the body of the loop. However, providing statements within the curly braces increases the readability of the code.

The body of the loop contains a set of statements. These statements will be executed until the conditional expression evaluates to `true`. When the conditional expression evaluates to `false`, the loop is terminated and the control passes to the statement immediately following the loop.

Code Snippet 11 demonstrates the code that displays multiples of 10 using the `while` loop.

Code Snippet 11:

```
public class PrintMultiplesWithWhileLoop {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Variable, num acts as a counter variable
        int num = 1;
        // Variable, product will store the result
        int product = 0;
        // Tests the condition at the beginning of the loop
        while (num <= 5) {
            product = num * 10;
            System.out.printf("\n %d * 10 = %d", num, product);
        }
    }
}
```

```

        num++; // Equivalent to n = n + 1
    } // Moves the control back to the while statement
    // Statement gets printed on loop termination
    System.out.println("\n Outside the Loop");
}
}

```

In Code Snippet 11, an integer variable, `num` is declared to store a number. The variable `num` is initialized to 1 and is used in the `while` loop to start multiplication from 1. The conditional expression `num <= 5` is evaluated at the beginning of the `while` loop. This ensures that the body of the loop is executed only if the conditional expression evaluates to `true`. In this case, as the value in the variable, `num` is less than 5, hence, the statements present in the body of the loop is executed. The first statement within the body of the loop calculates the product by multiplying `num` with 10. The next statement prints this value. The last statement `num++` increments the value of `num` by 1. The loop continues as long as the value of `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 output of the code is shown in Figure 3.14.

```

run:
1 * 10 = 10
2 * 10 = 20
3 * 10 = 30
4 * 10 = 40
5 * 10 = 50
Outside the Loop
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 3.14: Output of Code With `while` Statement

3.6.1 Rules for Using `while` Loop

Following points should be noted when using the `while` statement:

- Value 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. For example, `num = 1;`
- The body of the loop must have an expression that changes the value of the variable which is a part of the loop's expression. For example, `num++;` or `num--;`

3.6.2 Infinite Loop

An infinite loop is one which never terminates. The loop runs infinitely when the conditional expression or the increment/decrement expression of the loop is missing. Any type of loop can be an infinite loop.

Code Snippet 12 shows the implementation of an infinite loop using the `while` statement.

Code Snippet 12:

```
public class InfiniteWhileLoop {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        /*
         * Loop begins with a boolean value true and is executed
         * infinitely as the terminating condition is missing
         */
        while (true) {
            System.out.println("Welcome to Loops...");
        } //End of the while loop
    }
}
```

In Code Snippet 12, the conditional expression is set to a boolean value, `true`. The loop never terminates as the expression always returns a `true` value. This leads to an infinite loop. The program executing the loop infinitely must be terminated manually or a `break` statement should be used to terminate such loops.

3.7 do-while Statement

In the `while` loop, evaluation of conditional expression is done at the beginning of the loop. Hence, if the condition is initially `false`, then the body of the loop is not executed at all. Thus, to execute the body of the loop at least once, it is desirable to use the `do-while` loop.

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

The syntax to use the `do-while` statement is as follows:

Syntax:

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

`while (expression);`

where,

`expression`: A conditional expression which must return a boolean value, that is, `true` or `false`.

statement (s): Indicates body of the loop with a set of statements.

For each iteration, the `do-while` loop first executes the body of the loop and then, the conditional expression is evaluated. When the conditional expression evaluates to `true`, the body of the loop executes. When the conditional expression evaluates to `false`, the loop terminates and the statement following the loop is executed.

Code Snippet 13 demonstrates the use of `do-while` loop for finding the sum of 10 numbers.

Code Snippet 13:

```
public class SumOfNumbers {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int num = 1, sum = 0;
        /*
         * The body of the loop is executed first, then the condition is evaluated
         */
        do {
            sum = sum + num;
            num++;
        } while (num <= 10);
        // Prints the value of variable after the loop terminates
        System.out.printf("Sum of 10 Numbers: %d\n", sum);
    }
}
```

In Code Snippet 13, 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 increments the value of `num` 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.

The output of the code is shown in Figure 3.15.

```
run:
Sum of 10 Numbers: 55
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 3.15: Output of Code With `do-while` Loop

3.8 *for Statement*

The `for` loop is especially used when the user knows the number of times the statements are required to be executed. 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 syntax to use the `for` statement is as follows:

Syntax:

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

where,

initialization: Is an expression that will set the initial value of the loop control variable.

condition: Is a boolean expression that test the value of loop control variable. If the condition expression evaluates to `true`, the loop executes. If condition expression evaluates to `false`, the loop terminates.

increment/decrement: Comprises statement that changes the value of the loop control variable (`s`) in each iteration, till 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. There is no semicolon at the end of the increment/decrement expressions. All the three declaration parts are separated by semicolons (`;`).

The execution of the loop starts with the initialization section. Generally, this is an expression that sets the value of the loop control variable and acts as a counter variable that controls the loop. The initialization expression is executed only once, that is, when the loop starts. Next, the boolean expression is evaluated and 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 value of the control variable. In the next iteration, again the condition section is evaluated and depending on the result of evaluation the loop is either continued or terminated.

Code Snippet 14 demonstrates the use of `for` statement for displaying multiples of 10.

Code Snippet 14:

```
public class PrintMultiplesWithForLoop {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int num, product;
        // The for Loop with all the three declaration parts
        for (num = 1; num <= 5; num++) {
            product = num * 10;
            System.out.printf("\n %d * 10 = %d", num, product);
        } // Moves the control back to the for loop
    }
}
```

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 increment statement, `num++`, increments the value of `num` by 1. The increment/decrement expression is evaluated after the first round of iteration and continues till the condition evaluates to `false`. Finally, the loop terminates when the condition becomes `false`, that is, when the value of `num` becomes equal to 6.

The output of the code is shown in Figure 3.16.

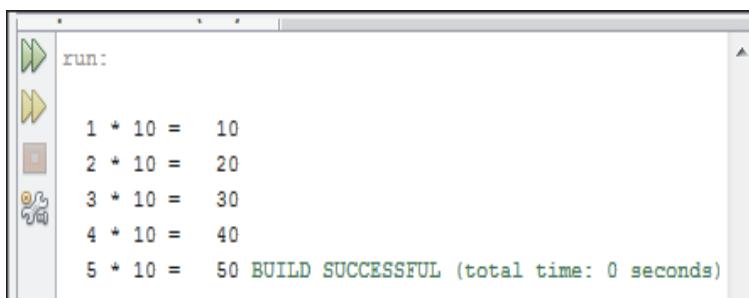


Figure 3.16: Output of Code With Simple `for` Loop

3.8.1 Scope of Control Variables in `for` Statement

Mostly control variables are used within the `for` loops and may not be used further in the program. In such a situation, it is possible to restrict the scope of variables by declaring them at the time of initialization.

Code Snippet 15 rewrites Code Snippet 5 to declare the counter variable inside the `for` statement.

Code Snippet 15:

```
public class ForLoopWithVariables {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int product;
        // The counter variable, num is declared inside the for loop
        for (int num = 1; num <= 5; num++) {
            product = num * 10;
            System.out.printf("\n %d * 10 = %d ", num, product);
        } // End of the for loop
    }
}
```

In Code Snippet 15, the variable `num` is not required further in the program, so, it has been declared inside the `for` statement. This restricts the scope of the variable, `num` to the `for` statement. The scope of variable completes when the loop terminates. The output of the program will be same as Code Snippet 5.

3.8.2 Use of Comma Operator in `for` Statement

The `for` statement can be extended by including more than one initialization or increment expressions in the `for` loop specification. The expressions are separated by using 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.

Code Snippet 16 demonstrates the use of `for` loop to print the addition table for two variables using the 'comma' operator.

Code Snippet 16:

```
public class ForLoopWithComma {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int i, j;
```

```

int max = 10;

/*
 * The initialization and increment/decrement section includes
 * more than one variable
 */

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

```

As shown in Code Snippet 16, 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 of `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 the code is shown in Figure 3.17.

Figure 3.17: Output of Code With `for` Loop Using Comma Operator

3.8.3 Variation in `for` Loop

The `for` loop is very powerful and flexible in its structure. This means that all three parts of the `for` loop is not required to be declared and used only within the loop.

The most common variation involves the conditional expression. Mostly, the conditional expression is tested with the targeted values, but, it can also be used for testing boolean expressions. Alternatively, the initialization or the iteration section in the `for` loop may be left empty, that is, they are not required to be present in the `for` loop.

Code Snippet 17 demonstrates the use of `for` loop without the initialization expression.

Code Snippet 17:

```
public class ForLoopWithNoInitialization {
    public static void main(String[] args) {
        /*
         * Counter variable declared and initialized outside for loop
         */
        int num = 1;
        /*
         * Boolean variable initialized to false
         */
        boolean flag = false;
        /*
         * The for loop starts with num value 1 and
         * continues till value of flag is not true
         */
        for (; !flag; num++) {
            System.out.println("Value of num: " + num);
            if (num == 5) {
                flag = true;
            }
        } // End of for loop
    }
}
```

The `for` loop in Code Snippet 17 continues to execute till the value of the variable `flag` is set to `true`.

The output of the code is shown in Figure 3.18.

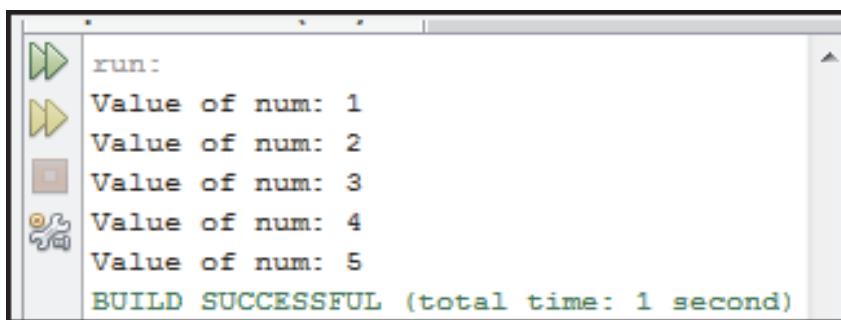


Figure 3.18: Output of Code With `for` Loop with Empty Initialization Section

3.8.4 Infinite `for` Loop

If all the three expressions are left empty, then it will lead to an infinite loop. The infinite `for` loop will run continuously because there is no condition specified to terminate it. Code Snippet 18 demonstrates the code for the infinite loop.

Code Snippet 18:

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

Code Snippet 18 will print 'This will go on and on' until the loop is terminated manually. The `break` statement can be used to terminate such loops. Infinite loops make 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.

3.8.5 Enhanced `for` Loop

The enhanced `for` loop is designed to retrieve or traverse through a collection of objects, such as an array. It is also used to iterate over the elements of the collection objects, such as `ArrayList`, `LinkedList`, `HashSet`, and so on. These classes are defined in the collection framework and are used to store objects.

The syntax for using the enhanced `for` loop is as follows:

Syntax:

```
for (type var: collection) {
    // block of statement
}
```

where,

`type`: Specifies the type of collection that is traversed.

`var`: Is an iteration variable that stores the elements from the collection. The `for-each` loop traverses from beginning to end and in each iteration the element is retrieved and stored in the `var` variable.

The enhanced `for` loop continues till all the elements from a collection are retrieved.

Table 3.2 shows the method for retrieving elements from an array object using enhanced `for` loop and its equivalent `for` loop.

for Loop	Enhanced for Loop
<pre>type var; for (int i = 0; i < arr.length; i++) { var = arr[i]; ... }</pre>	<pre>for (type var : arr) { ... // Body of the loop ... }</pre>

Table 3.2: Enhanced `for` Loop and its Equivalent `for` Loop

3.9 Nested Loops

The placing of a loop statement inside the body of another loop statement is called nesting of loops. 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 be executed till its condition section evaluates to `false`.

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 `for` loop can be nested within another `for` loop forming nested-for loop.

3.10 Jump Statements

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 must 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 within loops to change the flow of control based on conditions.

3.10.1 `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. If used within a set of nested loops, the `break` statement will terminate the innermost loop.

Code Snippet 19 demonstrates the use of `break` statement.

Code Snippet 19:

```
import java.util.Scanner;

public class AcceptNumbers {
    /**
     * @param args the command line arguments
     */

    public static void main(String[] args) {
        int count, number; // count variable is a counter variable
        for (count = 1, number = 0; count <= 10; count++) {
            // Scanner class is used to accept data from the keyboard
            Scanner input = new Scanner(System.in);
            System.out.println("Enter a number: ");
            number = input.nextInt();
            if (number == 0) {
                // break statement terminates the loop
                break;
            } // End if statement
        } // End of for statement
    }
}
```

In Code Snippet 19, the user is prompted to enter a number, and this is stored in the variable, `number`. This is repeated 10 times. However, if the user enters the number zero, the loop terminates and the control is passed to the next statement after the loop. The output of the code is shown in Figure 3.19 with different values entered by the user.

```
run:
Enter a number:
8
Enter a number:
6
Enter a number:
3
Enter a number:
0
BUILD SUCCESSFUL (total time: 8 seconds)
```

Figure 3.19: Output of Code Using `break` Statement

3.10.2 continue Statement

Java provides another keyword named `continue` to skip statements within a loop and proceed to the next iteration of the loop. In `while` and `do-while` loops, a `continue` statement transfers the control to the conditional expression which controls the loop.

Code Snippet 20 demonstrates the code that uses `continue` statement in printing the square and cube root of a number.

Code Snippet 20:

```
public class NumberRoot {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int count, square, cube;
        // Loop continues till the remainder of the division is 0
        for (count = 1; count < 300; count++) {
            if (count % 3 == 0) {
                continue;
            }
            square = count * count;
            cube = count * count * count;
            System.out.printf("\nSquare of %d is %d and Cube is %d", count, square,
            cube);
        } // End of the for loop
    }
}
```

Code Snippet 20 declares a variable `count` and uses the `for` statement which contains the initialization, termination, and increment expression. In the body of the loop, the value of `count` is divided by three 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 `count` is calculated and displayed.

The output of Code Snippet 20 is shown in Figure 3.20.

```

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
Square of 11 is 121 and Cube is 1331
Square of 13 is 169 and Cube is 2197
Square of 14 is 196 and Cube is 2744
Square of 16 is 256 and Cube is 4096
Square of 17 is 289 and Cube is 4913

```

Figure 3.20: Output of Code Using `continue` Statement

3.10.3 Labeled Statements

Java does not support `goto` statements, as they are difficult to understand and maintain. However, there are some situations where the `goto` statement proves to be useful. It can be used within constructs to control the flow of statements. For example, to exit from a deeply nested set of loops, `goto` statement can be useful.

Java defines an expanded form of `break` and `continue` statements. These expanded forms can be used within any block. It is not necessary that the blocks must be part of loop or a `switch` statement.

These forms are referred to as labeled statements. Using labeled statements, you can precisely specify the point from which the execution should resume.

The syntax to declare the labeled `break` statement is as follows:

Syntax:

`break label;`

where,

`label`: Is an identifier specified to put a name to a block. It can be any valid Java identifiers followed by a colon.

The labeled block must contain a `break` statement, but not necessarily in the immediate enclosing block. The labeled `break` statement can be used to exit from a set of nested blocks.

Code Snippet 21 demonstrates the use of labeled `break` statement.

Code Snippet 21:

```
public class TestLabeledBreak {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int i;
        outer:
        for (i = 0; i < 5; i++) {
            if (i == 2) {
                System.out.println("Hello");
                // Break out of outer loop
                break outer;
            }
            System.out.println("This is the outer loop.");
        }
        System.out.println("Good - Bye");
    }
}
```

In Code Snippet 21, the loop will execute for five times. The first two times it displays the sentence 'This is the outer loop'. In the third round of iteration the value of `i` is set to 2. Thus, it enters the `if` statement and prints 'Hello'. Next, the `break` statement is encountered and the control passes to the label named `outer`. Thus, the loop terminates and the last statement is printed.

The output of the code is shown in Figure 3.21.

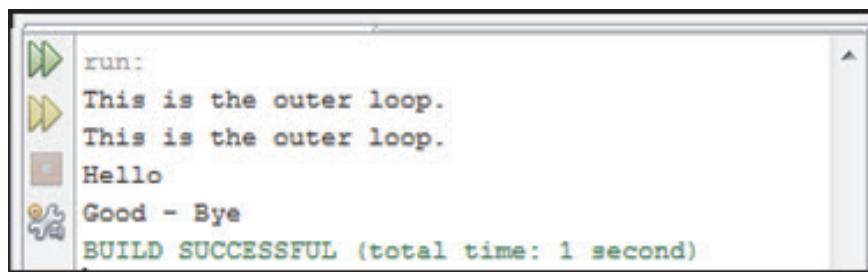


Figure 3.21: Output of Code Using Labeled `break` Statement

3.11 Check Your Progress

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)	<pre>int sum = 0; int number = 1; do { number++; sum += number; if (sum > 4) break; }while (number < 5); System.out.println(sum);</pre>	(C)	<pre>int sum = 0; int number = 0; do { number++; sum += number; if (sum > 4) break; }while (number < 5); System.out.println(sum);</pre>
(B)	<pre>int sum = 0; int number = 0; do { number++; sum += number; if (sum > 6) break; }while (number < 5); System.out.println(sum);</pre>	(D)	<pre>int sum = 0; int number = 0; do { number++; sum += number; if (sum > 4) break; }while (number == 5); System.out.println(sum);</pre>

2. 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) {

(A)	c, e, d, a, b	(C)	e, c, a, d, b
(B)	a, c, d, e, b	(D)	d, e, c, a, b

3. Which of the following statements about do-while statements are true?

a.	do-while statement 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		
(A)	a, b, and c	(C)	b, c, and d
(B)	a, b, and d	(D)	c, d, and e

4. Correct description against the loop.

	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.	continue
c.	Variable used in the expression is initialized before the loop starts	3.	for
d.	Skip statements within a loop and proceed to the next iteration of the loop	4.	do-while
(A)	a-3, b-4, c-1, d-2	(C)	a-2, b-1, c-3, d-4
(B)	a-1, b-2, c-3, d-4	(D)	a-4, b-3, c-1, d-2

5. One or more initialization or increment expressions used in for statement are separated by the _____ operator.

(A)	;	(C)	,
(B)	&&	(D)	

3.11.1 Answers

1.	B
2.	A
3.	B
4.	D
5.	C

Summary

- A Java program is a set of statements, that are executed sequentially in the order in which they appear.
- Three categories of control flow statements supported by Java programming language include: conditional, iteration, and branching statements.
- The if statement is the most basic decision-making statement that evaluates a given condition and based on result of evaluation executes a certain section of code.
- The if-else statement defines a block of statements to be executed when a condition is evaluated to false.
- The multiple if construct is known as the if-else-if ladder with conditions evaluated sequentially from the top of the ladder.
- The switch-case statement can be used as an alternative approach for multiple selections. It is used when a variable must be compared against different values. Java SE 7 and higher support strings and enumerations in the switch-case statement.
- A switch statement can also be used as a part of another switch statement. This is known as nested switch-case statements.
- Loops empower programmers to create succinct programs that would otherwise demand thousands of program statements.
- While loop is used to execute a block of statements until the specified condition is true.
- Infinite loop runs infinitely when the conditional expression of the loop is missing.
- To execute the body of the loop at least once users use the do-while loop.
- The for loop is especially used when the user knows the number of times the statements are required to be executed.
- The placing of a loop statement inside the body of another loop statement is called nesting of loops.

Try It Yourself

1. Lifemaxi is an insurance company situated in **Leeds, United Kingdom**. It has employed insurance agents that provide information to the customers about various policies and loan amount. To speed their business and also help agents to solve the customer queries, the company has decided to design a software application in Java.

You being a developer in the team have been assigned the task to develop the Java application. The application should help a loan agent to calculate whether the customer is eligible for loan or not.

The criteria that is required to be considered to automate the loan policy is as follows:

Age category	Gender	Profession	Personal assets	Loan amount eligible
16 -25	M / F	Self-Employed / Professional	>25000	10000/ Professional 15000
26 - 40	M	Self-Employed / Professional	> 40000	25000/ Female 30000
41 - 60	M / F	Self-Employed / Professional	> 50000	40000
> 60	M / F	Self-Employed / Retired	> 25000	35000 – Age * 100/ Retired 25000 – Age * 100

2. Write a program that accepts a letter as an input and checks whether the entered letter is a vowel or a consonant.
3. Write a program that accepts a deposit amount from the user and calculates the amount of interest earned in a year. The bank pays following interest rate depending on the amount deposited:
- 4% for deposits of up to 2000
 - 4.5% for deposits of up to 7000
 - 5% for deposits of more than 7000

Onlinevarsity



ON ALL DEVICES

Session - 4

Classes, Objects, and Methods

Welcome to the Session, **Classes, Objects, and Methods**.

The session explains class declaration and instantiation of objects using `new` operator. It further proceeds and explains members of a class such as instance variables, instance methods, and constructors. This session also explains creation and invocation of methods, passing and returning values to and from methods, and also use of Javadoc to lookup methods. Further, this session explains concept of access modifiers for restricting access to class members. The session covers concept of method and constructor overloading. Lastly, the session explains the use of `this` keyword in a program.

In this Session, you will learn to:

- Explain process of creation of classes in Java
- Explain instantiation of objects in Java
- Explain purpose of instance variables and instance methods
- Describe constructors and methods
- Explain memory management in Java
- Explain object initializers
- Describe access specifiers and the types of access specifiers
- Explain concept of method overloading
- Elaborate the use of `this` keyword



4.1 Introduction

The class is a logical construct that defines the shape and nature of an object. As it is the prime unit of execution for object-oriented programming in Java, so any concept in Java program must be encapsulated within the class. In Java, class is defined as a new data type. This data type is used to create objects of its type. Each object created from the class contains its own copy of the attributes defined in the class. The attributes are also referred to as fields and represents the state of an object. The initialization of objects is done using constructors and the behavior of the objects is defined using methods.

4.2 Declaring a Class

A class declaration should begin with the keyword `class` followed by the name of the class that is being declared. Besides this, following are some conventions to be followed while naming a class:

- Class name should be a noun and can be in mixed case, with the first letter of each internal word capitalized.
- Class name should be simple, descriptive, and meaningful.
- Class name cannot be Java keywords.
- Class name cannot begin with a digit. However, they can begin with a dollar (\$) symbol or an underscore character.

The syntax to declare a class in Java is as follows:

Syntax:

```
class<class_name> {
    // class body
}
```

Class declaration is enclosed within code blocks. In other words, the body of the class is enclosed between the area between the curly braces. In the class body, you can declare members, such as fields, methods, and constructors. Figure 4.1 shows the declaration of a sample class.

```
class Student {
    String studName;
    int studAge;

    void initialize()
    {
        studName = "James Anderson";
        studAge = 26;
    }

    void display()
    {
        System.out.println("Student Name: " + studName);
        System.out.println("Student Age:" + studAge);
    }

    public static void main(String[] args)
    {
        Student objStudent = new Student();
        objStudent.initialize();
        objStudent.display();
    }
}
```

The diagram highlights specific parts of the code with red boxes and arrows pointing to labels. The first two lines (`String studName;` and `int studAge;`) are grouped together with an arrow pointing to the label **Fields or Instance Variables**. The `void display()` method is highlighted with an arrow pointing to the label **Functions or Instance Methods**. The entire `public static void main(String[] args)` block is also highlighted with an arrow pointing to the same label.

Figure 4.1: A Sample Class

Code Snippet 1 shows the code for declaring a class **Customer**.

Code Snippet 1:

```
class Customer {  
    // body of class  
}
```

In the code, a class is declared that acts as a new data type. The name of the new data type is **Customer**. This data type declaration is just a template for creating multiple objects with similar features and does not occupy any memory.

4.3 Creating Objects

Objects are the actual instances of the class.

4.3.1 Declaring and Creating an Object

One of the simplest and easiest ways to create an object is created using the `new` operator. On encountering the `new` operator, JVM allocates memory for the object and returns a reference or memory address of the allocated object. The reference or memory address is then stored in a variable. This variable is also called as reference variable.

The syntax for creating an object is as follows:

Syntax:

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

where,

`new`: Is an operator that allocates the memory for an object at runtime.

`object_name`: Is the variable that stores the reference of the object.

Code Snippet 2 demonstrates the creation of an object in a Java program.

Code Snippet 2:

```
Customer objCustomer = new Customer();
```

The expression on the right side, `new Customer()` allocates the memory at runtime. After the memory is allocated for the object, it returns the reference or address of the allocated object, which is stored in the variable, `objCustomer`.

4.4 Members of a Class

The members of a class are fields and methods. Fields define the state of an object created from the class and are referred to as **instance variables**. The methods are used to implement the behavior of the objects and are referred as **instance methods**.

4.4.1 Instance Variables

The fields or variables defined within a class are called instance variables. Instance variables are used to store data in them. They are called instance variables because each instance of the class, that is, objects of that class will have its own copy of the instance variables. This means, each object of the class will contain instance variables during creation.

Consider a scenario where the **Customer** class represents the details of customers holding accounts in a bank. In this scenario, a typical question that can be asked is 'What are different data that are required to identify a customer in a banking domain and represent it as a single object?'.

Figure 4.2 shows a **Customer** object with its data requirement.

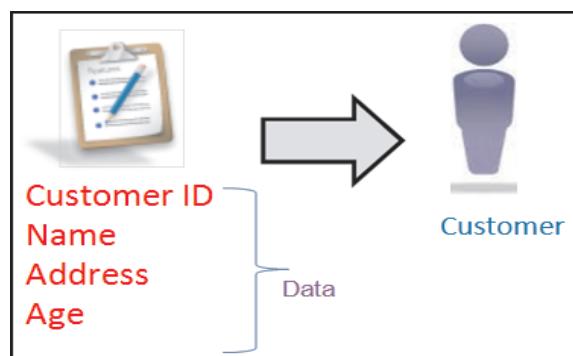


Figure 4.2: Data Requirements for Customer Object

As shown in Figure 4.2, the identified data requirements for a bank customer includes: Customer ID, Name, Address, and Age. To map these data requirements in a **Customer** class, instance variables are declared. Each instances created from the **Customer** class will have its own copy of the instance variables.

Figure 4.3 shows various instances of the class with their own copy of instance variables.

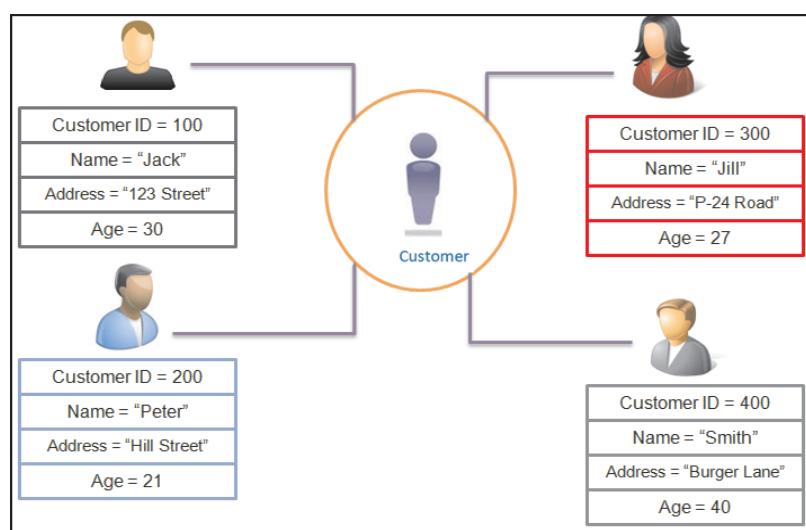


Figure 4.3: Concept of Instance Variables

As shown in Figure 4.3, each instance of the class has its own instance variables initialized with unique data. Any changes made to the instance variables of one object will not affect the instance variables of another object.

The syntax to declare an instance variable within a class is as follows:

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.

4.4.2 Instance Methods

Instance methods are functions declared in a class and are used to perform operations on the instance variables. An instance method implements the behavior of an object. It can be accessed by instantiating an object of the class in which it is defined and then, invoking the method. For example, the class **Car** can have a method **Brake ()** that represents the 'Apply Brake' action. To perform the action, the method **Brake ()** will have to be invoked by an object of class **Car**.

The instance method can access instance variables declared within the class and manipulate the data in the field.

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 4.4 shows the instance methods declared in the class, `Customer` that are invoked by all the instances of the class.

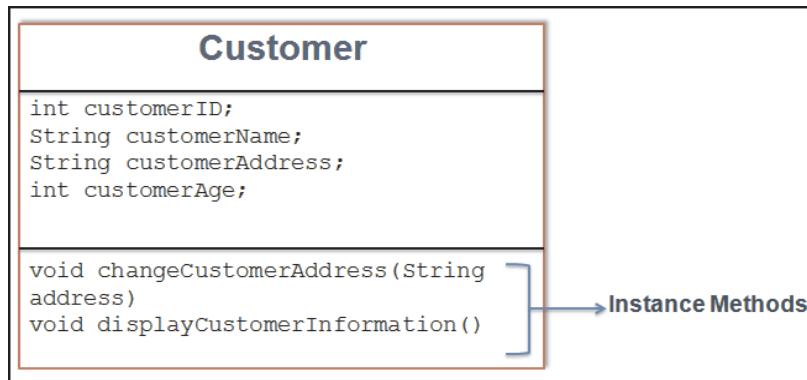


Figure 4.4: Instance Methods

The syntax to declare an instance method in a class is as follows:

Syntax:

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

where,

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

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.

4.4.3 Invoking Methods

You can access a method of a class by creating an object of the class. To invoke a method, the object name is followed by the dot operator (.) and the method name.

In Java, a method is always invoked from another method. The method which invokes a method is referred to as the **calling** method. The invoked method is referred to as the **called** method. After execution of all the statements within the code block of the invoked method, the control returns back to the **calling** method. Most of the methods are invoked from the `main()` method of the class, which is the entry point of the program execution.

Code Snippet 3 demonstrates a class with `main()` method which creates the instance of the class `Customer` and invokes the methods defined in the class.

Code Snippet 3:

```
public class TestCustomer {
    /**
     * @param args command line arguments
     * The main() method creates the instance of class Customer and invoke its methods
     */
    public static void main(String[] args) {
        // Creates an object of the class
        Customer objCustomer = new Customer();
        // Initialize the object
        objCustomer.customerID=100;
        objCustomer.customerName = "Jack";
        objCustomer.customerAddress = "123 Street";
        objCustomer.customerAge = 30;
        /*
         * Invokes the instance method to display the details of objCustomer object
         */
        objCustomer.displayCustomerInformation();
        /*
         * Invokes the instance method to change the address of the objCustomer object
         */
        objCustomer.changeCustomerAddress("123 Fort, Main Street");
        /*
         * Invokes the instance method after changing the address field
         * of objCustomer object
         */
        objCustomer.displayCustomerInformation();
    }
}
```

The code instantiates an object `objCustomer` of type `Customer` class and initializes its instance variables. The method `displayCustomerInformation()` is invoked using the object `objCustomer`. This method displays the values of the initialized instance variables on the console.

Then, the method `changeCustomerAddress("123 Fort, Main Street")` is invoked to change the data of the `customerAddress` field. Finally, the method `displayCustomerInformation()` is

invoked again to display the details of the `objCustomer` object.

The output of the code is shown in Figure 4.5.

```

run:
Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Street
Customer Age: 30

Modified Customer Details
=====
Customer Identification Number: 100
Customer Name: Jack
Customer Address: 123 Fort, Main Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 4.5: Output – Invocation of Instance Methods

4.5 Constructor

A class can contain multiple variables whose declaration and initialization becomes difficult to track if they are done within different blocks. Similarly, there may be other startup operations that are required to be performed in an application such as opening a file and so forth. Java programming language allows objects to initialize themselves immediately upon their creation. This behavior is achieved by defining constructors in the class.

A constructor is a method having the same name as that of the class. Constructors initialize the variables of a class or perform startup operations only once when the object of the class is instantiated. They are automatically executed whenever an instance of a class is created, before the `new` operator completes. Also, constructor methods do not have return types, but accepts parameters.

Figure 4.6 shows the constructor declaration.

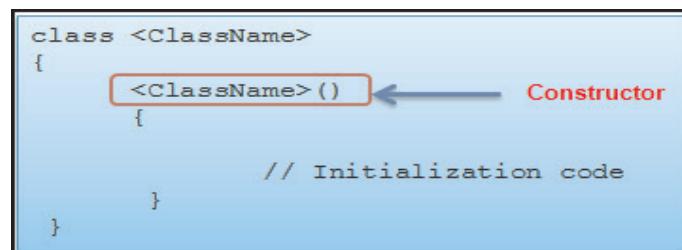


Figure 4.6: Constructor Declaration

Note - Constructors have no return type. This is because the implicit return type of a constructor is the class itself. It is also possible to have overloaded constructors in Java.

The syntax for declaring constructor in a class is as follows:

Syntax:

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

Code Snippet 4 demonstrates a class **Rectangle** with a constructor.

Code Snippet 4:

```
public class Rectangle {  
    int width;  
    int height;  
    /**  
     * Constructor for Rectangle class  
     */  
    Rectangle() {  
        width = 10;  
        height = 10;  
    }  
}
```

The code declares a method named **Rectangle()** which is a constructor. This method is invoked by JVM to initialize the two instance variables, **width** and **height**, when the object of type **Rectangle** is constructed. Also, the constructor does not have any parameters; hence, it is called as no-argument constructor.

4.5.1 Invoking Constructor

The constructor is invoked immediately during the object creation. This means that once the **new** operator is encountered, memory is allocated for the object. The constructor method, if provided in the class is invoked by the JVM to initialize the object.

Figure 4.7 shows the use of **new** operator to understand the constructor invocation.

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

Figure 4.7: Invocation of Constructor

As shown in Figure 4.7, the parenthesis after the class name indicates the invocation of the constructor.

Code Snippet 5 demonstrates the code to invoke the constructor for the class `Rectangle`.

Code Snippet 5:

```
public class TestConstructor {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiates an object of the Rectangle class
        Rectangle objRec = new Rectangle();
        // Accesses the instance variables using the object reference
        System.out.println("Width: " + objRec.width);
        System.out.println("Height: " + objRec.height);
    }
}
```

The code creates an object, `objRec` of type `Rectangle`. First, the memory allocation for the `Rectangle` object is done and then, the constructor is invoked. The constructor initializes the instance variables of the newly created object, that is, `width` and `height` to 10.

The output of the code is shown in Figure 4.8.

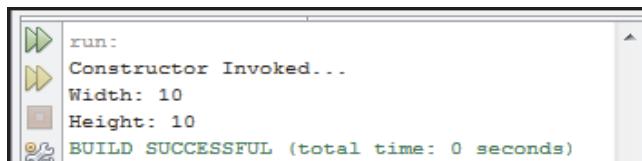


Figure 4.8: Output – Invocation of Constructor

4.5.2 Default Constructor

Consider a situation, where the constructor method is not defined for a class. In such a scenario, an implicit constructor is invoked by the JVM for initializing the objects. This implicit constructor is also known as default constructor and is created for the classes where explicit constructors are not defined. In other words, a default no-argument constructor is provided by the compiler for any class that does not have an explicit constructor. The default constructor initializes the instance variables of the newly created object to their default values.

Code Snippet 6 demonstrates a class `Employee` for which no constructor has been defined.

Code Snippet 6:

```
public class Employee {
    // Declares instance variables
    String employeeName;
```

```

int employeeAge;
double employeeSalary;
boolean maritalStatus;
/***
 * Accesses the instance variables and displays
 * their values using the println() method
 */
void displayEmployeeDetails() {
    System.out.println("Employee Details");
    System.out.println("=====");
    System.out.println("Employee Name: " + employeeName);
    System.out.println("Employee Age: " + employeeAge);
    System.out.println("Employee Salary: " + employeeSalary);
    System.out.println("Employee Marital Status: " + maritalStatus);
}
}

```

The code declares a class **Employee** with instance variables and an instance method **displayEmployeeDetails()**. The method prints the value of the instance variables on the console.

Code Snippet 7 demonstrates a class containing the **main()** method. This class creates an instance of the class **Employee** and invokes the methods of the **Employee** class.

Code Snippet 7:

```

public class TestEmployee {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiates an Employee object and initializes it
        Employee objEmp = new Employee();
        // Invokes the displayEmployeeDetails() method
        objEmp.displayEmployeeDetails();
    }
}

```

As the **Employee** class does not have any constructor defined for itself, a default constructor is created for the class at runtime.

When the statement **new Employee()** is executed, the object is allocated in memory and the instance variables are initialized to their default values by the constructor. Then, the method

`displayEmployeeDetails()` is executed which displays the values of the instance variables referenced by the object, `objEmp`.

Table 4.1 lists the default values assigned to instance variables of the class depending on their data types.

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0
char	'\u0000'
boolean	False
String (any object)	Null

Table 4.1: Default Values for Instance Variables

The output of the code is shown in Figure 4.9.

```

run:
Employee Details
=====
Employee Name: null
Employee Age: 0
Employee Salary: 0.0
Employee MaritalStatus:false
BUILD SUCCESSFUL (total time: 2 seconds)

```

Figure 4.9: Output – Default Constructor

However, if an explicit constructor is defined for the class by the developer, then the default constructor is not provided by the compiler for the class.

4.5.3 Parameterized Constructor

In the previous section, Code Snippet 5 defined the constructor for the class `Rectangle`. The constructor assigned the value 10 to the instance variables, `width` and `height`. This means all objects instantiated from the class `Rectangle` will be initialized with the same values. This may not be useful in many situations.

The parameterized constructor contains a list of parameters that initializes instance variables of an object. The value for the parameters is passed during the object creation. This means each object will be initialized with different set of values.

Code Snippet 8 demonstrates a code that declares parameterized constructor for the **Rectangle** class.

Code Snippet 8:

```
public class Rectangle {
    int width;
    int height;
    /**
     * A default constructor for Rectangle class
     */
    Rectangle() {
        System.out.println("Constructor Invoked...");
        width=10;
        height=10;
    }
    /**
     * A parameterized constructor with two parameters
     * @param wid will store the width of the rectangle
     * @param heig will store the height of the rectangle
     */
    Rectangle (int wid, int heig) {
        System.out.println("Parameterized Constructor");
        width=wid;
        height=heig;
    }
    /**
     * This method displays the dimensions of the Rectangle object
     */
    void displayDimensions () {
        System.out.println("Width: " + width);
        System.out.println("Width: " + height);
    }
}
```

The code declares a parameterized constructor, **Rectangle(int wid, int heig)**. During execution, the constructor will accept the values in two parameters and assigns them to **width** and **height** variable respectively.

Code Snippet 9 demonstrates the code with `main()` method. This class creates objects of type `Rectangle` and initializes them with parameterized constructor.

Code Snippet 9:

```
public class RectangleInstances {
    /**
     * @param args the commandline arguments
     */
    public static void main(String[] args) {
        // Declare and initialize two objects for Rectangle class
        Rectangle objRec1 = new Rectangle(10, 20);
        Rectangle objRec2 = new Rectangle(6, 9);
        // Invokes displayDimensions() method to display values
        System.out.println("\nRectangle1 Details");
        System.out.println("=====");
        objRec1.displayDimensions();
        System.out.println("\nRectangle2 Details");
        System.out.println("=====");
        objRec2.displayDimensions();
    }
}
```

The code creates two objects which invokes the parameterized constructor. For example, the statement `Rectangle objRec1 = new Rectangle(10, 20);` instantiates an object.

During instantiation, following things happen in a sequence:

1. Memory allocation is done for the new instance of the class.
2. Values 10 and 20 are passed to the parameterized constructor, `Rectangle(int wid, int heig)` which initializes the object's instance variables `width` and `height`.
3. Finally, the reference of the newly created instance is returned and stored in the object, `objRec1`.

The output of the code is shown in Figure 4.10.

```

run:
Parameterized Constructor Invoked...
Parameterized Constructor Invoked...

Rectangle1 Details
=====
Width: 10
Width: 20

Rectangle2 Details
=====
Width: 6
Width: 9
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 4.10: Output – Parameterized Constructor

Figure 4.11 displays both the instance of the class **Rectangle**. Each object contains its own copy of instance variables that are initialized through constructor.

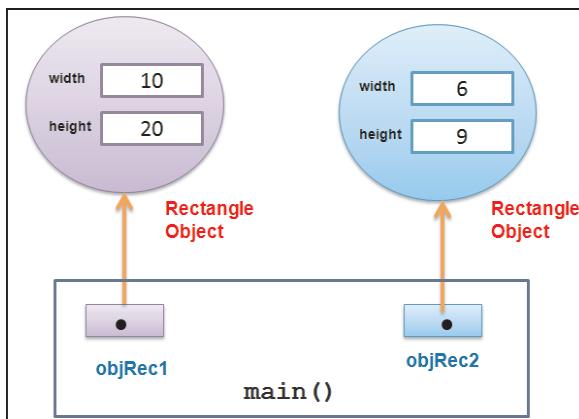


Figure 4.11: Instances of Rectangle Object

4.6 Memory Management in Java

The memory comprises two components namely, stack and heap. The stack is an area in the memory which stores object references and method information which includes: parameters of a method and its local variables. The heap area of memory deals with dynamic memory allocations. This means, Java objects are allocated physical memory space on the heap at runtime. The memory allocation is done whenever JVM executes the `new` operator.

The heap memory grows as and when the physical allocation is done for objects. Hence, JVM provides a garbage collection routine which frees the memory by destroying objects that are no longer required in Java program. This way the memory in the heap is re-used for objects allocation.

Figure 4.12 shows the memory allocation for objects in stack and heap for `Rectangle` object.

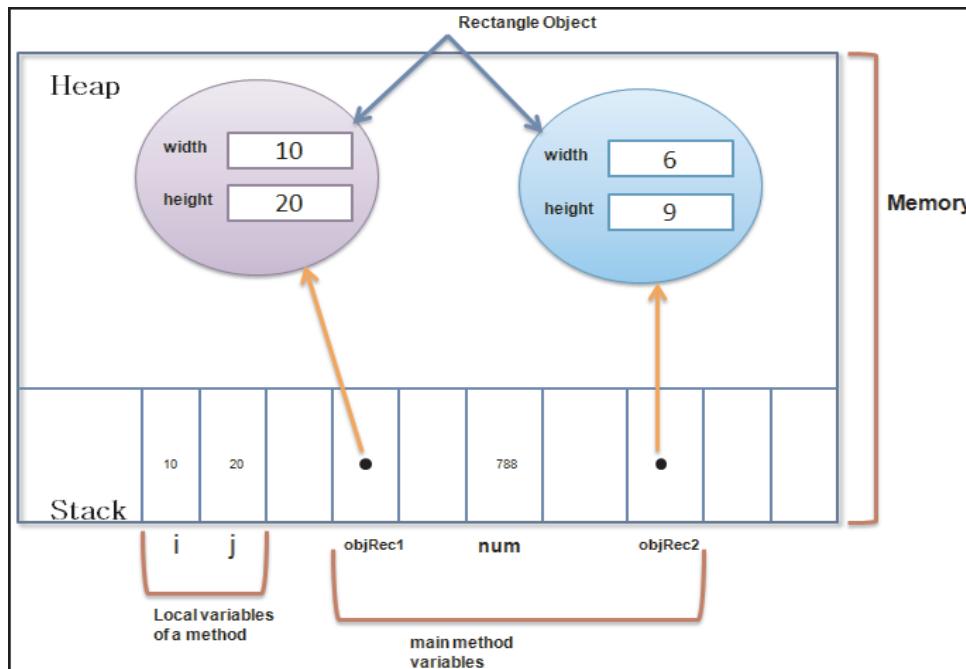


Figure 4.12: Memory Allocation for Method Variables and Objects

4.6.1 Assigning Object References

When working with primitive data types, the value of one variable can be assigned to another variable using the assignment operator.

For example,

```
int a = 10;
```

```
int b = a;
```

The statement `b = a;` copies the value from variable `a` and stores it in the variable `b`.

Figure 4.13 shows assigning of a value from one variable to another.

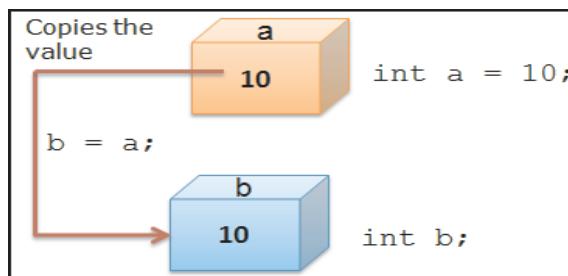


Figure 4.13: Copying of Value in Primitive Data Types

Consider the statement `Rectangle objRect1`. The `objRect1` is referred to as object reference variable as it stores the reference of an object.

As values between primitive data types can be copied, similarly the value stored in an object reference variable can be copied into another reference variable. As Java is a strongly-typed language, both the reference variables must be of same type.

In other words, both the references must belong to the same class.

Code Snippet 10 demonstrates assigning the reference of one object into another object reference variable.

Code Snippet 10:

```
public class TestObjectReferences {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        /* Instantiates an object of type Rectangle and stores its reference in the
         object reference variable, objRec1
        */
        Rectangle objRec1 = new Rectangle(10, 20);
        // Declares a reference variable of type Rectangle
        Rectangle objRec2;
        // Assigns the value of objRec1 to objRec2
        objRec2 = objRec1;
        System.out.println("\nRectangle1 Details");
        System.out.println("=====");
        /* Invokes the method that displays values of the instance variables for
         object, objRec1
        */
        objRec1.displayDimensions();
        System.out.println("\nRectangle2 Details");
        System.out.println("=====");
        objRec2.displayDimensions();
    }
}
```

The code creates two object reference variables, `objRec1` and `objRec2`. The `objRec1` points to the object that has been allocated memory and initialized to 10 and 20, whereas `objRec2` does not point to any object. For the statement, `objRec2 = objRec1;` reference stored in the `objRec1` is copied into `objRec2`, that is, the address in `objRec1` is copied into `objRec2`. Thus, the references are copied between the variables created on the stack without affecting the actual objects created on the heap.

Figure 4.14 shows the assigning of reference for the statement, `objRec2 = objRec1;`.

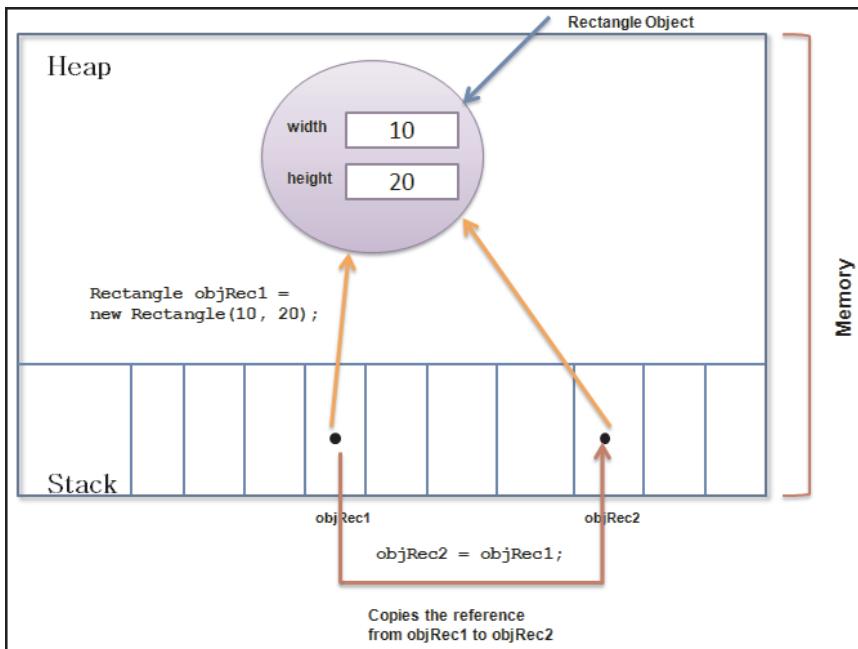


Figure 4.14: Copying of Reference on Stack

4.7 Encapsulation

Encapsulation is a mechanism which binds code and data together in a class. To understand this, consider a medicine capsule. Normally, the outer cover of a capsule is a combination of different colors which indicates that it is a combination of one or more medicines. Thus, the capsule illustrates an example of encapsulation as it hides the medicines under its cover.

Similarly, implementation details of what a class contains are not required to be visible to other classes and objects that use it. Instead, only specific information can be made visible to other components of the application and the rest can be hidden. This is achieved through encapsulation, also called data hiding. In other words, it can be said that in OOP languages, encapsulation covers the internal workings of a Java object. The main purpose of data hiding within a class is to reduce the complexity in the software development. By hiding the implementation details about what is required to implement the specific operation in the class, the usage of operation becomes simple. In Java, the data hiding is achieved by using access modifiers.

Data encapsulation hides the instance variables that represent the state of an object. Thus, the only interaction or modification is performed through methods. For example, for modifying the account holder name, the appropriate method such as `changeHolderName()` can be provided in the `Account` class. This way the implementation of the class is hidden from the rest of the application.

4.7.1 Access Modifiers

Access modifiers determine how members of a class, such as instance variable and methods are accessible from outside the class. In other words, they define the 'visibility' of the members. They are used as prefix

while declaring the members of a class. The scope or visibility of the members is also closely related to the packages. Usually, access modifiers are used wherever possible in order to ensure encapsulation and restricting access to class members.

Note: Sometimes, access modifiers are also referred to as access modifiers. Though both terms mean the same, it is recommended to use access modifiers as the formal term.

Using access modifiers provides following advantages:

- Access modifiers help to control the access of classes and class members.
- Access modifiers help to prevent misuse of class details as well as hide the implementation details that are not required by other classes.
- Access modifiers also determine whether classes and the members of the classes can be invoked by other classes or interfaces.
- Accessibility affects inheritance and how members are inherited by the subclass.
- A package is always accessible by default.

Java provides four types of access modifiers that are as follows:

→ **Public**

The `public` access modifier is the least restrictive of all access modifiers. A field, method, or class declared `public` is visible to any class in a Java application in the same package or in another package. Members declared as `public` can be accessed from anywhere in the class as well as from other classes.

→ **Private**

The `private` access modifier is the most restrictive of all access modifiers. The `private` access modifier cannot be used for classes and interfaces as well as fields and methods of an interface. Fields and methods declared `private` cannot be accessed from outside the enclosing class. A standard convention is to declare all fields `private` and provide `public` accessor or getter methods to access them. Thus, when data is important, sensitive, and cannot be shared with others, it is declared as `private`.

→ **Protected**

The `protected` access modifier is used with classes that share a parent-child relationship which is referred to as inheritance. The `protected` keyword cannot be used for classes and interfaces as well as fields and methods of an interface. Fields and methods declared `protected` in a parent or super class can be accessed only by its child or subclass in another packages. The `protected` access modifier allows the class members to be accessible from within the class as well as from within the derived classes. Derived classes are sub classes created on the basis of existing classes or interfaces. However, classes in the same package can also access protected fields and methods,

even if they are not a subclass of the protected member's class.

→ Package (Default)

The package access modifier allows only the public members of a class to be accessible to all the classes present within the same package. This is the default access level for all the members of the class. The package or default access modifier is used when no access modifier is present. This modifier is applied to any class, field, or method for which no access modifier has been mentioned. With this modifier, the class, field, or method is accessible only to the classes of the same package. This modifier cannot be used for fields and methods within an interface.

As a general rule in Java, the details and implementation of a class is hidden from the other classes or external objects in the application. This is done by making instance variables as private and instance methods as public.

Figure 4.15 shows various access modifiers.

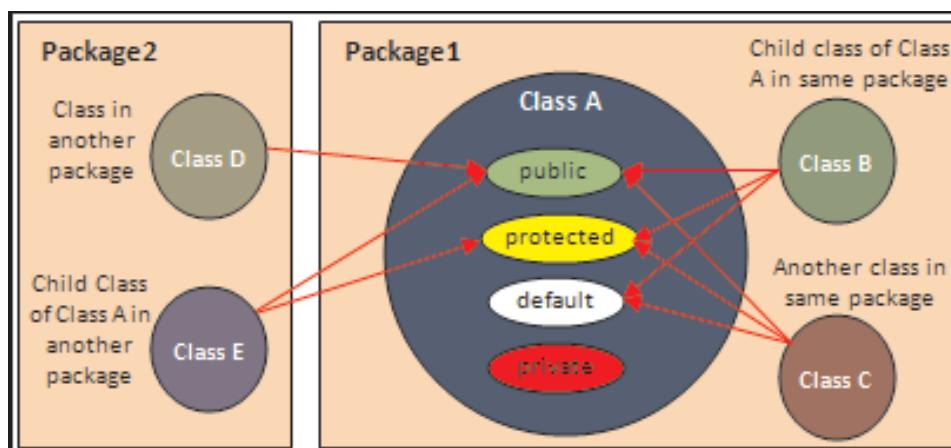


Figure 4.15: Using Access Modifiers

Figure 4.15 shows class **A** with four data members having public, protected, default, and private access modifiers. Class **B** is a child class of **A** and class **C** is another class belonging to the same package, **Package1**. From Figure 4.15, it is clear that both classes **B** and **C** have access to the public, protected, and default access members since they belong to the same package.

The package, **Package2** consists of classes **D** and **E**. Class **D** can only access the `public` members of class **A**. However, class **E** can access `public` as well as `protected` members of class **A** even though it belongs to another package. This is because class **E** is a child class of **A**. However, none of the classes **B**, **C**, **D**, or **E** can access the `private` members of class **A**.

Table 4.2 shows the access level for different access modifiers.

Access Modifiers	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
No modifier (default)	Y	Y	N	N

Access Modifiers	Class	Package	Subclass	World
private	Y	N	N	N

Table 4.2: Access Levels of Access Modifiers

The first column states whether the class itself has access to its own data members. As can be seen, a class can always access its own members. The second column states whether classes within the same package as the owner class (irrespective of their parentage) can access the member. As can be seen, all members can be accessed except `private` members.

The third column states whether the subclasses of a class declared outside this package can access a member. In such cases, `public` and `protected` members can be accessed. The fourth column states whether all classes can access a data member.

4.7.2 Rules for Access Control

Java has rules and constraints for usage of access modifiers as follows:

- While declaring members, a `private` access modifier cannot be used with `abstract`, but it can be used with `final` or `static`.
- No access 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.
- `Private` cannot be used with fields and methods of an interface.
- The most restrictive access level must be used that is appropriate for a particular member.
- Mostly, a `private` access modifier is used at all times unless there is a valid reason for not using it.
- Avoid using `public` for fields except for constants.

4.8 Object Initializers

Object initializers in Java provide a way to create an object and initialize its fields. In a normal approach, you invoke a constructor to initialize objects, but using object initializers you can complement the use of constructors.

There are two approaches to initialize the fields or instance variables of the newly created objects.

→ Using Instance Variable Initializers

In this approach, you specify the names of the fields and/or properties to be initialized, and give an initial value to each of them.

Code Snippet 11 demonstrates a Java program that declares a class, **Person** and initializes its fields.

Code Snippet 11:

```
public class Person {
    private String name = "John";
    private int age = 12;
    /**
     * Displays the details of Person object
     */
    void displayDetails() {
        System.out.println("Person Details");
        System.out.println("=====");
        System.out.println("Person Name: " + name);
        System.out.println("Person Age: " + age);
    }
}
```

In the code, the instance variables **name** and **age** are initialized to values '**John**' and 12 respectively. Initializing the variables within the class declaration does not require them to initialize in a constructor.

Code Snippet 12 shows the class with `main()` method that creates objects of type **Person**.

Code Snippet 12:

```
public class TestPerson {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Person objPerson1 = new Person();
        objPerson1.displayDetails();
    }
}
```

The code creates an object of type **Person** and invokes the method to display the details. The output of the code is shown in Figure 4.16.

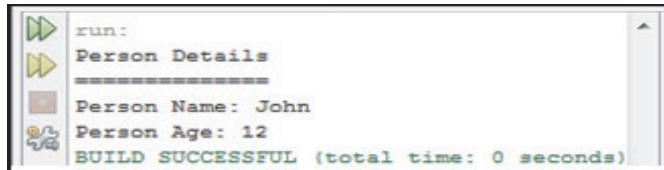


Figure 4.16: Output – Instance Variable Initializers

→ Using Initialization Block

In this approach, an initialization block is specified within the class. The initialization block is executed before the execution of constructors during an object initialization.

Code Snippet 13 demonstrates the class **Account** with an initialization block.

Code Snippet 13:

```
public class Account {
    private int accountID;
    private String holderName;
    private String accountType;
    /**
     * Initialization block
     */
    {
        accountID=100;
        holderName = "John Anderson";
        accountType = "Savings";
    }
    /**
     * Displays the details of Account object
     */
    public void displayAccountDetails() {
        System.out.println("Account Details");
        System.out.println("=====");
        System.out.println("Account ID: " + accountID + "\nAccount Type: " +
                           accountType);
    }
}
```

In the code, the initialization blocks initializes the instance variables or fields of the class. The initialization blocks are basically used to perform complex initialization sequences.

Code Snippet 14 shows the code with `main()` method to initialize the `Account` object through initialization block.

Code Snippet 14:

```
public class TestInitializationBlock {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Account objAccount = new Account();
        objAccount.displayAccountDetails();
    }
}
```

The output of the code is shown in Figure 4.17.

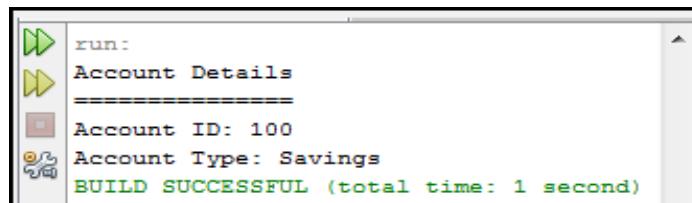


Figure 4.17: Output - Initialization Block

4.9 Passing and Returning Values from Methods

Parameters are the list of variables specified in a method declaration, whereas arguments are the actual values that are passed to the method when it is invoked. When a method is invoked, the type and order of arguments that are passed must match the type and order of parameters declared in the method.

A method can accept value of any data type as a parameter. A method can accept primitive data types such as `int`, `float`, `double`, and so on as well as reference data types such as arrays and objects as a parameter. In other words, arguments can be passed by value or by reference as follows:

→ Passing Arguments by Value

When arguments are passed by value it is known as call-by-value and it means that:

- A copy of the argument is passed from the calling method to the called method.
- Changes made to the argument passed 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.

Code Snippet 15 demonstrates an example of passing arguments by value.

Code Snippet 15:

```
package session4;

public class PassByValue {

    // method accepting the argument by value
    public void setVal(int num1) {
        num1 = num1 + 10;
    }

    public static void main(String[] args) {
        // Declare and initialize a local variable
        int num1 = 10;
        // Instantiate the PassByValue class
        PassByValue obj = new PassByValue();
        // Invoke the setVal() method with num1 as parameter
        obj.setVal(num1);
        // Print num1 to check its value
        System.out.println("Value of num1 after invoking setVal is "+
        num1);
    }
}
```

Figure 4.18 shows the output of the code.

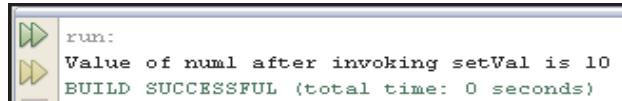


Figure 4.18: Output After Passing Arguments by Value

The class **PassByValue** consists of one method **setVal()** that accepts an integer value as parameter. The **main()** method declares an integer variable named **num1** and initializes it to 10.

Next, an object of **PassByValue** class is created to invoke the **setVal()** method with **num1** as argument. The **setVal()** method increments the value of **num1** by 10. After processing the value, **main()** method prints the value of **num1** to verify if it has changed. However, the output shows that the value of **num1** is still 10 even after invoking **setVal()** method where the value had been incremented.

This is because, **num1** was passed by value. That is, only a copy of the value of **num1** was passed to **setVal()** method and not the actual address of **num1** in memory. So that, the value of **num1** remains unchanged even after invoking the method **setVal()**.

→ Passing Arguments by Reference

When arguments are passed by reference it means that:

- The actual memory location of the argument is passed to the called method and the object or a copy of the object is not passed.
- The called method can change the value of the argument passed to it.
- Variables of reference types such as objects are passed to the methods by reference.

- There are two references of the same object namely, argument reference variable and parameter reference variable.

Code Snippet 16 demonstrates an example of passing arguments by reference.

Code Snippet 16:

```
package session4;
class Circle{
    // Method to retrieve value of PI
    public double getPI(){
        return 3.14;
    }
}
// Define another class PassByRef
public class PassByRef{
    // Method to calculate area of a circle that takes object of class
    // Circle as a parameter
    public void calcArea(Circle objPi, double rad){
        // Use getPI() method to retrieve the value of PI
        double area= objPi.getPI() * rad * rad;
        // Print the value of area of circle
        System.out.println("Area of the circle is "+ area);
    }
    public static void main(String[] args){
        // Instantiate the PassByRef class
        PassByRef p1 = new PassByRef();
        // Invoke the calcArea() method with object of class Circle as
        // a parameter
        p1.calcArea(new Circle(), 2);
    }
}
```

Figure 4.19 shows the output of the code.

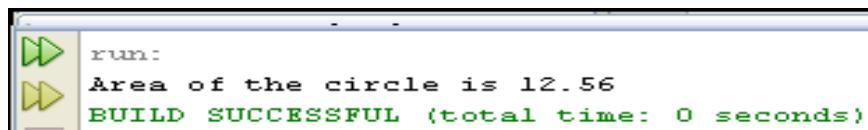


Figure 4.19: Output After Passing Arguments by Reference

The class **Circle** consists of a method named **getPI()** with return type set to **double** and the method returns the value **3.14** to the calling method. Another class named **PassByRef** consists of a method **calcArea()** that accepts the object **objPi** of class **Circle** and the radius as a parameter and calculates the area of the circle. The value of PI is retrieved by invoking the method **objPi.getPI()** that returns the value **3.14**. Next, the method prints the value of **area**.

Within the **main()** method, the **PassByRef** class is instantiated and an object of class **Circle** is passed as a parameter by the statement '**new Circle()**' and the value **2** is passed as the radius. When an object is passed to a method, it is always passed as reference. In other words, a reference of the object

is created and passed. Thus, when this reference is passed, the method that receives it will refer to the same object that is referred by the argument. After execution, the output shows the area of the circle as **12.54**. Note that the value of PI is passed by reference and not by value.

→ Returning Values from Methods

A method will return a value to the invoking method only when all the statements in the invoking method are complete or when it encounters a return statement, or when an exception is thrown. The return statement is written within the body of the method to return a value. A `void` method will not have a return type specified in its method body. A compiler error is generated when a `void` method returns a value.

The method `getPI()` in class `Circle` in Code Snippet 16 has the return type set to `double`.

This means that the method must return a value of type `double` to the calling method. The keyword `return` is used for this purpose. The statement returns **3.14** as the value of PI to the calling method. Instead of using **3.14**, one can also store the value in a variable and specify the name of the variable with the `return` keyword. For example, the class `Circle` and its `getPI()` method can be modified as shown in Code Snippet 17.

Code Snippet 17:

```
public class Circle {
    // Declare and initialize value of PI
    private double PI = 3.14;
    // Method to retrieve value of PI
    public double getPI() {
        return PI;
    }
}
```

In the modified class `Circle`, the value **3.14** is stored in a `private double` variable `PI`. Later, the method `getPI()` returns the value stored in the variable `PI` instead of the constant value **3.14**.

Note - The variable `PI` is declared `private` to restrict direct access to it by any object. Instead the `getPI()` method is used to access it. Therefore, the `getPI()` method becomes the accessor method for `PI`. Similarly, one can create a mutator method such as `setPI()` to modify the value of `PI`.

4.10 Declaring Variable Argument Methods

Java provides a feature called `varargs` to pass variable number of arguments to a method. `varargs` is used when the number of a particular type of argument that will be passed to a method is not known until runtime. It serves as a shortcut to creating an array manually.

To use `varargs`, the type of the last parameter is followed by ellipsis (`...`), then, a space, followed by the name of the parameter. This method can be called with any number of values for that parameter, including none.

The syntax of a variable argument method is as follows:

Syntax:

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

where,

'...': Indicates the variable number of arguments.

Code Snippet 18 demonstrates an example of a variable argument method.

Code Snippet 18:

```
package session4;
public class Varargs {
    // Variable argument method taking variable number of integer
    // arguments
    public void addNumber(int...num) {
        int sum=0;
        // Use for loop to iterate through num
        for(int i:num) {
            // Add up the values
            sum = sum + i;
        }
        System.out.println("Sum of numbers is "+sum);
    }
    public static void main(String[] args) {
        // Instantiate the Varargs class
        Varargs obj = new Varargs();
        // Invoke the addNumber() method with multiple arguments
        obj.addNumber(10,30,20,40);
    }
}
```

Figure 4.20 shows the output of the code.

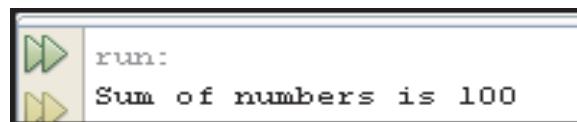


Figure 4.20: Output After Using Variable Argument Method

The class **Varargs** consists of a method called **addNumber(int...num)**. The method accepts variable number of arguments of type integer. The method uses the enhanced **for** loop to iterate through the variable argument parameter **num** and adds each value with the variable **sum**. Finally, the method prints the value of **sum**.

The **main()** method creates an object of the class and invokes the **addNumber()** method with multiple

arguments of type integer. The output displays 100 after adding up the numbers.

4.11 Method Overloading

Consider the class `Calculator` created earlier. The class has different methods for different operations. However, all the methods add only integers. What if a user wants to add numbers of different types such as two floating-point numbers or one integer and other floating-point number? One solution is to create a method with different names such as `addFloat()` or `addIntFloat()` for this purpose. However, this would be very tedious and unnecessary since the function of the two methods is still addition.

It would be more convenient to have a way to create different variations of the same `add()` method to add different types of values. The Java programming language provides the feature of method overloading to distinguish between methods with different method signatures. Using method overloading, multiple methods of a class can have the same name but with different parameter lists.

Method overloading can be implemented in following three ways:

1. Changing the number of parameters
2. Changing the sequence of parameters
3. Changing the type of parameters

4.11.1 Overloading with Different Parameter List

Methods can be overloaded by changing the number or sequence of parameters of a method. Figure 4.21 shows an example of overloaded `add()` methods.

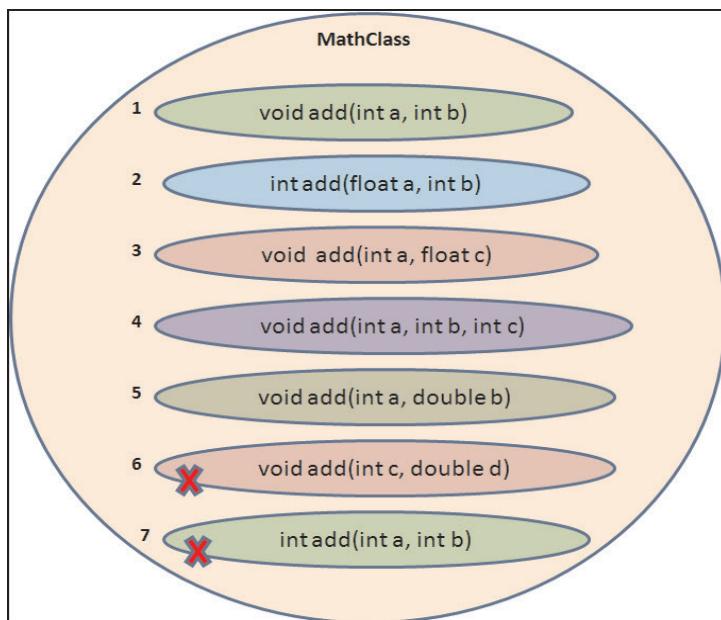


Figure 4.21: Overloading Methods

Figure 4.21 shows seven `add()` methods each with a different signature. The `add()` methods 1 and 4 both accept integers as parameter. However, they are different in the number of arguments they accept.

Similarly, `add()` methods numbered 2 and 3 accept `int` and `float` as parameters. However, they differ in the sequence in which they accept `int` and `float`. Figure 4.21 also shows other variations of `add()` methods.

4.11.2 Overloading with Different Data Types

Methods can be overloaded by changing the data type of parameters of a method. Figure 4.21 shows an example of `add()` method overloaded by changing the type of parameters. Each of the `add()` methods numbered 1, 2, 3, and 5 accepts two parameters. However, they differ in the type of parameters that they accept.

Notice that the `add()` method numbered 6 is similar to the method numbered 5. Both the methods accept first an integer argument and then, a float argument as a parameter. However, the parameter names are different. This is not sufficient to make a method overloaded. The methods must differ in argument type and number and not simply in name of arguments.

Similarly, the `add()` method numbered 7 has a signature similar to the method numbered 1. Both the method accepts two integers, `a` and `b` as parameters. However, the return type of method numbered 7 is `int` whereas that of method numbered 1 is `void`. Again, this does not make the method overloaded as it does not differ in argument type and number.

Thus, the `add()` methods numbered 6 and 7 are not overloaded methods and will lead to compilation error. Changing only the names of parameters or return type of method does not make it overloaded. Also, a method cannot be considered as overloaded if only the access modifiers are different.

Code Snippet 19 demonstrates an example of method overloading.

Code Snippet 19:

```
package session4;
public class MathClass {
    /**
     * Method to add two integers
     *
     * @param num1 an integer variable storing the value of first number
     * @param num2 an integer variable storing the value of second number
     * @return void
     */
    public void add(int num1, int num2) {
        System.out.println("Result after addition is "+ (num1+num2));
    }
    /**
     * Overloaded method to add three integers
     *
     * @param num1 an integer variable storing the value of first number
     * @param num2 an integer variable storing the value of second number
     * @param num3 an integer variable storing the value of third number
     * @return void
     */
    public void add(int num1, int num2, int num3) {
```

```

        System.out.println("Result after addition is "+ (num1+num2+num3));
    }
    /**
     * Overloaded method to add a float and an integer
     *
     * @param num1 a float variable storing the value of first number
     * @param num2 an integer variable storing the value of second number
     * @return void
     */
    public void add(float num1, int num2) {
        System.out.println("Result after addition is "+ (num1+num2));
    }
    /**
     * Overloaded method to add a float and an integer accepting the values
     * in a different sequence
     *
     * @param num1 an integer variable storing the value of first number
     * @param num2 a float variable storing the value of second number
     * @return void
     */
    public void add(int num1, float num2) {
        System.out.println("Result after addition is "+ (num1+num2));
    }
    /**
     * Overloaded method to add two floating-point numbers
     *
     * @param num1 a float variable storing the value of first number
     * @param num2 a float variable storing the value of second number
     * @return void
     */
    public void add(float num1, float num2) {
        System.out.println("Result after addition is "+ (num1+num2));
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the MathClass class
        MathClass objMath = new MathClass();
        // Invoke the overloaded methods with relevant arguments
        objMath.add(3.4F, 2);
        objMath.add(4,5);
        objMath.add(6,7,8);
    }
}

```

Figure 4.22 shows the output of the code.

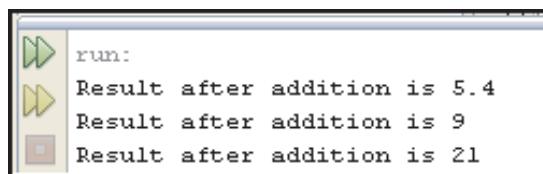


Figure 4.22: Output of Overloaded Methods

Figure 4.22 shows a class named **MathClass** consisting of overloaded **add()** methods. The **main()** method creates an object of **MathClass** and invokes the **add()** methods with different types and number of arguments. The compiler executes the appropriate **add()** method based on the type and number of arguments passed by the user. The output displays the result of addition of different values.

4.11.3 Constructor Overloading

Constructor is a special method of a class that has the same name as the class name. A constructor is used to initialize the variables of a class. Similar to a method, a constructor can also be overloaded to initialize different types and number of parameters. When the class is instantiated, the compiler will invoke the constructor based on the number, type, and sequence of arguments passed to it. Code Snippet 20 demonstrates an example of constructor overloading.

Code Snippet 20:

```
package session4;
public class Student {
    int rollNo; // Variable to store roll number
    String name; // Variable to store student name
    String address; // Variable to store address
    float marks; // Variable to store marks
    /**
     * No-argument constructor
     */
    public Student(){
        rollNo = 0;
        name = "";
        address = "";
        marks = 0;
    }
    /**
     * Overloaded constructor
     * @param rNo an integer variable storing the roll number
     * @param name a String variable storing student name
     */
    public Student(int rNo, String sname) {
        rollNo = rNo;
        name = sname;
    }
}
```

```

* Overloaded constructor
* @param rNo an integer variable storing the roll number
* @param score a float variable storing the score
*/
public Student(int rNo, float score) {
    rollNo = rNo;
    marks = score;
}
/**
* Overloaded constructor
* @param sName a String variable storing student name
* @param addr a String variable storing the address
*/
public Student(String sName, String addr) {
    name = sName;
    address = addr;
}
/**
* Overloaded constructor
*
* @param rNo an integer variable storing the roll number
* @param sName a String variable storing student name
* @param score a float variable storing the score
*/
public Student(int rNo, String sname, float score) {
    rollNo = rNo;
    name = sname;
    marks = score;
}
/**
* Displays student details
*
* @return void
*/
public void displayDetails() {
    System.out.println("Rollno :" + rollNo);
    System.out.println("Student name:" + name);
    System.out.println("Address " + address);
    System.out.println("Score " + marks);
    System.out.println("-----");
}
/**
* @param args the command line arguments
*/

```

```

public static void main(String[] args) {
    // Instantiate the Student class with two string arguments
    Student objStud1 = new Student("David", "302, Washington Street");
    // Invoke the displayDetails() method
    objStud1.displayDetails();
    // Create other Student class objects and pass different
    // parameters to the constructor
    Student objStud2 = new Student(103, 46);
    objStud2.displayDetails();
    Student objStud3 = new Student(104, "Roger", 50);
    objStud3.displayDetails();
}
}

```

The class **Student** consists of member variables named **rollNo**, **name**, **address**, and **marks**. **Student()** is the default or no-argument constructor of the **Student** class. The other constructors are overloaded constructors created by changing the number and type of parameters. The **main()** method creates three objects **objStud1**, **objStud2**, and **objStud3** of **Student** class. Each object passes different arguments to the constructor. Later, each object invokes the **displayDetails()** method to print the student details.

Figure 4.23 shows the output of the program.

```

run:
Rollno :0
Student name:David
Address 302, Washington Street
Score 0.0
-----
Rollno :103
Student name:null
Address null
Score 46.0
-----
Rollno :104
Student name:Roger
Address null
Score 50.0
-----
```

Figure 4.23: Output of Constructor Overloading

Figure 4.23 shows the output generated by **displayDetails()** method for objects **objStud1**, **objStud2**, and **objStud3** of **Student** class based on the constructor invoked by these objects. Notice the values **0** and **null** for the variables for which no argument was specified. These are the default values for integer and **String** data types in Java.

4.11.4 Using this Keyword

Java provides the keyword **this** which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being called. Any member of the

current object can be referred from within an instance method or a constructor by using the 'this' keyword. The keyword `this` is not explicitly used in instance methods while referring to variables and methods of a class.

For example, consider the method `calcArea()` of Code Snippet 21.

Code Snippet 21:

```
public class Circle {
    float area; // variable to store area of a circle
    /**
     * Returns the value of PI
     *
     * @return float
     */
    public float getPI(){
        return 3.14f;
    }

    /**
     * Calculates area of a circle
     * @param rad an integer to store the radius
     * @return void
     */
    public void calcArea(int rad) {
        this.area = getPI() * rad * rad;
    }
}
```

The method `calcArea()` calculates the area of a circle and stores it in the variable, `area`. It retrieves the value of PI by invoking the `getPI()` method. Here, the method call does not involve any object even though `getPI()` is an instance method. This is because of the implicit use of 'this' keyword.

For example, the method `calcArea()` can also be written as shown in Code Snippet 22.

Code Snippet 22:

```
public class Circle {
    float area; // Variable to store area of a circle
    /**
     * Returns the value of PI
     *
     * @return float
     */
    public float getPI(){
        return 3.14f;
    }

    /**
     * Calculates area of a circle
     * @param rad an integer to store the radius
     * @return void
     */
    public void calcArea(int rad) {
        this.area = getPI() * rad * rad;
    }
}
```

```

    * @param rad an integer to store the radius
    * @return void
    */
public void calcArea(int rad) {
    this.area = this.getPI() * rad * rad;
}
}

```

Notice the use of `this` to indicate the current object. The keyword `this` can also be used to invoke a constructor from within another constructor. This is also known as explicit constructor invocation as shown in Code Snippet 23.

Code Snippet 23:

```

public class Circle {
    private float rad; // Variable to store radius of a circle
    private float PI; // Variable to store value of PI
    /**
     * No-argument constructor
     *
     */
    public Circle() {
        PI = 3.14f;
    }
    /**
     * Overloaded constructor
     *
     * @param r a float variable to store the value of radius
     */
    public Circle(float r) {
        this(); // Invoke the no-argument constructor
        rad = r;
    }
    ...
}

```

The keyword `this` can be used to resolve naming conflicts when the names of actual and formal parameters of a method or a constructor are the same as depicted in Code Snippet 24.

Code Snippet 24:

```
public class Circle {
    // Variable to store radius of a circle
    private float rad; // line 1
    private float PI; // Variable to store value of PI
    /**
     * no-argument constructor
     *
     */
    public Circle() {
        PI = 3.14f;
    }
    /**
     * overloaded constructor
     *
     * @param rad a float variable to store the value of radius
     */
    public Circle(float rad) { // line2
        this();
        this.rad = rad; // line3
    }
    ...
}
```

Code Snippet 24 defines the constructor `Circle` with the parameter `rad` in line2 which is the formal parameter. Also, the parameter declared in line1 has the same name `rad` which is the actual parameter to which the user's value will be assigned at runtime. Now, while assigning a value to `rad` in the constructor, the user would have to write `rad = rad`. However, this would confuse the compiler as to which `rad` is the actual and which one is the formal parameter. To resolve this conflict, `this.rad` is written on the left of the assignment operator to indicate that it is the actual parameter to which value must be assigned.

4.12 Check Your Progress

1. Which of the following statements stating the characteristics of classes are true?

a.	Class names cannot be a keyword in Java		
b.	Class names can be in mixed case		
c.	Declaration of the class are not required to be preceded with the keyword <code>class</code>		
d.	Class names can begin with a digit		

(A)	a and c	(C)	a and b
(B)	c and d	(D)	All of these

2. Which of the following methods have the same name as class name?

(A)	Main	(C)	Instance Method
(B)	Constructor	(D)	Destructor

4. In Java, _____ provides a way to create an object and initialize its fields, before the constructor methods are invoked.

(A)	Instance variables	(C)	Instance Methods
(B)	Class Initializers	(D)	Object Initializers

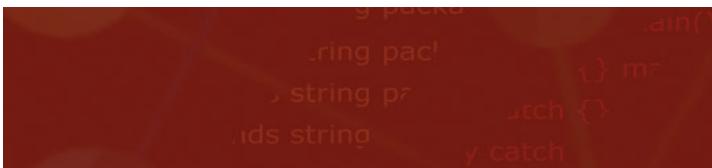
5. Which of the following options specifies a feature of OOP language that hides the instance variables within the class?

(A)	Abstraction	(C)	Data Encapsulation
(B)	Inheritance	(D)	Polymorphism

4.12.1 Answers

1.	C
2.	B
3.	D
4.	C
5.	D

Summary



- The class is a logical construct that defines the shape and nature of an object.
- Objects are the actual instances of the class and are created using the new operator. The new operator instructs JVM to allocate the memory for the object.
- The members of a class are fields and methods. Fields define the state and are referred to as instance variables, whereas methods are used to implement the behavior of the objects and are referred to as instance methods.
- Each instance created from the class will have its own copy of the instance variables, whereas methods are common for all instances of the class.
- Constructors are methods that are used to initialize the fields or perform startup operations only once when the object of the class is instantiated.
- Data encapsulation hides the instance variables that represents the state of an object through access modifiers. The only interaction or modification on objects is performed through the methods.
- A Java method is a set of statements grouped together for performing a specific operation.
- Parameters are the list of variables specified in a method declaration, whereas arguments are the actual values that are passed to the method when it is invoked.
- The variable argument feature is used in Java when the number of a particular type of arguments that will be passed to a method is not known until runtime.
- Java comes with four access modifiers namely, public, private, protected, and default.
- Using method overloading, multiple methods of a class can have the same name, but with different parameter lists.
- Java provides the `this` keyword which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being invoked.

Try It Yourself

- Consider a situation where you have been asked to develop a paint application. The paint application should be able to draw different types of shapes, such as circle, square, parallelogram, rectangle, and so on. Each of these shapes is a collection of many points. Thus, you decide to create a **Point** class in the application. Features to be implemented in the **Point** class are as follows:
 - As a point is formed with two co-ordinates on the plane, include **x** and **y** as the fields in the **Point** class.
 - Implement the methods, such as **setX()**, **setY()**, and **displayPoints()** in the **Point** class to represent the behavior of the **Point** class.
 - Implement necessary constructors to initialize the objects of the **Point** class during their creation.
 - Create two objects of **Point** type and compare their **x** and **y** co-ordinates. If co-ordinates have same values, then, display 'Points are Same', otherwise, display 'Points are Different'.
- What will be the output when you compile and run the following code:

```
public class MyClass {
    public static void main(String arguments[]) {
        someMethod(arguments);
    }
    public void someMethod(String[] parameters) {
        System.out.println(parameters);
    }
}
```

- Phoenix Systems** is a well known computer hardware store located in **L.A., California**. The manager of the store wishes to develop a software through which he can store and view data about the hardware devices that he sells. The manager has hired a programmer to develop the software. The programmer has written the following class to store and view details about the devices.

```
public class DeviceDetails {
    int deviceNo;
    String deviceName, deviceType;
    double devicePrice;
    public DeviceDetails() {
```

Try It Yourself

```

    package com.aptech;
    import java.util.*;
    public class DeviceDetails {
        int deviceNo;
        String deviceName;
        String deviceType;
        double devicePrice;

        public DeviceDetails() {
            deviceNo=0;
            deviceName="";
            deviceType="";
            devicePrice=0.0;
        }

        public DeviceDetails(int deviceNo, String deviceType) {
            deviceNo=deviceNo;
            deviceType=deviceType;
        }

        public void displayDetails() {
            System.out.println("Device number is "+deviceNo);
            System.out.println("Device name is "+deviceName);
            System.out.println("Device type is "+deviceType);
            System.out.println("Device price is "+devicePrice);
        }

        public static void main(String[] args) {
            DeviceDetails objDevice = new DeviceDetails();
            objDevice.displayDetails();
        }
    }
}

```

The program is giving compilation errors and not functioning as expected. Modify the program as follows:

- The user should be able to specify all details about a device at a time.
- The user should be able to specify only `deviceNo` and `devicePrice` when required.
- The variables should not be accessible outside the class.
- The program should display all details of a device properly even when some details are not provided.
- Generate javadoc for the class to view the details of methods of the class.



**MANY
COURSES
ONE
PLATFORM**

Session - 5

Arrays and Strings

Welcome to the Session, **Arrays and Strings**.

This session explains the creation and use of arrays in Java. Further, this session explains accessing values from an ArrayList using loops. The session also describes the working of String and StringBuilder classes. Lastly, the session describes Wrapper classes and the concept of autoboxing and unboxing.

In this Session, you will learn to:

- Describe an array
- Explain declaration, initialization, and instantiation of a single-dimensional array
- Explain declaration, initialization, and instantiation of a multi-dimensional array
- Explain the use of loops to process an array
- Describe ArrayList and accessing values from an ArrayList
- Describe String and StringBuilder classes
- Describe Wrapper classes, autoboxing, and unboxing



5.1 Introduction

Consider a situation where a user wants to store marks of ten students. For this purpose, the user can create ten different variables of type integer and store the marks in them. What if the user wants to store marks of hundreds or thousands of students? In such a case, one would require to create as many variables. This can be a very difficult, tedious, and time consuming task. Here, it is required to have a feature that will enable storing of all the marks in one location and access it with similar variable names. Array, in Java, is a feature that allows storing multiple values of similar type in the same variable.

5.2 Introduction to Arrays

An array is a special data store that can hold a fixed number of values of a single type in contiguous memory locations. It is implemented as objects. The size of an array depends on the number of values it can store and is specified when the array is created. After creation of an array, its size or length becomes fixed. Figure 5.1 shows an array of numbers.

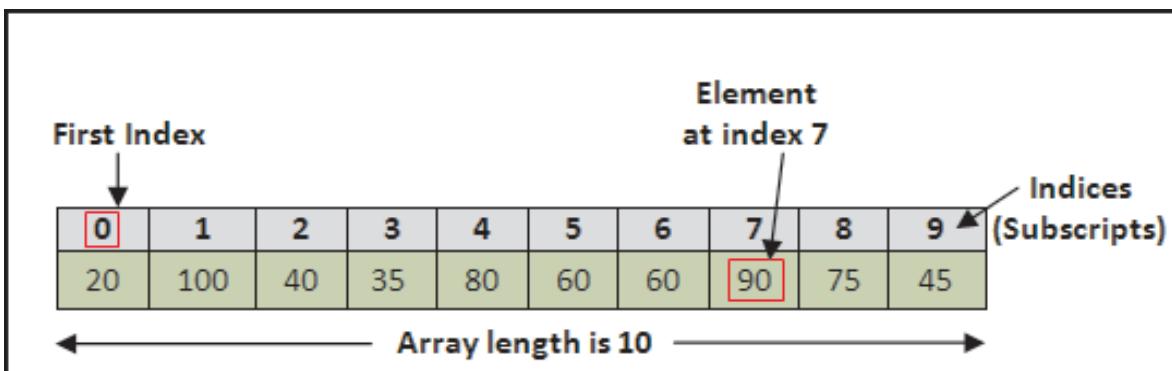


Figure 5.1: Array

Figure 5.1 displays an array of ten integers storing values such as, 20, 100, 40, and so on. Each value in the array is called an element of the array. The numbers 0 to 9 indicate the index or subscript of the elements in the array. The length or size of the array is 10. The first index begins with zero. Since the index begins with zero, the index of the last element is always length - 1. Therefore, the last, that is, tenth element in the given array has an index value of 9. Each element of the array can be accessed using the subscript or index. Array can be created from primitive data types such as `int`, `float`, `boolean` as well as from reference type such as `object`. The array elements are accessed using a single name but with different subscripts. The values of an array are stored at contiguous locations in memory. This induces less overhead on the system while searching for values. Use of arrays has following benefits:

- Arrays are the best way of operating on multiple data elements of the same type at the same time.
- Arrays make optimum use of memory resources as compared to variables.
- Memory is 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 it is declared.

Arrays in Java are of following two types:

- Single-dimensional arrays
- Multi-dimensional arrays

5.2.1 Declaring, Instantiating, and Initializing Single-dimensional Array

A single-dimensional array has only one dimension and is visually represented as having a single column with several rows of data. Each element is accessed using the array name and the index at which the element is located. Figure 5.2 shows a single-dimensional array named **marks**.

marks[4]	
Element	Value
marks[0]	65
marks[1]	47
marks[2]	75
marks[3]	50

Figure 5.2: Array **marks**

The size or length of the array is specified as **4** in square brackets '[]'. This means that **marks** array can hold a maximum of four values. Each element of the array is accessed by using the array name and index. For example, **marks[0]** indicates the first element in the array. Similarly, **marks[3]**, that is, **marks[length-1]** indicates the last element of the array. Notice that there is no element with index 4. Therefore, an attempt to write **marks[4]** will issue an exception. An exception is an abnormal event that occurs during the program execution and disrupts the normal flow of instructions. Array creation involves following tasks:

- Declaring an array
- Instantiating an array
- Initializing an array

Note - Array name should follow the same rules as naming a variable.

These tasks are explained as follows:

→ **Declaring an Array:**

Declaring an array is similar to declaring any other variable. Declaring an array notifies the compiler that the variable will contain an array of the specified data type. It does not create an array. The syntax for declaring a single-dimensional array is as follows:

Syntax:

```
datatype[] <array-name>;
```

where,

datatype: Indicates the type of elements that will be stored in the array.

[] : Indicates that the variable is an array.

array-name : Indicates the name by which the elements of the array will be accessed.

For example,

```
int[] marks;
```

Similarly, arrays of other types can also be declared as follows:

```
byte[] byteArray;
float[] floatsArray;
boolean[] booleanArray;
char[] charArray;
String[] stringArray;
```

→ Instantiating an Array:

Since array is an object, memory is allocated only when it is instantiated. The syntax for instantiating an array is as follows:

Syntax:

```
datatype[] <array-name> = new datatype[size];
```

where,

new : Allocates memory to the array.

size : Indicates the number of elements that can be stored in the array.

For example,

```
int[] marks = new int[4];
```

Similarly, arrays of other types can also be instantiated according to requirement.

→ Initializing an Array:

Since, array is an object that can store multiple values, array must be initialized with the values to be stored in it. Array can be initialized in following two ways:

- **During creation:**

To initialize a single-dimensional array during creation, one must specify the values to be stored while creating the array as follows:

```
int[] marks = {65, 47, 75, 50};
```

Notice that while initializing an array during creation, the `new` keyword or size is not required. This is because all the elements to be stored have been specified and accordingly the memory gets automatically allocated based on the number of elements. This is also known as declaration with initialization.

- **After creation:**

A single-dimensional array can also be initialized after creation and instantiation.

In this case, individual elements of the array must be initialized with appropriate values. For example,

```
int[] marks = new int[4];
marks[0] = 65;
marks[1] = 47;
marks[2] = 75;
marks[3] = 50;
```

Notice that in this case, the array must be instantiated and size must be specified. This is because, actual values are specified later and to store the values, memory must be allocated during creation of the array.

Another way of creating an array is to split all the three stages as follows:

```
int marks[]; // declaration
marks = new int[4]; // instantiation
marks[0] = 65; // initialization
```

Code Snippet 1 demonstrates an example of single-dimensional array.

Code Snippet 1:

```
package session5;
public class OneDimension {
    //Declare a single-dimensional array named marks
    int marks[]; // line 1
    /**
     * Instantiates and initializes a single-dimensional array
     *
     * @return void
     */
    public void storeMarks() {
        // Instantiate the array
        marks = new int[4]; // line 2
        System.out.println("Storing Marks. Please wait....");
        // Initialize array elements
        marks[0] = 65; // line 3
        marks[1] = 47;
        marks[2] = 75;
        marks[3] = 50;
    }
}
```

```

/**
 * Displays marks from a single-dimensional array
 *
 * @return void
 */
public void displayMarks() {
    System.out.println("Marks are:");
    // Display the marks
    System.out.println(marks[0]);
    System.out.println(marks[1]);
    System.out.println(marks[2]);
    System.out.println(marks[3]);
}
/** 
 * @param args the command line arguments
 */
public static void main(String[] args) {
    //Instantiate class OneDimension
    OneDimension oneDimenObj = new OneDimension(); //line 4
    //Invoke the storeMarks() method
    oneDimenObj.storeMarks(); // line 5
    //Invoke the displayMarks() method
    oneDimenObj.displayMarks(); // line 6
}
}

```

The class **OneDimension** consists of an array named **marks[]** declared in line 1. To instantiate and initialize the array elements, the method **storeMarks()** is created. The array is instantiated using the **new** keyword in line 2.

The elements of the array are assigned values by using the array name and the subscript of the element, that is, **marks[0]**. Similarly, to display array elements, the **displayMarks()** method is created and the values stored in each element of **marks[]** array is displayed.

The object **oneDimenObj** of the class **OneDimension** is created in line 4. The object is used to invoke the **storeMarks()** and **displayMarks()** methods.

Figure 5.3 shows the output of the code.

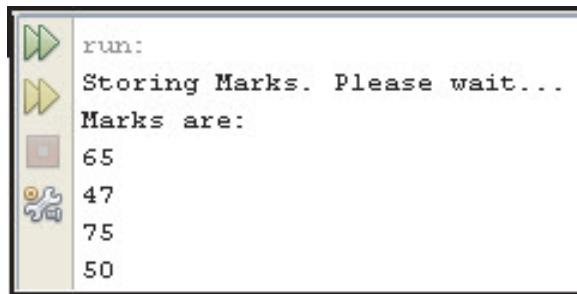


Figure 5.3: Output of Single-dimensional Array

5.2.2 Declaring, Instantiating, and Initializing Multi-dimensional Array

A user can create an array of arrays, that is, a multi-dimensional array. It can be created by using two or more sets of square brackets, such as `int[][] marks`. Therefore, each element must be accessed by the corresponding number of indices.

A multi-dimensional array in Java is an array whose elements are also arrays. This allows the rows to vary in length.

The syntax for declaring and instantiating a multi-dimensional array is as follows:

Syntax:

```
datatype[][] <array-name> = new datatype [rowsize] [colszie];
```

where,

`datatype`: Indicates the type of elements that will be stored in the array.

`rowsize` and `colszie`: Indicates the number of rows and columns that the array will contain.

`new`: Keyword used to allocate memory to the array elements.

For example,

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

The array named `marks` consists of four rows and two columns. Similarly, arrays of other types can also be instantiated according to the requirement.

A multi-dimensional array can be initialized in following two ways:

→ During creation:

To initialize a multi-dimensional array during creation, one must specify the values to be stored while creating the array as follows:

```
int[][] marks = {{23, 65}, {42, 47}, {60, 75}, {75, 50}};
```

Notice that while initializing an array during creation, the elements in rows are specified in a set of curly brackets separated by a comma delimiter. Also, the individual rows are separated by a comma separator.

This is a two-dimensional array that can be represented in a tabular form as shown in Figure 5.4.

Rows	Columns	
	0	1
0	23	65
1	42	47
2	60	75
3	75	50

Figure 5.4: Two-dimensional Array

→ **After creation:**

A multi-dimensional array can also be initialized after creation and instantiation. In this case, individual elements of the array must be initialized with appropriate values. Each element is accessed with a row and column subscript. For example,

```
int[][] marks = new int[4][2];
marks[0][0] = 23; // first row, first column
marks[0][1] = 65; // first row, second column

marks[1][0] = 42;
marks[1][1] = 47;

marks[2][0] = 60;
marks[2][1] = 75;
marks[3][0] = 75;
marks[3][1] = 50;
```

Here, the element **23** is said to be at position (0,0), that is, first row and first column. Therefore, to store or access the value **23**, one must use the syntax **marks [0] [0]**. Similarly, for other values, the appropriate row-column combination must be used. Similar to row index, column index also starts at zero. Therefore, in the given scenario, an attempt to write **marks [0] [2]** would result in an exception as the column size is **2** and column indices are **0** and **1**.

Code Snippet 2 demonstrates an example of two-dimensional array.

Code Snippet 2:

```
package session5;
public class TwoDimension {
    //Declare a two-dimensional array named marks
    int marks[][]; //line 1
    /**
     * Stores marks in a two-dimensional array
     *
     * @return void
     */
    public void storeMarks() {
        // Instantiate the array
        marks = new int[4][2]; // line 2
        System.out.println("Storing Marks. Please wait...");
        // Initialize array elements
        marks[0][0] = 23; // line 3
        marks[0][1] = 65;
        marks[1][0] = 42;
        marks[1][1] = 47;
        marks[2][0] = 60;
        marks[2][1] = 75;
        marks[3][0] = 75;
        marks[3][1] = 50;
    }
    /**
     * Displays marks from a two-dimensional array
     *
     * @return void
     */
    public void displayMarks() {
        System.out.println("Marks are:");
        // Display the marks
        System.out.println("Roll no.1:" + marks[0][0] + "," + marks[0][1]);
    }
}
```

```

System.out.println("Roll no.2:" + marks[1][0]+ "," + marks[1][1]);
System.out.println("Roll no.3:" + marks[2][0]+ "," + marks[2][1]);
System.out.println("Roll no.4:" + marks[3][0]+ "," + marks[3][1]);

}

/**
 * @param args the command line arguments
 */

public static void main(String[] args) {
    //Instantiate class TwoDimension
    TwoDimension twoDimenObj = new TwoDimension(); // line 4
    //Invoke the storeMarks() method
    twoDimenObj.storeMarks();
    //Invoke the displayMarks() method
    twoDimenObj.displayMarks();
}
}

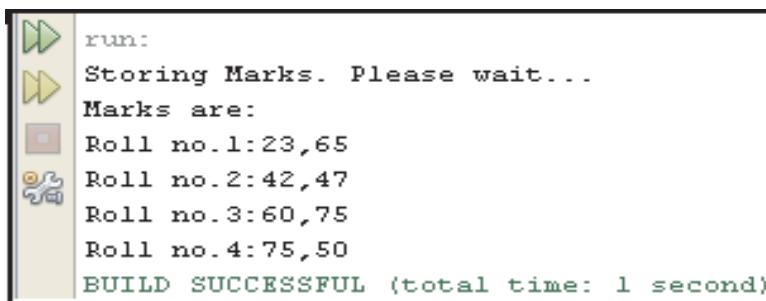
```

The class **TwoDimension** consists of an array named **marks[][]** declared in line 1. To instantiate and initialize array elements, the method **storeMarks()** is created.

The array is instantiated using the **new** keyword in line 2. The elements of the array are assigned values by using the array name and the row-column subscript of the element, that is, **marks[0][0]**. Similarly, to display the array elements, the **displayMarks()** method is created and invoked and the values stored in each element of **marks[][]** array is displayed.

The object **twoDimenObj** of the class **TwoDimension** is created in line 4. The object is used to invoke the **storeMarks()** and **displayMarks()** methods.

Figure 5.5 shows the output of the code, that is, marks of four students are displayed from the array **marks[][]**.



```

run:
Storing Marks. Please wait...
Marks are:
Roll no.1:23,65
Roll no.2:42,47
Roll no.3:60,75
Roll no.4:75,50
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 5.5: Output of Two-dimensional Array

5.2.3 Using Loops to Process and Initialize an Array

Consider a situation when a user has to display hundreds and thousands of values in an array. The methods mentioned earlier to access individual elements of the array would be very tedious and time consuming. In such a case, the user can use loops to process and initialize an array. Code Snippet 3 depicts the revised `displayMarks()` method of the single-dimensional array named `marks[]`.

Code Snippet 3:

```
...
public void displayMarks() {
    System.out.println("Marks are:");
    // Display the marks using for loop
    for(int count = 0; count < marks.length; count++) {
        System.out.println(marks[count]);
    }
}
...
```

In Code Snippet 3, a `for` loop has been used to iterate the array from zero to `marks.length`. The property, `length`, of the array object is used to obtain the size of the array. Within the loop, each element is displayed by using the element name and the variable `count`, that is, `marks[count]`.

Code Snippet 4 depicts the revised `displayMarks()` method of the two-dimensional array `marks[][]`.

Code Snippet 4:

```
...
public void displayMarks() {
    System.out.println("Marks are:");
    // Display the marks using nested for loop
    // outer loop
    for (int row = 0; row < marks.length; row++) {
        System.out.println("Roll no." + (row+1));
        // inner loop
        for (int col = 0; col < marks[row].length; col++) {
            System.out.println(marks[row][col]);
        }
    }
}
...
```

In Code Snippet 4, a nested `for` loop (outer and inner) has been used to iterate through the array `marks`. The outer loop keeps track of the number of rows and inner loop keeps track of the number of columns in each row. For each row, the inner loop iterates through all the columns using `marks[row].length`. Within the inner loop, each element is displayed by using the element name and the row-column count, that is, `marks[row][column]`.

Figure 5.6 shows the output of the two-dimensional array `marks[][]`, after using the `for` loop.

```

run:
Storing Marks. Please wait...
Marks are:
Roll no.1
23
65
Roll no.2
42
47
Roll no.3
60
75
Roll no.4
75
50
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 5.6: Output of Two-dimensional Array Using Loop

One can also use the enhanced `for` loop to iterate through an array. Code Snippet 5 depicts the modified `displayMarks()` method of single-dimensional array `marks[]` using the enhanced `for` loop.

Code Snippet 5:

```

...
public void displayMarks () {
    System.out.println("Marks are:");
    // Display the marks using enhanced for loop
    for(int value:marks) {
        System.out.println(value);
    }
}
...

```

Here, the loop will print all the values of `marks[]` array till `marks.length` without having to explicitly specify the initializing and terminating conditions for iterating through the loop.

Code Snippet 6 demonstrates the calculation of total marks of each student by using the `for` loop and the enhanced `for` loop together with the two-dimensional array `marks[][]`.

Code Snippet 6:

```
...
public void totalMarks() {
    System.out.println("Total Marks are:");
    // Display the marks using for loop and enhanced for loop
    for (int row = 0; row < marks.length; row++) {
        System.out.println("Roll no." + (row+1));
        int sum = 0;
        // enhanced for loop
        for (int value : marks[row]) {
            sum = sum + value;
        }
        System.out.println(sum);
    }
}
...
```

Here, enhanced `for` loop is used to iterate through columns of row selected in the outer loop using `marks[row]`. The code `sum = sum + value` will add up values of all columns of currently selected row. The selected row is indicated by the subscript variable `row`.

Figure 5.7 shows sum of the values of the two-dimensional array named `marks[][]` using `for` loop and enhanced `for` loop together.

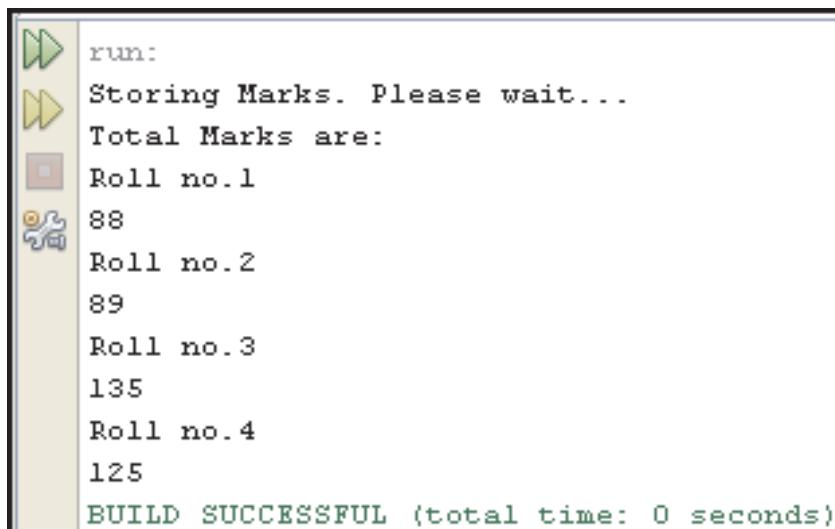


Figure 5.7: Sum of Values of a Two-dimensional Array

5.2.4 Initializing an ArrayList

One major disadvantage of an array is that its size is fixed during creation. The size cannot be modified later. However, it is sometimes not possible to know in advance as to how many elements will be stored in an array. For example, when a person is shopping online, the number of items that will be added to the shopping cart is not fixed from the beginning. In such a case, one might have to create an array of the largest possible size that may be sufficient to store all the items. However, if the user adds only a few items to the cart, then, the rest of the memory occupied by the array will be wasted. Similarly, if the user tries to add more items to the cart that exceeds the size of an array, it will generate an error. Also, addition and deletion of values from an array is a difficult task. Another disadvantage of an array is that it can hold only one type of elements.

To resolve this issue, it is required to have a construct to which memory can be allocated based on requirement. Also, addition and deletion of values can be performed easily. Java provides the concept of collections to address this problem.

A collection is a single object that groups multiple elements into a single unit. Collections are used to store, retrieve, and manipulate aggregate data. Typically, they represent data items that form a natural group, such as a collection of cards, a collection of letters, or a set of names and phone numbers.

Java provides a set of collection interfaces to create different types of collections. The core Collection interfaces that encapsulate different types of collections are shown in Figure 5.8.

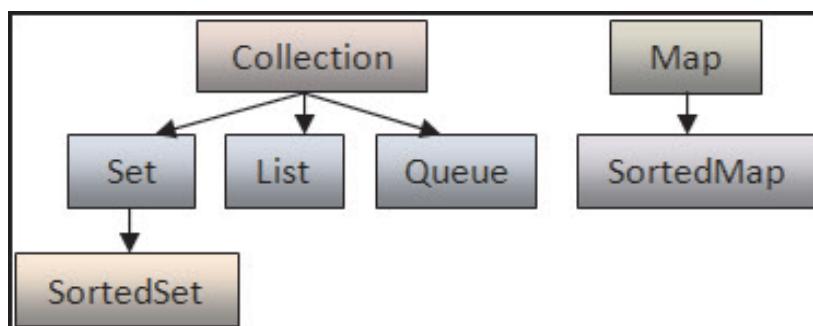


Figure 5.8: Core Collection Interfaces

The general-purpose implementations are summarized in Table 5.1.

Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet	-	TreeSet	-	LinkedHashSet
List	-	ArrayList	-	LinkedList	-
Queue	-	-	-	-	-
Map	HashMap	-	TreeMap	-	LinkedHashMap

Table 5.1: General-purpose Implementations of Core Collections

Various implementations of the core collections can be used in different situations. However, `HashSet`, `ArrayList`, and `HashMap` are useful for most applications. Also, the `SortedSet` and the `SortedMap`

interfaces are not included in the Table 5.1 since each has one implementation `TreeSet` and `TreeMap` respectively and is listed in the `Set` and the `Map` rows. `Queue` has two implementations namely, `LinkedList` which is the `List` implementation, and `PriorityQueue`, which is not listed in the table. These two implementations provide very different semantics. `LinkedList` uses First In First Out (FIFO) ordering, while `PriorityQueue` orders the elements according to their values.

Each of the implementations provides all optional operations present in its interface. All implementations permit `null` elements, keys, and values. To use the interfaces, the user must import the `java.util` package into the class.

Note - A package is a collection of related classes. The `java.util` package consists of all the collection interfaces and classes.

The `ArrayList` class is a frequently used collection that has following characteristics:

- It is flexible and can be increased or decreased in size as required.
- Provides several useful methods to manipulate the collection.
- Insertion and deletion of data is simpler.
- It can be traversed by using `for` loop, enhanced `for` loop, or other iterators.

The `ArrayList` collection provides methods to manipulate the size of the array. `ArrayList` extends `AbstractList` and implements the interfaces such as `List`, `Cloneable`, and `Serializable`. The capacity of an `ArrayList` grows automatically. It stores all elements including `null`.

Table 5.2 lists the constructors of `ArrayList` class.

Constructor	Description
<code>ArrayList()</code>	Creates an empty array list.
<code>ArrayList(Collection c)</code>	Creates an array list initialized with the elements of the collection <code>c</code> .
<code>ArrayList(int capacity)</code>	Creates an array list with a specified initial capacity. Capacity is the size of the underlying array used to store the elements. The capacity can grow automatically as elements are added to an array list.

Table 5.2: Constructors of `ArrayList` Class

`ArrayList` consists of several methods for adding elements. These methods can be broadly divided into following two categories:

- Methods that append one or more elements to the end of the list.
- Methods that insert one or more elements at a position within the list.

Table 5.3 lists the methods of `ArrayList` class.

Method	Description
<code>void add(int index, Object element)</code>	Inserts the specified element at the given index in this list. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .
<code>boolean add(Object o)</code>	Appends the specified element to the end of this list.
<code>boolean addAll(Collection c)</code>	Appends all elements in the specified collection to the end of this list. If the specified collection is <code>null</code> , it throws <code>NullPointerException</code> .
<code>boolean addAll(int index, Collection c)</code>	Inserts all elements in the specified collection into this list, starting at the specified index. If the collection is <code>null</code> , it throws <code>NullPointerException</code> .
<code>void clear()</code>	Removes all elements from this list.
<code>Object clone()</code>	Returns a copy of the <code>ArrayList</code> .
<code>boolean contains(Object o)</code>	Returns true if and only if the list contains the specified element.
<code>void ensureCapacity(int minCapacity)</code>	Increases the capacity of the <code>ArrayList</code> , if required, to ensure that it can store at least as many number of elements as indicated by the minimum capacity.
<code>Object get(int index)</code>	Returns the element at the specified index in this list. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .
<code>int indexOf(Object o)</code>	Returns the index of the first occurrence of the specified element in the list. If the element is not found, it returns <code>-1</code> .
<code>int lastIndexOf(Object o)</code>	Returns the index of the last occurrence of the specified element in this list. If the element is not found, it returns <code>-1</code> .
<code>Object remove(int index)</code>	Removes the element at the specified index in this list. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .
<code>protected void removeRange(int fromIndex, int toIndex)</code>	Removes all the elements between <code>fromIndex</code> , inclusive and <code>toIndex</code> , exclusive of the list.
<code>Object set(int index, Object element)</code>	Replaces the element at the specified index in this list with the newly specified element. If <code>index >= size()</code> or <code>index < 0</code> , it throws <code>IndexOutOfBoundsException</code> .
<code>int size()</code>	Returns the number of elements in this list.
<code>Object[] toArray()</code>	Returns an array containing all elements in the list in the correct order. If the array is <code>null</code> , it throws <code>NullPointerException</code> .

Method	Description
Object[] toArray(Object[] a)	Returns an array containing all elements in the list in the correct order. The type of the returned array is same as that of the specified array.
void trimToSize()	Trims the capacity of the ArrayList to the list's actual size.

Table 5.3: Methods of ArrayList Class

To traverse an ArrayList, one can use any of these approaches:

- A for loop
- An enhanced for loop
- Iterator
- ListIterator

Iterator interface provides methods for traversing a set of data. It can be used with arrays as well as various classes of the Collection framework.

The Iterator interface provides following methods for traversing a collection:

- next(): This method returns the next element of the collection.
- hasNext(): This method returns true if there are additional elements in the collection.
- remove(): This method removes the element from the list while iterating through the collection.

There are no specific methods in the ArrayList class for sorting. However, one can use the sort() method of the Collections class to sort an ArrayList. The syntax for using the sort() method is as follows:

Syntax:

```
Collections.sort(<list-name>);
```

Code Snippet 7 demonstrates instantiation and initialization of an ArrayList.

Code Snippet 7:

```
ArrayList marks = new ArrayList(); // Instantiate an ArrayList
marks.add(67); // Initialize an ArrayList
marks.add(50);
```

5.2.5 Accessing Values in an ArrayList

An ArrayList can be iterated by using the for loop or by using the Iterator interface. Code Snippet 8 demonstrates the use of ArrayList named **marks** to add and display marks of students.

Code Snippet 8:

```
package session5;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
public class ArrayListDemo{

    // Create an ArrayList instance
    ArrayList marks = new ArrayList(); // line 1

    /**
     * Stores marks in ArrayList
     *
     * @return void
     */
    public void storeMarks () {
        System.out.println("Storing marks. Please wait....");
        marks.add(67); // line 2
        marks.add(50);
        marks.add(45);
        marks.add(75);
    }

    /**
     * Displays marks from ArrayList
     *
     * @return void
     */
    public void displayMarks () {
        System.out.println("Marks are:");
        // iterating the list using for loop
        System.out.println("Iterating ArrayList using for loop:");
        for (int i = 0; i < marks.size(); i++) {
            System.out.println(marks.get(i));
        }
    }
}
```

```

System.out.println("-----");
// Iterate the list using Iterator interface
Iterator imarks = marks.iterator(); // line 3
System.out.println("Iterating ArrayList using Iterator:");
while (imarks.hasNext()) { // line 4
    System.out.println(imarks.next()); // line 5
}
System.out.println("-----");
// Sort the list
Collections.sort(marks); // line 6
System.out.println("Sorted list is: " + marks);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // Instantiate the class OneDimension
    ArrayListDemo obj = new ArrayListDemo(); // line 7
    // Invoke the storeMarks() method
    obj.storeMarks();
    // Invoke the displayMarks() method
    obj.displayMarks();
}
}

```

The `ArrayList` named `marks` has been instantiated in line 1. The `storeMarks()` method is used to add elements to the collection by using `marks.add()` method as shown in line 2.

Within the `displayMarks()` method, a `for` loop is used to iterate the `ArrayList` `marks` from 0 to `marks.size()`. The `get()` method is used to retrieve the element at index `i`.

Similarly, the `Iterator` object `imarks` is instantiated in line 3 and attached with the `marks` `ArrayList` using `marks.iterator()`. It is used to iterate through the collection. The `Iterator` interface provides the `hasNext()` method to check if there are any more elements in the collection as shown in line 4. The method, `next()` is used to traverse to the next element in the collection. The retrieved element is then displayed to the user in line 5.

The static method, `sort()` of `Collections` class is used to sort the `ArrayList` `marks` in line 6 and print the values on the screen. Within `main()` method, the object of `ArrayLists` class has been created in line 7 and the methods `storeMarks()` and `displayMarks()` have been invoked.

Figure 5.9 shows the output of the code.

```

run:
Storing marks. Please wait...
Marks are:
Iterating ArrayList using for loop:
67
50
45
75
-----
Iterating ArrayList using Iterator:
67
50
45
75
-----
Sorted list is: [45, 50, 67, 75]

```

Figure 5.9: Output of `ArrayList` Marks

The values of an `ArrayList` can also be printed by simply writing `System.out.println("Marks are:" + marks)`. In this case the output would be: `Marks are: [67, 50, 45, 75]`.

5.3 Introduction to Strings

Consider a scenario, where in a user wants to store the name of a person. One can create a character array as shown in Code Snippet 9.

Code Snippet 9:

```
char[] name = {'J', 'u', 'l', 'i', 'a'}
```

Similarly, to store names of multiple persons, one can create a two-dimensional array. However, the number of characters in an array must be fixed during creation. This is not possible since the names of persons may be of variable sizes. Also, manipulating the character array would be tedious and time consuming. Java provides the `String` data type to store multiple characters without creating an array.

5.3.1 Strings

`String` literals such as "Hello" in Java are implemented as instances of the `String` class. `Strings` are constant and immutable, that is, their values cannot be changed once they are created. `String buffers` allow creation of mutable strings.

A simple `String` instance can be created by enclosing a string literal inside double quotes as shown in Code Snippet 10.

Code Snippet 10:

```
...
String name = "Mary";
// This is equivalent to:
char name[] = {'M', 'a', 'r', 'y'};
...
```

An instance of a `String` class can also be created using the `new` keyword, as shown in Code Snippet 11.

Code Snippet 11:

```
String str = new String();
```

The code creates a new object of class `String`, and assigns its reference to the variable `str`.

Java also provides special support for concatenation of strings using the plus (+) operator and for converting data of other types to strings as depicted in Code Snippet 12.

Code Snippet 12:

```
...
String str = "Hello";
String str1 = "World";
// The two strings can be concatenated by using the operator '+'
System.out.println(str + str1);
// This will print 'HelloWorld' on the screen
...
```

One can convert a character array to a string as depicted in Code Snippet 13.

Code Snippet 13:

```
char[] name = {'J', 'o', 'h', 'n'};
String empName = new String(name);
```

The `java.lang.String` class is a final class, that is, no class can extend it. The `java.lang.String` class differs from other classes, in that one can use '`+=`' and '`+`' operators with `String` objects for concatenation.

If the string is not likely to change later, one can use the `String` class. Thus, a `String` class can be used for following reasons:

- String is immutable and so it can be safely shared between multiple threads.
- The threads will only read them, which is normally a thread safe operation.

Note - A thread is a single unit of execution in a program. The JVM allows an application to execute multiple threads of a process concurrently.

However, if the string is likely to change later and it will be shared between multiple threads, one can use the `StringBuffer` class. The use of `StringBuffer` class ensures that the string is updated correctly. However, the drawback is that the method execution is comparatively slower.

If the string is likely to change later but will not be shared between multiple threads, one can use the `StringBuilder` class. The `StringBuilder` class can be used for following reasons:

- It allows modification of the strings without the overhead of synchronization.
- Methods of `StringBuilder` class execute as fast as, or faster, than those of the `StringBuffer` class

5.3.2 Working with String Class

The `String` class provides methods for manipulating individual characters of the string, comparing strings, extracting substrings, searching strings, and for converting a string to uppercase or to lowercase. Some of the frequently used methods of `String` class are as follows:

- `length(String str)`

The `length()` method is used to find the length of a string. For example,

```
String str = "Hello";
System.out.println(str.length()); // output: 5
```

- `charAt(int index)`

The `charAt()` method is used to retrieve the character value at a specific index. The index ranges from zero to `length() - 1`. The index of the first character starts at zero. For example,

```
System.out.println(str.charAt(2)); // output: 'l'
```

- `concat(String str)`

The `concat()` method is used to concatenate a string specified as argument to the end of another string. If the length of the string is zero, the original `String` object is returned, otherwise a new `String` object is returned.

```
System.out.println(str.concat("World")); // output: 'HelloWorld'
```

→ compareTo(String str)

The `compareTo()` method is used to compare two `String` objects. The comparison returns an integer value as the result. The comparison is based on the Unicode value of each character in the strings. That is, the result will return a negative value, if the argument string is alphabetically greater than the original string. The result will return a positive value, if argument string is alphabetically lesser than the original string and the result will return a value of zero, if both the strings are equal. For example,

```
System.out.println(str.compareTo("World")); // output: -15
```

The output is **15** because, the second string "**World**" begins with '**W**' which is alphabetically greater than the first character '**H**' of the original string, `str`. The difference between the position of '**H**' and '**W**' is **15**. Since '**H**' is smaller than '**W**', the result will be **-15**.

Note - Unicode is a standard that provides a unique number for every character irrespective of the platform, program, or language. The Unicode Standard has been adopted for programming by major industry leaders such as IBM, Apple, Microsoft, HP, Oracle, Sun, SAP, and several others.

→ indexOf(String str)

The `indexOf()` 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**. For example,

```
System.out.println(str.indexOf("e")); // output: 1
```

→ lastIndexOf(String str)

The `lastIndexOf()` method returns the index of the last occurrence of a specified character or string from within a string. The specified character or string is searched backwards that is the search begins from the last character. For example,

```
System.out.println(str.lastIndexOf("l")); // output: 3
```

→ replace(char old, char new)

The `replace()` method is used to replace 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. For example,

```
System.out.println(str.replace('e', 'a')); // output: 'Hallo'
```

→ substring(int beginIndex, int endIndex)

The `substring()` method is used to retrieve a part of a string, that is, substring from the given string. One can specify the start index and the end index for the substring. If end index is not specified, all characters from the start index to the end of the string will be returned.

The substring begins at the specified position denoted by `index` or `beginIndex` and extends to the character specified by `index` or `endIndex - 1`.

Thus, the length of the substring is endIndex – beginIndex.

Here, the beginIndex is included in the output whereas the endIndex is excluded.

For example,

```
System.out.println(str.substring(2, 5)); // output: 'llo'
```

→ **toString()**

The `toString()` method is used to return a `String` object. It is used to convert values of other data types into strings. For example,

```
Integer length = 5;
```

```
System.out.println(length.toString()); // output: 5
```

Notice that the output is still 5. However, now it is represented as a string instead of an integer.

Note - The class `Integer` used in the example is a Wrapper class for the primitive data type `int`.

→ **trim()**

The `trim()` method returns a new string by trimming the leading and trailing whitespace from the current string. For example,

```
String str1 = " Hello ";
```

```
System.out.println(str1.trim()); // output: 'Hello'
```

The `trim()` method will return '`Hello`' after removing the spaces.

Code Snippet 14 demonstrates the use of `String` class methods.

Code Snippet 14:

```
public class Strings {
    String str = "Hello"; // Initialize a String variable
    Integer strLength = 5; // Use the Integer wrapper class
    /**
     * Displays strings using various String class methods
     *
     * @return void
     */
    public void displayStrings() {
        // using various String class methods
        System.out.println("String length is:"+ str.length());
        System.out.println("Character at index 2 is:"+ str.charAt(2));
    }
}
```

```
System.out.println("Concatenated string is:"+ str.concat("World"));

System.out.println("String comparison is:"+ str.compareTo("World"));

System.out.println("Index of o is:"+ str.indexOf("o"));

System.out.println("Last index of l is:"+ str.lastIndexOf("l"));

System.out.println("Replaced string is:"+ str.replace('e', 'a'));

System.out.println("Substring is:"+ str.substring(2, 5));

System.out.println("Integer to String is:"+ strLength.toString());

String str1="Hello ";

System.out.println("Untrimmed string is:"+ str1);

System.out.println("Trimmed string is:"+ str1.trim());

}

/** 

* @param args the command line arguments

*/



public static void main(String[] args) {

//Instantiate class, Strings

    Strings objString = new Strings(); // line 1

//Invoke the displayStrings() method

    objString.displayStrings();

}

}
```

The class **Strings** consists of a **String** variable **str**. The member variable has been initialized with the value "Hello" and an Integer variable named **strLength** has been initialized with a value 5. The **displayStrings()** method is used to display the values generated by using various **String** class methods. The class **Strings** is instantiated in line 1. The object, **objString** is used to invoke the **displayStrings()** method.

Figure 5.10 shows the output of the `Strings.java` class.

```

run:
String length is:5
Character at index 2 is:1
Concatenated string is:HelloWorld
String comparison is:-15
Index of o is:4
Last index of l is:3
Replaced string is:Hallo
Substring is:llo
Integer to String is:5
Untrimmed string is: Hello
Trimmed string is:Hello

```

Figure 5.10: Output of `Strings.java` Class

5.3.3 Working with `StringBuilder` Class

`StringBuilder` objects are similar to `String` objects, except that they are mutable. Internally, the system treats these objects as a variable-length array containing a sequence of characters. The length and content of the sequence of characters can be changed through methods available in the `StringBuilder` class. However, developers prefer to use `String` class unless `StringBuilder` offers an advantage of simpler code in some cases. For example, for concatenating a large number of strings, using a `StringBuilder` object is more efficient.

The `StringBuilder` class also provides a `length()` method that returns the length of the character sequence in the class.

However, unlike strings a `StringBuilder` object also has a property `capacity` that specifies the number of character spaces that have been allocated. The `capacity` is returned by the `capacity()` method and is always greater than or equal to the length. The capacity to hold data in the instance of the `StringBuilder` class will automatically expand according to the user requirement to accommodate the new strings when added to the string builder.

Thus, the `StringBuilder` class is used for manipulating the `String` object. Objects of `StringBuilder` class are mutable and flexible. `StringBuilder` object allows insertion of characters and strings as well as appending characters and strings at the end.

Constructors of the `StringBuilder` class are as follows:

- `StringBuilder():` Default constructor that provides space for 16 characters.
- `StringBuilder(int capacity):` Constructs an object without any characters in it. However, it reserves space for the number of characters specified in the argument, `capacity`.

- `StringBuilder(String str)`: Constructs an object that is initialized with the contents of the specified string, str.

5.3.4 Methods of `StringBuilder` Class

The `StringBuilder` class provides several methods for appending, inserting, deleting, and reversing strings as follows:

- `append()`

The `append()` method is used to append values at the end of the `StringBuilder` object. This method accepts different types of arguments, including `char`, `int`, `float`, `double`, `boolean`, and so on, but the most common argument is `String`.

For each `append()` method, `String.valueOf()` method is invoked to convert the parameter into a corresponding string representation value and then, the new string is appended to `StringBuilder` object.

For example,

```
StringBuilder str = new StringBuilder("JAVA ");
System.out.println(str.append("SE ")); // output: JAVA SE
System.out.println(str.append(7)); // output: JAVA SE 7
```

- `insert()`

The `insert()` method is used to insert one string into another. Similar to the `append()` method, it calls the `String.valueOf()` method to obtain the string representation of the value. The new string is inserted into the invoking `StringBuilder` object.

The `insert()` method has several versions as follows:

- `StringBuilder insert(int insertPosition, String str)`
- `StringBuilder insert(int insertPosition, char ch)`
- `StringBuilder insert(int insertPosition, float f)`

For example,

```
StringBuilder str = new StringBuilder("JAVA 7 ");
System.out.println(str.insert(5, "SE")); // output: JAVA SE 7
```

- `delete()`

The `delete()` method deletes the specified number of characters from the invoking `StringBuilder` object.

For example,

```
StringBuilder str = new StringBuilder("JAVA SE 7 ");
System.out.println(str.delete(4, 7));
```

→ reverse()

The `reverse()` method is used to reverse the characters within a `StringBuilder` object.

For example,

```
StringBuilder str = new StringBuilder("JAVA SE 7");
System.out.println(str.reverse()); // output: 7 ES AVAJ
```

Code Snippet 15 demonstrates the use of methods of the `StringBuilder` class.

Code Snippet 15:

```
public class StringBuilders {
    // Instantiate a StringBuilder object
    StringBuilder str = new StringBuilder("JAVA ");
    /**
     * Displays strings using various StringBuilder methods
     * @return void
     */
    public void displayStrings() {
        // Use various methods of the StringBuilder class
        System.out.println("Appended String is "+ str.append("7"));
        System.out.println("Inserted String is "+ str.insert(5, "SE "));
        System.out.println("Deleted String is "+ str.delete(4, 7));
        System.out.println("Reverse String is "+ str.reverse());
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //Instantiate the StringBuilders class
        StringBuilders objStrBuild = new StringBuilders(); // line 1
        //Invoke the displayStrings() method
        objStrBuild.displayStrings();
    }
}
```

The class **StringBuilder**s consists of a **StringBuilder** object named **str** and initializes it with the value "JAVA ". The **displayStrings()** method is used to display the output generated after using various **StringBuilder** class methods. The class **StringBuilder**s is instantiated in line 1. The object **objStrBuild** is used to invoke the **displayStrings()** method.

Figure 5.11 shows the output of the **StringBuilder.java** class.

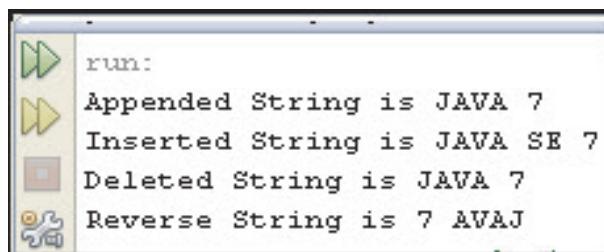


Figure 5.11: Output of **StringBuilder.java** Class

5.3.5 String Arrays

Sometimes there is a requirement to store a collection of strings. **String arrays** can be created in Java in the same manner as arrays of primitive data types. For example,

```
String[] empNames = new String[10];
```

This statement will allocate memory to store references of 10 strings. However, no memory is allocated to store the characters that make up the individual strings. Loops can be used to initialize as well as display the values of a **String array**.

Code Snippet 16 demonstrates the creation of a **String array**.

Code Snippet 16:

```
public class StringArray {
    // Instantiate a String array
    String[] empID = new String[5];
    /**
     * Creates a String array
     * @return void
     */
    public void createArray() {
        System.out.println("Creating Array. Please wait....");
        // Use a for loop to initialize the array
        for(int count = 1; count < empID.length; count++) {
            empID[count] = "E00"+count; // storing values in the array
        }
    }
}
```

```

    }

}

/***
 * Displays the elements of a String array
 * @return void
 */

public void printArray() {
    System.out.println("The Array is:");
    // Use a for loop to print the array
    for(int count = 1; count < empID.length; count++) {
        System.out.println("Employee ID is: "+ empID[count]);
    }
}

/***
 * @param args the command line arguments
 */

public static void main(String[] args) {
    //Instantiate class Strings
    StringArray objStrArray = new StringArray(); // line 1
    //Invoke createArray() method
    objStrArray.createArray();
    //Invoke printArray() method
    objStrArray.printArray();
}
}

```

The class **StringArray** consists of a String array object named **empID** having a size of 5. The **createArray()** method is used to initialize the elements of the array using the **for** loop and the **printArray()** method is used to print the values of the array. The class **StringArray** is instantiated in line 1. The object **objStrArray** is used to invoke the **createArray()** and **printArray()** methods.

Figure 5.12 shows the output of the `StringArray.java` class.

```

run:
Creating Array. Please wait...
The Array is:
Employee ID is: E001
Employee ID is: E002
Employee ID is: E003
Employee ID is: E004

```

Figure 5.12: Output of `StringArray.java` Class

5.4 Wrapper Classes

Java provides a set of classes known as wrapper classes for each of its primitive data type that 'wraps' the primitive type into an object of that class. In other words, the wrapper classes allow accessing primitive data types as objects. The wrapper classes for the primitive data types are: `Byte`, `Character`, `Integer`, `Long`, `Short`, `Float`, `Double`, and `Boolean`.

The wrapper classes are part of the `java.lang` package. The primitive types and the corresponding wrapper types are listed in Table 5.4.

Primitive Type	Wrapper Class
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>short</code>	<code>Short</code>
<code>boolean</code>	<code>Boolean</code>

Table 5.4: Primitive Types and Wrapper Classes

Now, one might ask 'What is the requirement for wrapper classes?'. The use of primitive types as objects can simplify tasks at times. For example, most of the collections store objects and not primitive data types. In other words, many of the activities reserved for objects will not be available to primitive data types. Also, many utility methods are provided by the wrapper classes that help to manipulate data. Since, wrapper classes convert primitive data types to objects, they can be stored in any type of collection and also passed as parameters to methods.

Wrapper classes can convert numeric strings to numeric values. The `valueOf()` method is available with all the wrapper classes to convert a type into another type. However, the `valueOf()` method of the `Character` class accepts only `char` as an argument whereas any other wrapper class accepts either the corresponding primitive type or `String` as an argument. The `typeValue()` method can also be used to return the value of an object as its primitive type.

Some of the wrapper classes and their methods are listed in Table 5.5.

Wrapper Class	Methods	Example
Byte	byteValue() – returns a byte value of the invoking object. parseByte() – returns the byte value from a string storing a byte value.	byte byteVal = Byte.byteValue(); byte byteVal = Byte.parseByte("45");
Character	isDigit() – checks if a character is a digit. isLowerCase() – checks if a character is a lower case alphabet. isLetter() – checks if a character is an alphabet.	if(Character.isDigit('4')) System.out.println("Digit"); if(Character.isLetter('L')) System.out.println("Letter");
Integer	intValue() – returns the Integer value as a primitive type int. parseInt() - returns the int value from a string storing an integer value.	int intValue = Integer.intValue(); int intVal= Integer.parseInt("45");

Table 5.5: Methods of Wrapper Classes

The difference between creation of a primitive type and a wrapper type is as follows:

Primitive type:

```
int x = 10;
```

Wrapper type:

```
Integer y = new Integer(20);
```

The first statement declares and initializes the int variable **x** to 10 whereas the second statement instantiates an Integer object **y** and initializes it with the value 20. In this case, the reference of the object is assigned to the object variable **y**. The memory assignment for the two statements is shown in Figure 5.13.

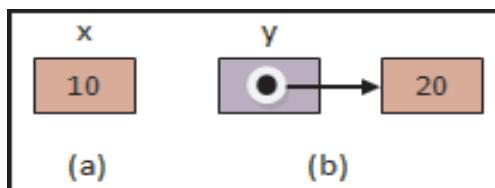


Figure 5.13: Primitive Type and Wrapper Type

It is clear from the Figure 5.13 that **x** is a variable that holds a value whereas **y** is an object variable that holds a reference to an object.

The six methods of type `parseXxx()` available for each numeric wrapper type are in close relation to the `valueOf()` methods of all the numeric wrapper classes including `Boolean`.

The two type of methods, that is, `parseXxx()` and `valueOf()`, take a `String` as an argument and if the `String` argument is not properly formed, both the methods throw a `NumberFormatException`. These methods can convert `String` objects of different bases if the underlying primitive type is any of the four integer types.

However, the `parseXxx()` method returns a named primitive whereas the `valueOf()` method returns a new wrapped object of the type that invoked the method.

Code Snippet 17 demonstrates the use of `Integer` wrapper class to convert the numbers passed by user as strings at command line into integer types to perform the calculation based on the selected operation.

Code Snippet 17:

```
package session5;
public class Wrappers {
    /**
     * Performs calculation based on user input
     *
     * @return void
     */
    public void calcResult(int num1, int num2, String choice) {
        // Switch case to evaluate the choice
        switch(choice) {
            case "+": System.out.println("Result after addition is: "+
                (num1+num2));
            break;
            case "-": System.out.println("Result after subtraction is: "+
                (num1-num2));
            break;
            case "*": System.out.println("Result after multiplication is: "+
                (num1*num2));
            break;
            case "/": System.out.println("Result after division is: " + (num1/
                num2));
            break;
        }
    }
}
```

```

    }
}

/** 
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // Check the number of command line arguments
    if(args.length==3) {
        // Use the Integer wrapper to convert String argument to int type
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);
        // Instantiate the Wrappers class
        Wrappers objWrap = new Wrappers();

        // Invoke the calcResult() method
        objWrap.calcResult(num1, num2, args[2]);
    }
    else{
        System.out.println("Usage: num1 num2 operator");
    }
}
}

```

The class **Wrappers** consists of the **calcResult()** method that accepts two numbers and an operator as the parameter. The **main()** method is used to convert the **String** arguments to **int** type by using the **Integer** wrapper class.

Next, the object, **objWrap** of **Wrappers** class is created to invoke the **calcResult()** method with three arguments namely, **num1**, **num2**, and **args[2]** which is the operator specified by the user as the third argument.

To run the class, specify the command line values as 35, 48, and – in the **Arguments** box as shown in Figure 5.14.

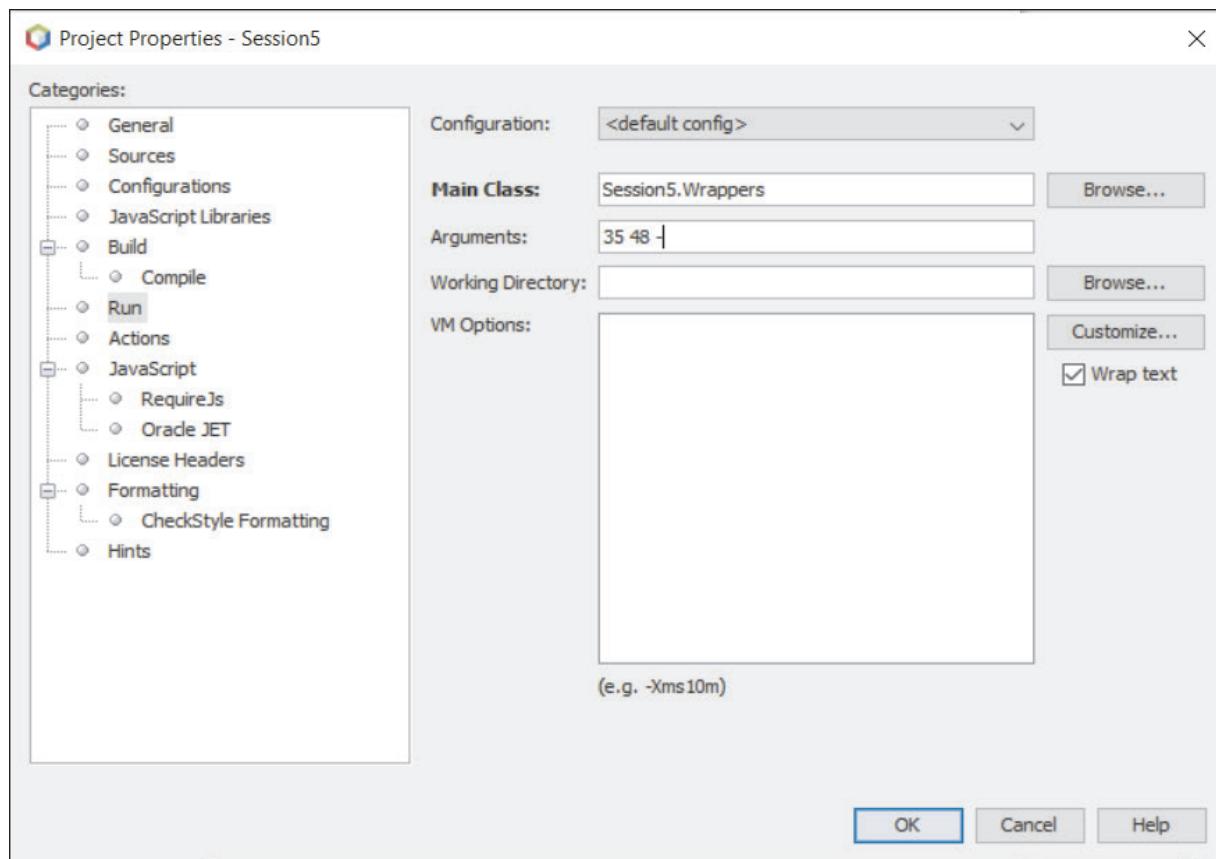


Figure 5.14: Specifying Command Line Arguments

When the program is executed, the first two arguments are stored in the `args[0]` and `args[1]` elements and then, converted to integers. The third argument is stored in the `args[2]` element. It is the operator to be applied on the numbers. The output of the code is shown in Figure 5.15.

```
run:
Result after subtraction is: -13
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 5.15: Output of `Wrappers.java` Class

5.4.1 Autoboxing and Unboxing

Java provides the feature of automatic conversion of primitive data types such as `int`, `float`, and so on to their corresponding object types such as `Integer`, `Float`, and so on during assignments and invocation of methods and constructors. This automatic conversion of primitive types to object types is known as autoboxing.

For example,

```
ArrayList<Integer> intList = new ArrayList<Integer>();
intList.add(10); // autoboxing
Integer y = 20; // autoboxing
```

Similarly, conversion of object types to primitive data types is known as unboxing.

For example,

```
int z = y; // unboxing
```

Autoboxing and unboxing helps a developer to write a cleaner code. Also, using autoboxing and unboxing, one can make use of the methods of wrapper classes as and when required.

Code Snippet 18 demonstrates an example of autoboxing and unboxing.

Code Snippet 18:

```
package session5;
public class AutoUnbox {
    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        Character chBox = 'A'; // Autoboxing a character
        char chUnbox = chBox; // Unboxing a character
        // Print the values
        System.out.println("Character after autoboxing is:" + chBox);
        System.out.println("Character after unboxing is:" + chUnbox);
    }
}
```

The class **AutoUnbox** consists of two variable declarations **chBox** and **chUnbox**. **chBox** is an object type and **chUnbox** is a primitive type of character variable. Figure 5.16 shows the output of the code.

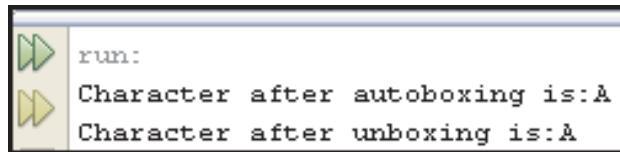


Figure 5.16: Output of AutoUnbox.java Class

Figure 5.16 shows that both primitive as well as object type variable give the same output. However, the variable of type object, that is **chBox**, can take advantage of the methods available with the wrapper class **Character** which are not available with the primitive type **char**.

5.5 Check Your Progress

1. _____ objects are similar to String objects, except that they are mutable.

(A)	StringTokenizer	(C)	StringBuilder
(B)	StringEditor	(D)	StringDesigner

2. Which of the following statements about an array are true?

a.	An array is a container object that can hold a fixed number of values of a single type	c.	After creation of the array, its length becomes fixed
b.	The first index begins with zero and the index of last element is always equal to the length	d.	Values of an array are stored at scattered locations in memory

(A)	a, c	(C)	b, d
(B)	a, d	(D)	b, c

3. Match the following String method Code Snippets with the corresponding outputs with respect to the statement, `String str="Java"`.

	Code Snippet		Output
a.	<code>System.out.println(str.compareTo("World"));</code>	1.	v
b.	<code>System.out.println(str.concat("World"));</code>	2.	-13
c.	<code>System.out.println(str.indexOf("e"));</code>	3.	JavaWorld
d.	<code>System.out.println(str.charAt(2));</code>	4.	-1

(A)	a-3, b-4, c-1, d-2	(C)	a-4, b-3, c-2, d-1
(B)	a-2, b-4, c-3, d-1	(D)	a-2, b-3, c-4, d-1

4. Which of the following options will declare and initialize a single-dimensional array?

(A)	<code>int marks = { 65, 47, 75, 50};</code>	(C)	<code>int[] marks = {"65", "47", "75", "50"};</code>
(B)	<code>int[] marks = {65, 47, 75, 50};</code>	(D)	<code>int[] marks = {65}, {47}, {75}, {50};</code>

5. Consider the following Code Snippet:

```
Integer x = 10; // line 1
int y = x; // line 2
int x1=10;
float z=x1; // line 3
int w = (int)z; // line 4
```

Match the Code Snippet at following lines with the corresponding Java feature.

	Line Number		Output
a.	line 1	1.	Implicit Type Casting
b.	line 2	2.	Autoboxing
c.	line 3	3.	Explicit Type Casting
d.	line 4	4.	Unboxing

(A)	a-3, b-4, c-1, d-2	(C)	a-4, b-3, c-2, d-1
(B)	a-2, b-4, c-1, d-3	(D)	a-2, b-3, c-4, d-1

5.5.1 Answers

1.	C
2.	A
3.	D
4.	B
5.	B

```
g package;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String str = "Hello Java";
        System.out.println(str);
    }
}
```

Summary

- An array is a special data store that can hold a fixed number of values of a single type in contiguous memory locations.
- A single-dimensional array has only one dimension and is visually represented as having a single column with several rows of data.
- A multi-dimensional array in Java is an array whose elements are also arrays.
- A collection is an object that groups multiple elements into a single unit.
- Strings are constant and immutable, that is, their values cannot be changed once they are created.
- StringBuilder objects are similar to String objects, except that they are mutable.
- Java provides a set of classes known as Wrapper classes for each of its primitive data type that 'wrap' the primitive type into an object of that class.
- The automatic conversion of primitive types to object types is known as autoboxing and conversion of object types to primitive types is known as unboxing.

Try It Yourself

```

public class Main {
    public static void main() {
        String pac
        String pr
        String str
    }
}

```

- CompuTech is a well known computer training institute in **Los Angeles, USA**. The institute teaches several computer courses such as Java, .NET, and so on. The institute also organizes coding competitions for students of the institute. As a student of Java course, you have participated in the Java coding competition and you have been assigned the following problem:

Create a Java program to accept employee details for employees of Sales department such as Employee ID, Name, Designation, Salary, and Sales. Based on the employee's sales, calculate the commission according to the rules given in Table 5.6.

Sales	Commission
≥ 10000	30% of basic salary
≥ 8000	30% of basic salary
≥ 6000	20% of basic salary
≥ 4000	10% of basic salary

Table 5.6: Rules for Calculating Commission

Based on the commission, calculate the total salary by adding the commission amount to the basic salary. Display the details of the employee as follows:

Employee ID:

Employee Name:

Designation:

Basic Salary:

Sales Done:

Commission:

Total Salary:

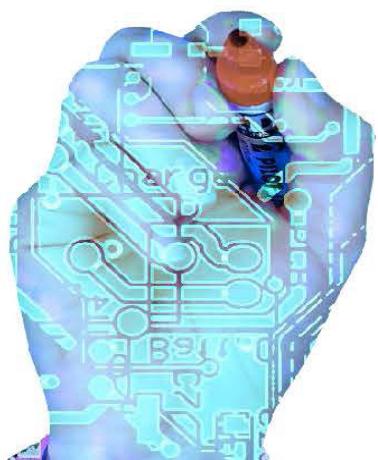


LEARN

BETTER

@

Onlinevarsity



Session - 6

Modifiers and Packages

Welcome to the Session, **Modifiers and Packages**.

This session explains the use of different field and method modifiers in Java with the rules and best practices for using field modifiers. Further, this session describes class variables, methods, and creation and advantages of using packages. Lastly, the session explains the creation of .jar files for deployment.

In this Session, you will learn to:

- Describe field and method modifiers
- Explain different types of modifiers
- Explain rules and best practices for using field modifiers
- Describe class variables
- Explain creation of static variables and methods
- Describe package and its advantages
- Explain creation of user-defined package
- Explain creation of .jar files for deployment



6.1 Introduction

Java is a tightly encapsulated language. This means, that no code can be written outside the class. Not even the `main()` method. However, even within the class, code can become vulnerable to access from external environment. For this purpose, Java provides a set of access specifiers such as `public`, `private`, `protected`, and `default` that help to restrict access to class and class members. However, at times, it is required to further restrict access to the members of a class to prevent modification by unauthorized code. Java provides additional field and method modifiers for this purpose.

Also, in some cases, it may be required to have a common data member that can be shared by all objects of a class as well as other classes. Java provides the concept of class variables and methods to solve this purpose. Classes sharing common attributes and behavior can also be grouped together for better understanding of the application. Java provides packages that can be used to group related classes. Also, the entire set of packages can be combined into a single file called the `.jar` file for deployment on the target system.

6.2 Field and Method Modifiers

Field and method modifiers are keywords used to identify fields and methods that provide controlled access to users. Some of these can be used in conjunction with access specifiers such as `public` and `protected`. Different field modifiers that can be used are as follows:

- `volatile`
- `native`
- `transient`
- `final`

6.2.1 `volatile` Modifier

The `volatile` modifier allows the content of a variable to be synchronized across all running threads. A thread is an independent path of execution of code within a program. Many threads can run concurrently within a program. The `volatile` modifier is applied only to fields. Constructors, methods, classes, and interfaces cannot use this modifier. The `volatile` modifier is not frequently used.

While working with a multithreaded program, the `volatile` keyword is used. When multiple threads of a program are using the same variable, in general, each thread has its own copy of that variable in the local cache. In such a case, if the value of the variable is updated, it updates the copy in the local cache and not the main variable present in the memory. The other thread using the same variable does not get the updated value.

To avoid this problem, a variable is declared as `volatile` to indicate that it will not be stored in the local cache.

Also, whenever a thread updates the values of the variable, it updates the variable present in the main memory. This helps other threads to access the updated value.

For example,

```
private volatile int testValue; // volatile variable
```

6.2.2 native Modifier

In some cases, it is required to use a method in Java program that resides outside JVM. For this purpose, Java provides the `native` modifier. The `native` modifier is used only with methods. It indicates that the implementation of the method is in a language other than Java such as C or C++. Constructors, fields, classes, and interfaces cannot use this modifier. 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. In case of the `native` modifier, the implementation of the method exists in a library outside the JVM. Before invoking a native method, the library that contains the method implementation must be loaded by making following system call:

```
System.loadLibrary("libraryName");
```

To declare a native method, the method is preceded with the `native` modifier. Also, the implementation is not provided for the method. For example,

```
public native void nativeMethod();
```

After declaring a native method, a complex series of steps are used to link it with the Java code.

Code Snippet 1 demonstrates an example of loading a library named `NativeMethodDefinition` containing a native method named `nativeMethod()`.

Code Snippet 1:

```
class NativeModifier {
    native void nativeMethod(); // declaration of a native method
    /**
     * static code block to load the library
     */
    static {
        System.loadLibrary("NativeMethodDefinition");
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        NativeModifier objNative = new NativeModifier(); // line1
        objNative.nativeMethod(); // line2
    }
}
```

Notice that a static code block is used to load the library. Here, the `static` keyword indicates that the library is loaded as soon as the class is loaded. This ensures that the library is available when the call to the native method is made. The native method can be used in the same way as a non-native method. For example, to invoke the native method `nativeMethod()`, one can write the code as shown in line1 and line2 of Code Snippet 1.

Native methods allow access to existing library routines created outside the JVM. However, the use of native methods introduces two major problems. They are as follows:

- **Impending security risk:** The native method executes actual machine code and therefore, it can gain access to any part of the host system. That is, the native code is not restricted to the JVM execution environment. This may lead to a virus infection on the target system.
- **Loss of portability:** The native code is bundled in a Dynamic Link Library (DLL), so that it can be loaded on the machine on which the Java program is executing. Each native method is dependent on the Central Processing Unit (CPU) and the OS. This makes the DLL inherently non-portable. This means, that a Java application using native methods will run only on a machine in which a compatible DLL has been installed.

6.2.3 *transient* Modifier

When a Java application is executed, the objects are loaded in the Random Access Memory (RAM). However, objects can also be stored in a persistent storage outside the JVM so that it can be used later. This determines the scope and life span of an object. The process of storing an object in a persistent storage is called serialization. For any object to be serialized, the class must implement the `Serializable` interface.

However, if `transient` modifier is used with a variable, it will not be stored and will not become part of the object's persistent state. The `transient` modifier is useful to prevent security sensitive data from being copied to a source in which no security mechanism has been implemented. Also, `transient` modifier reduces the amount of data being serialized, improves performance, and reduces costs.

The `transient` modifier can only be used with instance variables. It informs the JVM not to store the variable when the object, in which it is declared, is serialized. Thus, when the object is stored in persistent storage, the instance variable declared as `transient` is not persisted. Code Snippet 2 depicts the creation of a `transient` variable.

Code Snippet 2:

```
class Circle {
    transient float PI; // transient variable that will not persist
    float area; // instance variable that will persist
    ...
}
```

6.2.4 *final* Modifier

The `final` modifier is used when modification of a class or data member is to be restricted. The `final` modifier can be used with a variable, method, and class.

A variable declared as `final` is a constant whose value cannot be modified. A `final` variable is 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 declaration. Code Snippet 3 shows the creation of a `final` variable.

Code Snippet 3:

```
final float PI = 3.14;
```

The variable `PI` is declared `final` so that its value cannot be changed later.

A method declared `final` cannot be overridden or hidden in a Java subclass. The reason for using a `final` method is to prevent subclasses from changing the meaning of the method and increase the efficiency of code by allowing the compiler to turn method calls into inline Java code. A `final` method is commonly used to generate a random constant in a mathematical application. Code Snippet 4 depicts the creation of a `final` method.

Code Snippet 4:

```
final float getCommission(float sales) {
    System.out.println("A final method... ");
}
```

The method `getCommission()` can be used to calculate commission based on monthly sales. The implementation of the method cannot be modified by other classes as it is declared as `final`. A `final` method cannot be declared `abstract` as it cannot be overridden.

Note - `abstract` keyword is used with a method and class. An `abstract` method cannot have a body and an `abstract` class cannot be instantiated and must be inherited or subclassed.

A class declared `final` cannot be inherited or subclassed. Such a class becomes a standard and must be used as it is. The variables and methods of a class declared `final` are also implicitly `final`. The reason for declaring a class as `final` is to limit extensibility and to prevent the modification of the class definition. Code Snippet 5 shows the creation of a `final` class.

Code Snippet 5:

```
public final class Stock {
    ...
}
```

The class **Stock** is declared **final**. All data members within this class are implicitly **final** and cannot be modified by other classes.

Code Snippet 6 demonstrates an example of creation of a **final** class.

Code Snippet 6:

```
public class FinalDemo {
    // Declare and initialize a final variable
    final float PI = 3.14F; // variable to store value of PI

    /**
     * Displays the value of PI
     *
     * @param pi a float variable storing the value of PI
     * @return void
     */
    public void display(float pi) {
        PI=pi; // generates compilation error
        System.out.println("The value of PI is:"+PI);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the FinalDemo class
        final FinalDemo objFinalDemo = new FinalDemo();
        // Invoke the display() method
        objFinalDemo.display(22.7F);
    }
}
```

The class **FinalDemo** consists of a **final** float variable **PI** set to **3.14**. The method **display()** is used to set a new value passed by the user to **PI**. However, this leads to compilation error '**cannot assign a value to final variable PI**'.

If the user chooses to run the program anyway, a runtime error is issued as shown in Figure 6.1.

```
run:
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - cannot assign a value to final
variable PI
    at FinalDemo.display(FinalDemo.java:12)
    at FinalDemo.main(FinalDemo.java:22)
Java Result: 1
BUILD SUCCESSFUL (total time: 2 seconds)
```

Figure 6.1: RuntimeException Generated

To remove the error, the method signature should be changed and the statement, `PI = pi;` should be removed.

6.2.5 Rules and Best Practices 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.
- A transient variable may not be declared as final or static.

6.3 Class Variables

Consider a situation, where in a user wants to create a counter that keeps track of the number of objects accessing a particular method. If the user creates an instance variable named `counter`, each object will have a separate copy of the variable, `counter`. Due to this, the user cannot keep track of the number of times a method is accessed. In such a scenario, a variable is required that can be shared among all the objects of a class and any changes made to the variable are updated only in one common copy. Java provides implementation of such a concept of class variables by using the `static` keyword.

6.3.1 Declaring Class Variables

Class variables are also known as `static` variables. Note that the `static` variables are not constants. Such variables are associated with the class rather than any object. In other words, all instances of the class share the same value of the class variable. The value of a `static` variable can be modified using class methods or instance methods. However, unlike instance variable, there exists only one copy of a class variable for all objects in one fixed location in memory. However, a `static` variable declared as `final` becomes a constant whose value cannot be modified.

For example,

```
static int PI=3.14; // static variable that can be modified
static final int PI=3.14; // static constant that cannot be modified
```

6.3.2 Creating Static Variables, Static Methods, and Static Blocks

One can also create `static` methods and `static` initializer blocks along with `static` variables. `Static` variables and methods can be manipulated without creating an instance of the class. This is because there is only one copy of a `static` data member that is shared by all objects. A `static` method can only access static variables and not instance variables.

Methods declared as `static` have following restrictions:

- Can invoke only `static` methods.
- Can access only `static` data.
- Cannot use `this` or `super` keywords.

A `static` block is used to initialize `static` variables as soon as the class is launched. They are used when a block of code must be executed during loading of the class by JVM. It is enclosed within {} braces.

Generally, a constructor is used to initialize variables. It is the best approach as the constructor is invoked implicitly when an object is created. However, a programmer sometime must create objects before anything can be done in a program. This is because, an object is required to call an instance variable or method. However, instead of a constructor, a `static` block can be used to initialize `static` variables because `static` block is executed even before the `main()` method is executed. That is, the execution of Java code starts from `static` blocks and not from `main()` method. There can be more than one `static` block in a program. They can be placed anywhere in the class. A `static` initialization block can reference only those class variables that have been declared before it.

Code Snippet 7 demonstrates an example of `static` variables, `static` method, and `static` block.

Code Snippet 7:

```
package session 6;
public class StaticMembers {
    // Declare and initialize static variable
    public static int staticCounter = 0;
    // Declare and initialize instance variable
    int instanceCounter = 0;
    /**
     * static block
     *
     */
    static{
        System.out.println("I am a static block");
    }
    /**
     * Static method
     *
     * @return void
    }
```

```
/*
public static void staticMethod() {
    System.out.println("I am a static method");
}

/**
 * Displays the value of static and instance counters
 *
 * @return void
 */
public void displayCount() {
    //Increment the static and instance variable
    staticCounter++;
    instanceCounter++;
    // Print the value of static and instance variable
    System.out.println("Static counter is:"+ staticCounter);
    System.out.println("Instance counter is:"+ instanceCounter);
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    System.out.println("I am the main method");
    // Invoke the static method using the class name
    StaticMembers.staticMethod();
    // Create first instance of the class
    StaticMembers objStatic1 = new StaticMembers();
    objStatic1.displayCount();
    // Create second instance of the class
    StaticMembers objStatic2 = new StaticMembers();
    objStatic2.displayCount();
    // Create third instance of the class
    StaticMembers objStatic3 = new StaticMembers();
    objStatic3.displayCount();
}
}
```

The class **StaticMembers** consists of two variables, a `static` variable and an instance variable. Also, a `static` block and a `static` method have been created. The instance method `displayCount()` is used to increment the value of the `static` and instance counters by 1 and then, display the resulting value of both variables. In the `main()` method, the `static` method is invoked in line 1 using the class name `StaticMembers.staticMethod()` instead of creating an object of the class. Next, three objects of the class **StaticMembers** are created namely `objStatic1`, `objStatic2`, and `objStatic3`. Each object is used to invoke the `displayCount()` method.

Figure 6.2 shows the output of the program.

```

run:
I am a static block
I am the main method
I am a static method
Static counter is:1
Instance counter is:1
Static counter is:2
Instance counter is:1
Static counter is:3
Instance counter is:1

```

Figure 6.2: Output of `StaticMembers.java` Class

From Figure 6.2 it is clear that the static block is executed even before the `main()` method. Also, the value of static counter is incremented to 1, 2, 3, ..., and so on whereas the value of instance counter remains 1 for all the objects. This is because a separate copy of the instance counter exists for each object. Hence, every time a new object is created, the count is incremented from an initial value of 0 to 1 when the `displayCount()` method is invoked.

However, for the `static` variable, only one copy exists per class. Every object increments the same copy of the variable, `staticCounter`. Therefore, when the first object increments the value of the variable `staticCounter`, it is set to 1. Then, the second object increments it by 1 and the value is set to 2, and so on.

Thus, by using a static counter, a user can keep track of the number of instances of a class.

6.4 Packages

Consider a situation where in a user has about fifty files of which some are related to sales, others are related to accounts, and some are related to inventory. Also, the files belong to different years. All these files are kept in one section of a cupboard. Now, when a particular file is required, the user has to search the entire cupboard.

This is very time consuming and difficult. For this purpose, the user creates separate folders and divides the files according to the years and further groups them according to the content as shown in Figure 6.3.

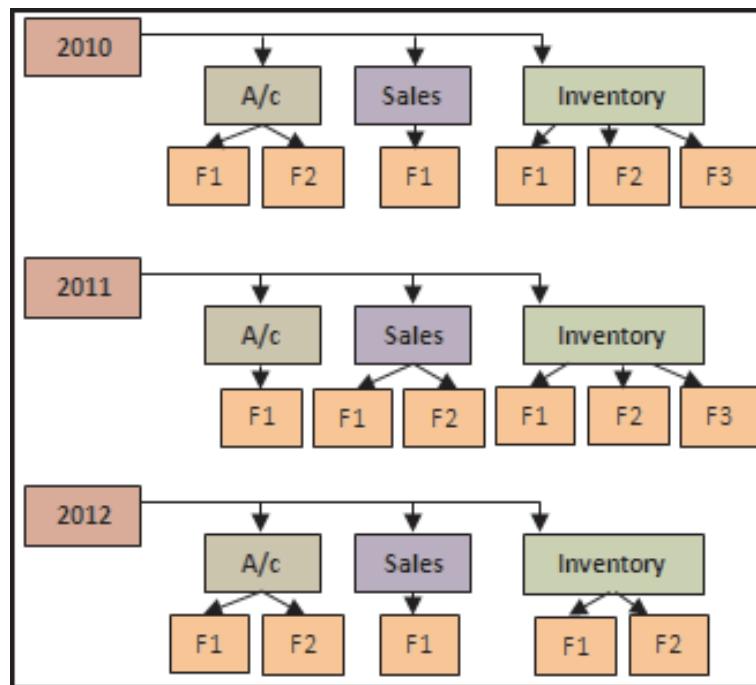


Figure 6.3: Files Organized in Specific Folders

Similarly, in Java, one can organize the files using packages. A package is a namespace that groups related classes and interfaces and organizes them as a unit. Conceptually, one can think of packages as being similar to different folders created on a computer to store files. For example, one can keep source files in one folder, images in another, and executables in yet another folder. Software written in Java is composed of several classes. Therefore, it is advisable to keep the classes organized by placing related classes and interfaces into packages.

Packages have following features:

- 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 within a package in a single Java source file, 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`.

6.4.1 Advantages of Using Packages

One should group the related classes and interfaces in a package for following reasons:

- One can easily determine that these classes are related.
- One can know where to find the required type that can provide the required functions.
- The names of classes of one package would not conflict with the class names in other packages as the package creates a new namespace. For example, `myPackage1.Sample` and `myPackage2.Sample`.
- One can allow classes within one package to have unrestricted access to one another while restricting access to classes outside the package.
- Packages can also store hidden classes that can be used within the package, but are not visible or accessible outside the package.
- Packages can also have classes with data members that are visible to other classes, but not accessible outside the package.
- When a program from a package is called for the first time, the entire package gets loaded into the memory. Due to this, subsequent calls to related subprograms of the same package do not require any further disk Input/Output (I/O).

6.4.2 Types of Packages

The Java platform comes with a huge class library which is a set of packages. These classes can be used in applications by including the packages in a class. This library is known as the Application Programming Interface (API). The packages of the API represent the most common tasks associated with general-purpose programming. Every Java application or applet has access to the core package in the API, the `java.lang` package.

For example, the `String` class stores the state and behavior related to character strings; the `File` class allows the developer to create, delete, compare, inspect, or modify a file on the file system. Similarly, the `Socket` class allows the developer to create and use network sockets, and various Graphical User Interface (GUI) control classes such as `Button`, `Checkbox`, and so on provide ready to use GUI controls. There are thousands of such classes to select and use. These ready to use classes allow a programmer to focus on the design of the application rather than on the infrastructure required to make it work.

Different types of Java packages are as follows:

- Predefined packages
- User-defined packages

Predefined packages are part of the Java API. Predefined packages that are commonly used are as follows:

- java.io
- java.util
- java.awt

User-defined packages are created by the developers. To create user-defined packages, perform following steps:

1. Select an appropriate name for the package by using following naming conventions:
 - Package names are usually written in lower case to avoid conflict with the names of classes or interfaces.
 - Companies usually attach their reversed Internet domain name as a prefix to their package names. For example, `com.sample.mypkg` for a package named `mypkg` created by a programmer at `sample.com`.
 - Naming conflicts occurring in the projects of a single company are handled according to the naming conventions specific to that company. This is done usually by including the region name or the project name after the company name. For example, `com.sample.myregion.mypkg`.
 - Package names should not begin with `java` or `javax` as they are used for packages that are part of Java API.
 - In certain cases, the Internet domain name may not be a valid package name. For instance, when the domain name includes special characters such as hyphens. Alternatively, the package name employs a reserved Java keyword such as "char." Another case is if the package name commences with a digit or another prohibited character, which violates Java package naming conventions. In such a case, it is advisable to use an underscore as shown in Table 6.1.

Domain Name	Suggested Package Name
sample-name.sample.org	org.sample.sample_name
sample.int	int_.sample
007name.sample.com	com.sample._007name

Table 6.1: Package Names

2. Create a folder with the same name as the package.
3. Place the source files in the folder created for the package.

4. Add the package statement as the first line in all the source files under that package as depicted in Code Snippet 8. Note that there can only be one package statement in a source file.

Code Snippet 8:

```
package session6;
class StaticMembers{
    public static void main(String[] args)
    {}
}
```

5. Save the source file **StaticMembers.java** in the package **session6**.

6. Compile the code as follows:

javac StaticMembers.java

OR

Compile the code with **-d** option as follows:

javac -d . StaticMembers.java

where, **-d** stands for directory and **'.'** stands for current directory. The command will create a sub-folder named **session6** and store the compiled class file inside it.

7. From the parent folder of the source file, execute it using the fully qualified name as follows:

java session6.StaticMembers

Note - The **CLASSPATH** variable must be set to the source file directory that has the **StaticDemo.java** file before executing the program. For example, **set classpath="D:\session6"** or **CLASSPATH** can be specified while executing. For example, **java -cp "D:\session6" session6.StaticDemo**.

Java allows the user to import the classes from predefined as well as user-defined packages using an **import** statement. For example, one can use the **StaticMembers** class created in Code Snippet 7 as well as the built-in class **ArrayList** of the **java.util** package by importing them into another class belonging to another package. However, the access specifiers associated with the class members will determine if the class members can be accessed by a class of another package.

A member of a public class can be accessed outside the package by doing any of following:

- Referring to the member class by its fully qualified name, that is, **package-name.class-name**.
- Importing the package member, that is, **import package-name.class-name**.
- Importing the entire package, that is, **import package-name.***.

To create a new package using NetBeans IDE, perform following steps:

1. Open the project in which the package is to be created. In this case, **Session6** project has been chosen.
2. Right-click **Source Packages** → **New** → **Java Package** to display the **New Java Package** dialog box. For example, in Figure 6.4, the project **Session6** is opened and the **Java Package** option is selected.

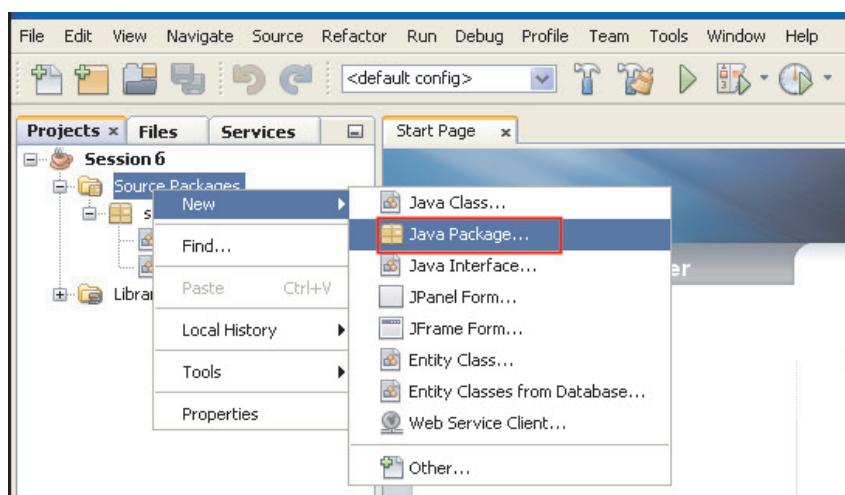


Figure 6.4: Creating a New Java Package

3. Type **userpkg** in the **Package Name** box of the **New Java Package** dialog box that is displayed.
4. Click **Finish**. The **userpkg** package is created as shown in Figure 6.5.

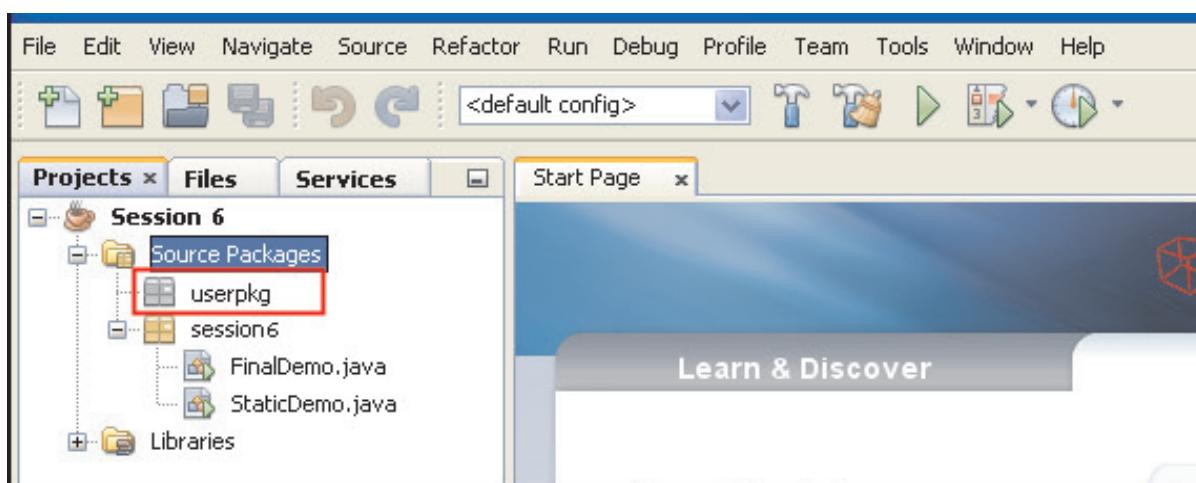


Figure 6.5: userpkg Package Created

5. Right-click **userpkg** and select **New** → **Java Class** to add a new class to the package.

6. Type **UserClass** as the **Class Name** box of the **New Java Class** dialog box and click **Finish**.

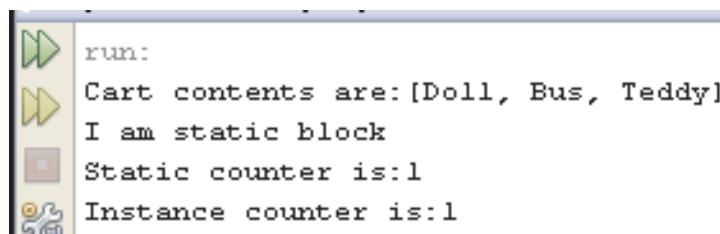
7. Type the code in the class as depicted in Code Snippet 9.

Code Snippet 9:

```
package userpkg;
// Import the predefined and user-defined packages
import java.util.ArrayList;
import session6.StaticMembers;
public class UserClass {
    // Instantiate ArrayList class of java.util package
    ArrayList myCart = new ArrayList(); // line 1
    /**
     * Initializes an ArrayList
     *
     * @return void
     */
    public void createList() {
        // Add values to the list
        myCart.add("Doll");
        myCart.add("Bus");
        myCart.add("Teddy");
        // Print the list
        System.out.println("Cart contents are:" + myCart);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the UserClass class
        UserClass objUser = new UserClass();
        objUser.createList(); // Invoke the createList() method
        // Instantiate the StaticMembers class
        StaticMembers objStatic = new StaticMembers();
        objStatic.displayCount(); // Invoke the displayCount() method
    }
}
```

The class `UserClass` is defined within the package, `userpkg`. The two import statements namely, `java.util.ArrayList` and `session6.StaticMembers` are used to import the packages `java.util` and `session6` into the `UserClass` class. However, only `ArrayList` and `StaticMembers` classes are imported from the respective packages. To import all classes of the packages, one must write the statements `import java.util.*` and `import session6.*` respectively.

Next, the `ArrayList` is instantiated in line 1 and the `createList()` method is used to initialize and display the list. In the `main()` method, the object of `UserClass` is created to invoke the `createList()` method and the object of `StaticMembers` class is created to invoke the `displayCount()` method. Figure 6.6 shows the output of the program.



```

run:
Cart contents are:[Doll, Bus, Teddy]
I am static block
Static counter is:1
Instance counter is:1

```

Figure 6.6: Output of `UserClass.java` Class

Notice that the output shows the execution of the `static` block of `StaticMembers` class also. This is because the `static` block is executed as soon as the class is launched.

6.4.3 Creating .jar Files for Deployment

All the source files of a Java application are bundled into a single archive file called the Java Archive (JAR). The `.jar` file contains the class files and additional resources associated with the application.

The `.jar` file format provides several advantages as follows:

- **Security:** The `.jar` file can be digitally signed so that only those users who recognize your signature can optionally grant the software security privileges that the software might not otherwise have.
- **Decrease in Download Time:** The source files bundled in a `.jar` file can be downloaded to a browser in a single HTTP transaction without having to open a new connection for each file.
- **File Compression:** The `.jar` format compresses the files for efficient storage.
- **Packaging for Extensions:** The extension framework in Java allows adding additional functionality to the Java core platform. The `.jar` file format defines the packaging for extensions. The `.jar` file format allows converting the software into extensions as well.
- **Package Sealing:** Java provides an option to seal the packages stored in the `.jar` files so that the packages can enforce version consistency. When a package is sealed within a `.jar` file, it implies that all classes defined in that package must be available in the same `.jar` file.
- **Package Versioning:** A `.jar` file can also store additional information about the files, such as

vendor and version information.

- **Portability:** The .jar files are packaged in a ZIP file format. This enables the user to use them for tasks such as lossless data compression, decompression, archiving, and archive unpacking.

To perform basic tasks with .jar files, one can use the Java Archive Tool. This tool is provided with the JDK. The Java Archive Tool is invoked by using the jar command. The basic syntax for creating a .jar file is as follows:

Syntax:

```
jar cf jar-file-name input-file-name(s)
```

where,

c: indicates that the user wants to create a .jar file.

f: indicates that the output should go to a file instead of stdout.

jar-file-name: represents the name of the resulting .jar file. Any name can be used for a .jar file. The .jar extension is provided with the file name, though it is not required.

input-file-name (s): represents a list of one or more files to be included in the .jar separated by a space. This argument can contain the wildcard symbol '*' as well. If any of the input files specified is a directory, the contents of that directory are added to the .jar recursively.

Options c and f can be used in any order, but without any space in between. The jar command generates a compressed .jar file and places it by default in the current directory. Also, it will generate a default manifest file for the .jar file. The metadata in the .jar file such as entry names, contents of the manifest, and comments must be encoded in UTF8.

Note - UTF 8 is an encoding standard that represent every character in the Unicode character set.

Some other options, apart from cf, available with the jar command are listed in Table 6.2.

Option	Description
v	Produces VERBOSE output on stdout while the .jar is begin built. The output displays the name of each of the files that are included in the .jar file.
0 (zero)	Indicates that the .jar file must not be compressed.
M	Indicates that the default manifest file must not be created.
m	Allows inclusion of manifest information from an existing manifest file. jar cmf existing-manifest-name jar-file-name input-file-name(s)
-c	Used to change directories during execution of the command.

Table 6.2: jar Command Options

When a .jar file is created, the time of creation is stored in the .jar file. Therefore, even if the contents of the .jar file are not changed, if the .jar file is created multiple times, the resulting files will not be exactly identical. For this reason, it is advisable to use versioning information in the manifest file instead

of creation time, to control versions of a .jar file.

For example, consider following files of a simple **BouncingBall** game application as shown in Figure 6.7.

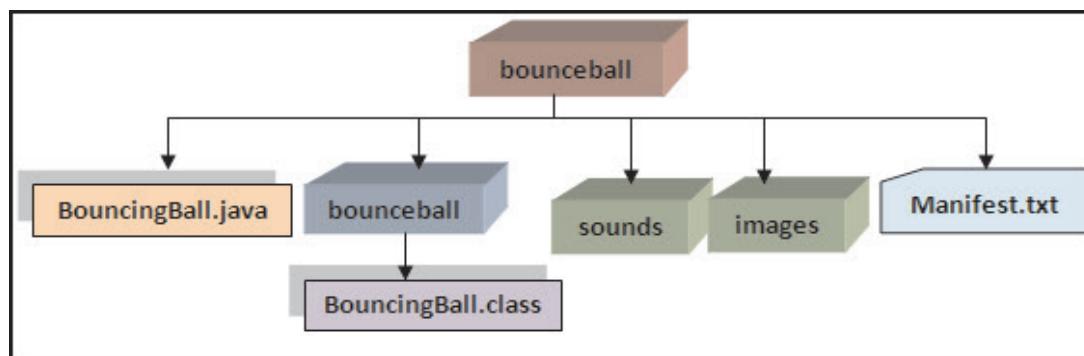


Figure 6.7: BouncingBall Application

Figure 6.7 displays BouncingBall app, including source & class files, sound & image folders with files for application use.

To create a .jar of the application using command line, perform following steps:

1. Create the directory structure as shown in Figure 6.7.
2. Create a text file with the code depicted in Code Snippet 10.

Code Snippet 10:

```

package bounceball;
public class BouncingBall {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("This is the bouncing ball game");
    }
}
  
```

3. Save the file as **BouncingBall.java** in the source package **bounceball**.
4. Compile the .java file at command prompt by writing following command:
javac -d . BouncingBall.java

The command will create a subfolder with the same name **bounceball** and store the class file **BouncingBall.java** in that directory as shown in Figure 6.7.

Note - The PATH variable must be set before executing the command. For example, `E:\bouncingball\set path="C:\Program Files\Java\jdk20\bin"`. If the class file must be stored in some other folder, use the `-cp` option to specify the CLASSPATH for the class files.

5. Create a text file with the Main-class attribute as shown in Figure 6.8.

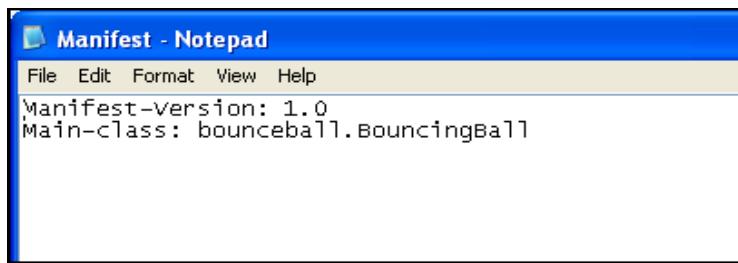


Figure 6.8: Manifest.txt File

6. Save the file as **Manifest.txt** in the source **bounceball** folder as shown in Figure 6.7. The **Manifest.txt** file will be referred by the Jar tool for the Main class during **.jar** creation. This will inform the Jar tool about the starting point of the application.
7. To package the application in a single **.jar** named **BouncingBall.jar**, write following command:
`jar cvmf Manifest.txt bounceball.jar bounceball/BouncingBall.class sounds images`

Note - The name of the manifest and the **.jar** file must be in the same order as the options **m** and **f**.

This will create a **bounceball.jar** file in the source folder as shown in Figure 6.9.

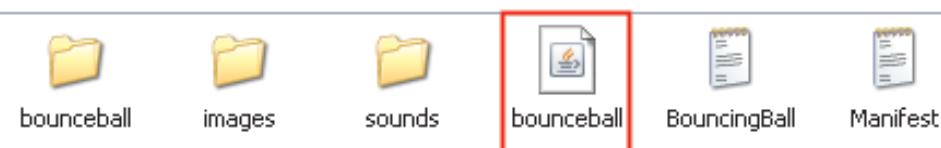


Figure 6.9: bounceball.jar File Created in Source Folder

The command options `cvmf` indicate that the user wants to create a **.jar** file with verbose output using the existing manifest file, **Manifest.txt**. The name of the output file is specified as **bounceball.jar** instead of `stdout`. Additionally, the class file "BouncingBall.class" is located in "bounceball/" and includes the `sounds` and `images` directories, ensuring their inclusion in the **.jar** file.

8. To execute the .jar file at command prompt, type following command:

```
java -jar bounceball.jar
```

The command will execute the main() class of the .jar file and print following output:

This is the bouncing ball game.

Figure 6.10 shows entire series of steps for creation of .jar file with verbose output and final output after .jar file execution.

```
C:\WINDOWS\system32\cmd.exe
E:\bounceball>set path="C:\Program Files\Java\jdk20"
E:\bounceball>javac -d . BouncingBall.java
E:\bounceball>jar cvmf Manifest.txt bounceball.jar bounceball/BouncingBall.class
adding: sounds/
adding: images/
added manifest
adding: BouncingBall.class<(in = 459) <out= 307><deflated 33%>
adding: sounds/<(in = 0) <out= 0><stored 0%>
adding: images/<(in = 0) <out= 0><stored 0%>
E:\bounceball>java -jar bounceball.jar
This is the bouncing ball game
E:\bounceball>
```

Figure 6.10: Steps for Creation and Execution of .jar File

Since **sounds** and **images** are directories, the Jar tool will recursively place the contents in the .jar file. The resulting .jar file **BouncingBall.jar** will be placed in the current directory. Also, use of the option '**v**' will show verbose output of all files that are included in the .jar file.

In the example, the files and directories in the archive retained their relative path names and directory structure. However, one can use the **-c** option to create a .jar file in which the relative paths of the archived files will not be preserved.

For example, suppose one wants to put sound files and images used by the **BouncingBall** program into a .jar file and that all the files should be on the top level, with no directory hierarchy. One can accomplish this by executing following command from the parent directory of the **sounds** and **images** directories:

```
jar cf SoundImages.jar -c sounds . -c images .
```

Here, '**-c sounds**' directs the Jar tool to the **sounds** directory and the **'..'** following '**-c sounds**' directs the Jar tool to archive all the contents of that directory. Similarly, '**-c images .**' performs the same task with the **images** directory.

The resulting .jar file would consist of all the sound and image files of the **sounds** and **images** folder as follows:

```
META-INF/MANIFEST.MF
```

```
beep.au
```

```
failure.au
```

```
ping.au
```

```
success.au
```

```
greenball.gif
```

```
redball.gif
```

```
table.gif
```

However, if following command is used without the **-c** option:

```
jar cf SoundImages.jar sounds images
```

The resulting .jar file would have following contents:

```
META-INF/MANIFEST.MF
```

```
sounds/beep.au
```

```
sounds/failure.au
```

```
sounds/ping.au
```

```
sounds/success.au
```

```
images/greenball.gif
```

```
images/redball.gif
```

```
images/table.gif
```

Table 6.3 shows a list of frequently used **jar** command options.

Task	Command
To create a .jar file	<code>jar cf jar-file-name input-file-name(s)</code>
To view contents of a .jar file	<code>jar tf jar-file-name</code>
To extract contents of a .jar file	<code>jar xf jar-file-name</code>
To run the application packaged into the .jar file. Manifest.txt file is required with Main-class header attribute.	<code>java -jar jar-file-name</code>

Table 6.3: Frequently Used **jar** Command Options

To create a .jar file of the application using NetBeans IDE, perform following steps:

1. Create a new package **bounceball** in the **Session6** application.
2. Create a new java class named **BouncingBall.java** within the **bounceball** package.
3. Type the code depicted in Code Snippet 11 in the **BouncingBall** class.

Code Snippet 11:

```
package bounceball;
public class BouncingBall {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("This is the bouncing ball game");
    }
}
```

4. Set **BouncingBall.java** as the main class in the **Run** properties of the application.
5. Run the application by clicking the **Run** icon on the toolbar. The output will be shown in the **Output** window.
6. To create a .jar file, right-click the **Session6** application and select **Clean and Build** option as shown in Figure 6.11.

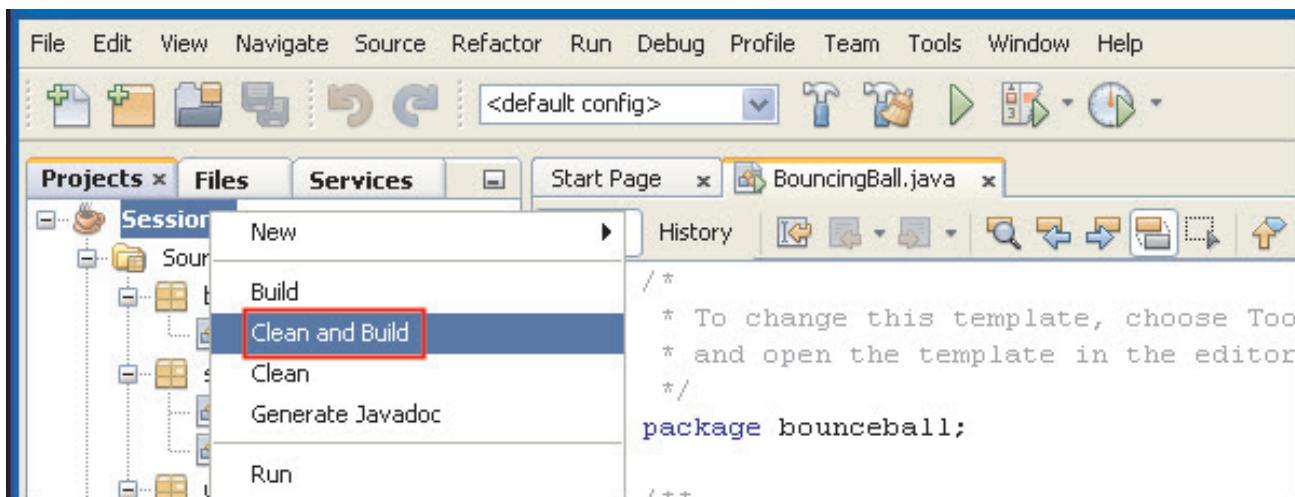


Figure 6.11: Clean and Build an Application

7. The IDE will build the application and generate the .jar file. A message as shown in Figure 6.12 will be displayed to the user in the status bar, once .jar file generation is finished.

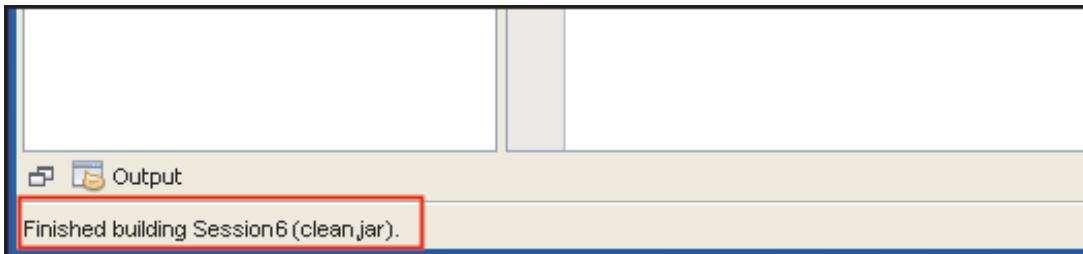


Figure 6.12: Status Message After .jar Generation

The **Clean and Build** command creates a new **dist** folder into the application folder and places the .jar file into it as shown in Figure 6.13.



Figure 6.13: Session6.jar File

User can load this .jar file in any device that has a JVM and execute it by double-clicking the file or run it at command prompt by writing following command:

```
java -jar Session6.jar
```

Note - If double-click does not work, right-click the .jar file and select **Open with**. Select the **Java(TM) Platform SE Binary** as the default program for .jar files.

6.5 Check Your Progress

1. Every Java application or applet has access to the _____ core package in the Java API.

(A)	java.util	(C)	java.io
(B)	java.lang	(D)	java.awt

2. Which of the following statements about the `final` modifier are true?

a.	The <code>final</code> modifier is used when modification of a class or data member is to be restricted	c.	A class declared <code>final</code> must be inherited or subclassed
b.	A method declared <code>final</code> must be overridden	d.	A variable declared <code>final</code> is a constant whose value cannot be modified

(A)	b, c	(C)	a, c
(B)	a, d	(D)	b, d

3. Match following method modifiers with the corresponding description.

	Modifier		Description
a.	<code>volatile</code>	1.	Used with a variable, method, and class
b.	<code>native</code>	2.	Applied only to fields
c.	<code>final</code>	3.	Used only with instance variables
d.	<code>transient</code>	4.	Used only with methods

(A)	a-3, b-2, c-4, d-1	(C)	a-2, b-4, c-1, d-3
(B)	a-2, b-3, c-4, d-1	(D)	a-3, b-4, c-1, d-2

4. Which of the following option represents the correct command for generating a `.jar` file using an existing manifest file and displaying a verbose output? (**Assumption:** jar file - `MyJar.jar`, Manifest file - `Manifest.txt`, class file - `MyClass.class`, package - `mypkg`).

(A)	<code>jar cvfm Manifest.txt MyJar.jar mypkg/MyClass.class</code>
(B)	<code>jar cfvm Manifest.txt MyJar.jar mypkg/MyClass.class</code>
(C)	<code>jar cvmf Manifest.txt MyJar.jar mypkg/MyClass.class</code>
(D)	<code>jar cmf Manifest.txt MyJar.jar mypkg/MyClass.class</code>

5. Consider following code to check the number of users logged in to a system:

```
package myPkg;

public class UserLogin{
    public int userNo=0; // line 1

    public void displayUserCount () {
        userNo++;

        System.out.println("User number is:"+ userNo);

    }

    public static void main(String[] args) {
        UserLogin objUser1 = new UserLogin();

        objUser1.displayUserCount ();

        UserLogin objUser2 = new UserLogin();

        objUser2.displayUserCount ();

    }
}
```

The code is executing properly without any error. However, the value of variable, **userNo** is displaying **1** for every new object that is created. What change must be made in line 1 to ensure that the **userNo** is incremented when a new object is created? (**Assumption:** The program is not using multiple threads.)

(A)	public transient int userNo=0;	(C)	public volatile int userNo=0;
(B)	public final int userNo=0;	(D)	public static int userNo=0;

6. Identify the correct command to execute the .jar file **myJar.jar** at the command prompt.

(A)	jar myJar.jar	(C)	javac -jar myJar.jar
(B)	java -jar myJar.jar	(D)	java -jar myJar

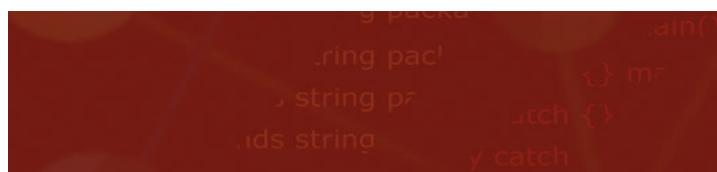
6.5.1 Answers

1.	B
2.	B
3.	C
4.	C
5.	D
6.	B

Summary

- Field and method modifiers are used to identify fields and methods that have controlled access to users.
- The volatile modifier allows the content of a variable to be synchronized across all running threads.
- A thread is an independent path of execution within a program.
- The native modifier indicates that the implementation of the method is in a language other than Java such as C or C++.
- The transient modifier can only be used with instance variables. It informs the JVM not to store the variable when the object, in which it is declared, is serialized.
- The final modifier is used when modification of a class or data member is to be restricted.
- Class variables are also known as static variables and there exists only one copy of that variable for all objects.
- A package is a namespace that groups related classes and interfaces and organizes them as a unit.
- All the source files of a Java application are bundled into a single archive file called the Java Archive (JAR).

Try It Yourself



1. **SmartDesigns** is a well-known company of architects situated in **Chicago, Illinois**. Currently, the management is hiring IT professionals to automate the tasks performed in the company such as drawing, animation, and calculation of dimensions by developing a software application in Java. For this purpose, the company has organized technical interviews for aspiring developers. As one of the candidates, you have been assigned following tasks:

- Create a program to calculate area of shapes such as circle and square.
- Specify an appropriate package name to store the source file and class file.
- The value for shape must be taken from the user along with the value for radius or length according to the shape.
- If a user does not provide any value, appropriate message must be displayed to the user.
- If a user specifies necessary values, calculate the area of the specified shape.
- Display the output as follows:

User Number:

Shape:

Value of radius/length:

Calculated Area:

- The user number must be incremented with every user that runs the program.
- Package the files into a .jar file. (Either by using command prompt or by NetBeans IDE).
- Execute the .jar file at command prompt.

Session - 7

Inheritance and Polymorphism

Welcome to the Session, **Inheritance and Polymorphism**.

This session explains concept of inheritance and types of inheritance in Java, creation of super class, and subclass as well as the use of super keyword. Further, the session describes the concepts of method overriding and polymorphism, static and dynamic binding, and virtual method invocation. Lastly, the session explains the use of abstract keyword.

In this Session, you will learn to:

- Describe inheritance
- Explain the types of inheritance
- Explain super class and subclass
- Explain the use of super keyword
- Explain method overriding
- Describe Polymorphism
- Distinguish type of reference and type of objects
- Explain static and dynamic binding
- Explain virtual method invocation
- Explain the use of abstract keyword



7.1 Introduction

Consider a situation where a student is developing an online Website showing food habits of different animals on the earth. While designing and developing the online food Website, the student realizes that there are many animals and birds that eat same type of food and have similar characteristics. The child groups of all the animals that eat plants are known as herbivores, those that eat animals are known as carnivores, and those that eat both plants and animals are known as omnivores. This kind of grouping or classification of things is called subclassing and the child groups are known as subclasses. Similarly, Java provides the concept of inheritance for creating subclasses of a particular class.

Also, animals such as chameleon change their color based on the environment. Human beings also play different roles in their daily life such as father, son, husband, and so on. This means, that they behave differently in different situations. Similarly, Java provides a feature called polymorphism in which objects behave differently based on the context in which they are used.

7.2 Inheritance

In daily life, one often comes across objects that share a kind-of or is-a relationship with each other. For example, Car is-a four-wheeler, a four-wheeler is-a vehicle, and a vehicle is-a machine. Similarly, many other objects can be identified having such relationship. All such objects have properties that are common. For example, all four wheelers have wipers and a rear view mirror. All vehicles have a vehicle number, wheels, and engine irrespective of a four-wheeler or two-wheeler.

Figure 7.1 shows some examples of is-a relationship.

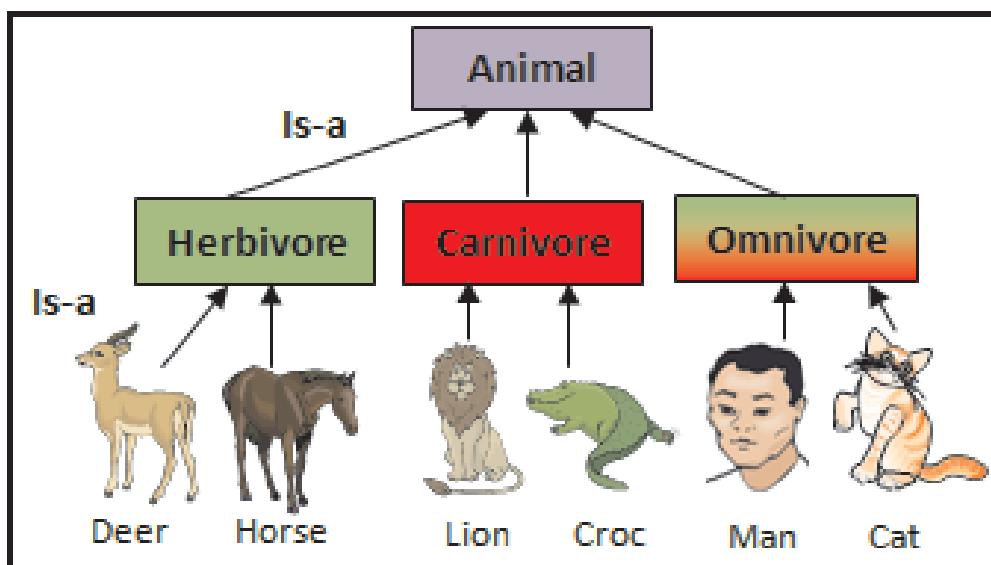


Figure 7.1: Is-a Relationship Between Real World Entities

Figure 7.1 shows is-a relationship between different objects. For example, Deer is-a herbivore and a herbivore is-a animal. Common properties of all herbivores can be stored in class herbivore. Similarly, common properties of all types of animals such as herbivore, carnivore, and omnivore can be stored in the Animal class.

Thus, the class Animal becomes the top-level class from which the other classes such as Herbivore, Carnivore, and Omnivore inherit properties with behavior. The classes Deer, Horse, Lion, and so on inherit properties from the classes Herbivore, Carnivore, and so forth.

This is called inheritance. Thus, inheritance in Java is a feature through which classes can be derived from other classes and inherit fields and methods from those classes.

7.2.1 Features and Terminologies

In Java, while implementing inheritance, the class that is derived from another class is called a subclass, derived class, child class, or extended class. The class from which the subclass is derived is called a super class, base class, or parent class.

The concept of inheritance is simple but powerful in that, it allows creating a new class from an existing class that already has some of the code required. The new class derived from the existing class can reuse the fields and methods of the existing class without having to re-write or debug the code again.

A subclass inherits all the members such as fields, nested classes, and methods from its super class except those with private access specifier. However, constructors of a class are not considered as members of a class and are not inherited by subclasses. The child class can however invoke the constructor of the super class from its own constructor.

Members having default accessibility in the super class are not inherited by subclasses of other packages. These members can only be accessed by subclasses within the same package as the super class. The subclass will have its own specific characteristics along with those inherited from the super class.

There are several types of inheritance as shown in Figure 7.2.

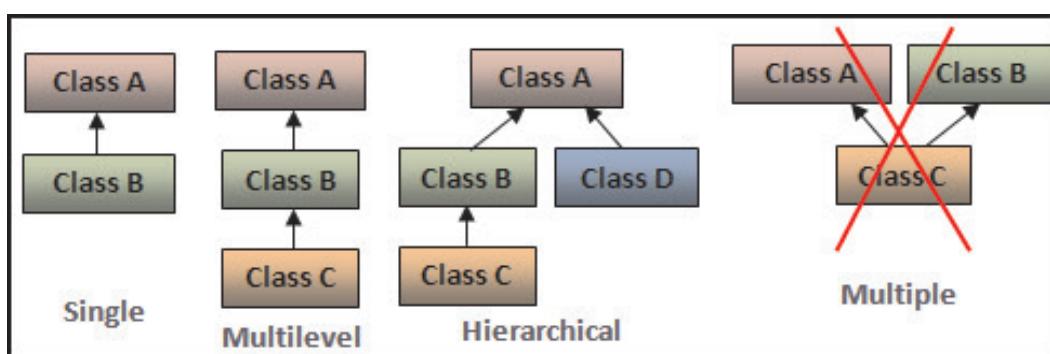


Figure 7.2: Types of Inheritance in Java

Different types of inheritance in Java are as follows:

- **Single Inheritance:** When a child class inherits from one and only one parent class, it is called single inheritance. In Figure 7.2, class B inherits from a single class A.

- **Multilevel Inheritance:** When a child class derives from a parent that itself is a child of another class, it is called multilevel inheritance. In Figure 7.2, class C derives from class B which is derived from class A.
- **Hierarchical Inheritance:** When a parent class has more than one child classes at different levels, it is called hierarchical inheritance. In Figure 7.2, class C is derived from class B and classes B and D is derived from class A.
- **Multiple Inheritance:** When a child class derives from more than one parent class, it is called multiple inheritance. Java does not support multiple inheritance. This means, a class in Java cannot inherit from more than one parent class. However, Java provides a workaround to this feature in the form of interfaces. Using interfaces, one can simulate inheritance by implementing more than one interface in a class.

7.2.2 Working with Super Class and Subclass

Within a subclass, one can use the inherited members as is, hide them, replace them, or enhance them with new members as follows:

- The inherited members, including fields and methods, can be directly used similar to any other fields.
- One can declare a field with the same name in the subclass as the one in the super class. This will lead to hiding of super class field which is not advisable.
- One can declare new fields in the subclass that are not present in the super class. These members will be specific to the subclass.
- One can write a new instance method with the same signature in the subclass as the one in the super class. This is called method overriding.
- A new static method can be created in the subclass with the same signature as the one in the super class. This will lead to hiding of the super class method.
- One can declare new methods in the subclass that are not present in the super class.
- A subclass constructor can be used to invoke the constructor of the super class, either implicitly or by using the keyword `super`.

The `extends` keyword is used to create a subclass. A class can be directly derived from only one class. If a class does not have any super class, it is implicitly derived from `Object` class.

The syntax for creating a subclass is as follows:

Syntax:

```
public class <class1-name> extends <class2-name>
{
    ...
    ...
}
```

where,

class1-name: Specifies the name of the child class.

class2-name: Specifies the name of the parent class.

Code Snippet 1 demonstrates the creation of super class **Vehicle**.

Code Snippet 1:

```
package session7;

public class Vehicle {
    // Declare common attributes of a vehicle
    protected String vehicleNo; // Variable to store vehicle number
    protected String vehicleName; // Variable to store vehicle name
    protected int wheels; // Variable to store number of wheels

    /**
     * Accelerates the vehicle
     *
     * @return void
     */
    public void accelerate(int speed) {
        System.out.println("Accelerating at:" + speed + " kmph");
    }
}
```

The parent class **Vehicle** consists of common attributes of a vehicle such as **vehicleNo**, **vehicleName**, and **wheels**. Also, it consists of a common behavior of a vehicle, that is, **accelerate()** that prints the speed at which the vehicle is accelerating.

Code Snippet 2 demonstrates the creation of subclass **FourWheeler**.

Code Snippet 2:

```
package session7;

class FourWheeler extends Vehicle{

    // Declare a field specific to child class
    private boolean powerSteer; // Variable to store steering information
    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param vID a String variable storing vehicle ID
     * @param vName a String variable storing vehicle name
     * @param numWheels an integer variable storing number of wheels
     * @param pSteer a String variable storing steering information
     */
    public FourWheeler(String vId, String vName, int numWheels, boolean pSteer) {
        // Attributes inherited from parent class
        vehicleNo = vId;
        vehicleName = vName;
        wheels = numWheels;
        // Child class' own attribute
        powerSteer = pSteer;
    }
    /**
     * Displays vehicle details
     *
     * @return void
     */
    public void showDetails() {
        System.out.println("Vehicle no:"+ vehicleNo);
        System.out.println("Vehicle Name:"+ vehicleName);
        System.out.println("Number of Wheels:"+ wheels);
        if(powerSteer == true)
            System.out.println("Power Steering:Yes");
        else
    }
}
```

```

        System.out.println("Power Steering:No");
    }

}

/***
 * Define TestVehicle class
 */

public class TestVehicle {

    /**
     * @param args the command line arguments
     */

    public static void main(String[] args) {
        // Create an object of child class and specify the values
        FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen", 4,
                true);

        objFour.showDetails(); // Invoke child class method
        objFour.accelerate(200); // Invoke inherited method
    }
}

```

Save the code in Code Snippet 2 as **TestVehicle.java**. Code Snippet 2 depicts the child class **FourWheeler** with its own attribute **powerSteer**. Values for the inherited attributes and its own attribute are specified in the constructor. The **showDetails()** method is used to display all the details.

The class **TestVehicle** consists of the **main()** method. In the **main()** method, the object **objFour** of child class is created and the parameterized constructor is invoked by passing appropriate arguments. Next, the **showDetails()** method is invoked to print the details. Also, the child class object is used to invoke the **accelerate()** method inherited from parent class and the value of speed is passed as argument.

Figure 7.3 shows the output of the program.

```

Output - session7 (run) x
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Accelerating at:200 kmph
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 7.3: Output of **TestVehicle**

From Code Snippet 2, it is clear that one can access the protected and public members of a parent class directly within the child class due to inheritance.

7.2.3 Overriding Methods

Java allows creation of an instance method in a subclass having the same signature and return type as an instance method of the super class. This is called method overriding. Method overriding allows a class to inherit behavior from a super class and then, to modify the behavior as required.

Rules to remember when overriding:

- The overriding method must have the same name, type, and number of arguments as well as return type as the super class method.
- An overriding method cannot have a weaker access specifier than the access specifier of the super class method.

The `accelerate()` method in Code Snippet 1 can be overridden in the subclass as shown in Code Snippet 3. The modified code is demonstrated in Code Snippet 3.

Code Snippet 3:

```
package session7;
class FourWheeler extends Vehicle{
    // Declare a field specific to child class
    private boolean powerSteer; // Variable to store steering information
    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param vID a String variable storing vehicle ID
     * @param vName a String variable storing vehicle name
     * @param numWheels an integer variable storing number of wheels
     * @param pSteer a String variable storing steering information
     */
    public FourWheeler(String vId, String vName, int numWheels, boolean pSteer) {
        // attributes inherited from parent class
        vehicleNo=vId;
        vehicleName=vName;
        wheels=numWheels;
        // child class' own attribute
    }
}
```

```
powerSteer=pSteer;  
}  
/**  
 * Displays vehicle details  
 *  
 * @return void  
 */  
public void showDetails() {  
    System.out.println("Vehicle no:"+ vehicleNo);  
    System.out.println("Vehicle Name:"+ vehicleName);  
    System.out.println("Number of Wheels:"+wheels);  
    if(powerSteer==true)  
        System.out.println("Power Steering:Yes");  
    else  
        System.out.println("Power Steering:No");  
}  
/**  
 * Overridden method  
 * Accelerates the vehicle  
 *  
 * @return void  
 */  
@Override  
public void accelerate(int speed) {  
    System.out.println("Maximum acceleration:"+ speed + " kmph");  
}  
}  
/**  
 * Define the TestVehicle class  
 */  
public class TestVehicle {
```

```

    /**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // Create an object of child class and specify the values
    FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen", 4,
    true);
    objFour.showDetails(); // Invoke child class method
    objFour.accelerate(200); // Invoke inherited method
}
}

```

The **accelerate()** method is overridden in the child class with the same signature and return type, but with a modified message. Notice the use of `@Override` on top of the method. This is an annotation that instructs the compiler that the method that follows is overridden from the parent class. If the compiler detects that such a method does not exist in the super class, it will generate compilation error.

Note - Annotations provide additional information about a program. Annotations have no direct effect on the functioning of the code they annotate.

Figure 7.4 shows the output of Code Snippet 3.

```

Output - session7 (run) ×
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Maximum acceleration:200 kmph
BUILD SUCCESSFUL (total time: 3 seconds)

```

Figure 7.4: Output of `TestVehicle` Class after Overriding

Notice that the **accelerate()** method now prints the message specified in the subclass. This means that a call to the **accelerate()** method using the subclass object `objFour.accelerate()` first searches for the method in the same class. Since the **accelerate()** method is overridden in the subclass, it invokes the subclass version of the **accelerate()** method and not the super class **accelerate()** method.

7.2.4 Accessing Super Class Constructor and Methods

In Code Snippet 3, the subclass constructor is used to initialize the values of the common attributes inherited from the super class `Vehicle`. However, this is not the correct approach because all the subclasses of the `Vehicle` class will have to initialize the values of common attributes every time in their constructors.

This duplication of code is not only inefficient but also implies that to use these members, a subclass is granted direct access to them. However, in certain cases, the user might want to keep the data members present in a super class private. In such a case, the subclass would not be able to access the members directly nor initialize these variables on its own. Java provides a solution by allowing the subclass to invoke the super class constructor and methods using the keyword `super`. For example, '`super.member-name`' where member may be a method or an instance variable. The use of `super` keyword is extremely helpful when member names of subclass hide the members existing by the same name in the super class.

Also, notice that when the `accelerate()` method is overridden in the subclass, the statement(s) written in the `accelerate()` method of the super class, `Vehicle`, are not printed.

To address these issues, one can use the `super` keyword to invoke the super class method using `super.method-name()` and the `super()` method to invoke the super class constructors from the subclass constructors.

Code Snippet 4 demonstrates the modified super class `Vehicle.java` using a parameterized constructor.

Code Snippet 4:

```
package session7;
class Vehicle {
    protected String vehicleNo; // Variable to store vehicle number
    protected String vehicleName; // Variable to store vehicle name
    protected int wheels; // Variable to store number of wheels
    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param vId a String variable storing vehicle ID
     * @param vName a String variable storing vehicle name
     * @param numWheels an integer variable storing number of wheels
     */
    public Vehicle(String vId, String vName, int numWheels) {
        vehicleNo=vId;
        vehicleName=vName;
        wheels=numWheels;
    }
    /**
     * Accelerates the vehicle
     * @return void
     */
    public void accelerate(int speed) {
```

```

        System.out.println("Accelerating at:"+ speed + " kmph");
    }
}

```

Code Snippet 4 demonstrates the use of parameterized constructor in **Vehicle** class for initializing the common attributes of a vehicle.

Code Snippet 5 depicts the modified subclass **FourWheeler** using the **super** keyword to invoke super class constructor and methods.

Code Snippet 5:

```

package session7;
class FourWheeler extends Vehicle{
    private boolean powerSteer; // Variable to store steering information
    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param vID a String variable storing vehicle ID
     * @param vName a String variable storing vehicle name
     * @param numWheels an integer variable storing number of wheels
     * @param pSteer a String variable storing steering information
     */
    public FourWheeler(String vId, String vName, int numWheels, boolean pSteer) {
        // Invoke the super class constructor
        super(vId, vName, numWheels);
        powerSteer=pSteer;
    }
    /**
     * Displays vehicle details
     *
     * @return void
     */
    public void showDetails() {
        System.out.println("Vehicle no:"+ vehicleNo);
        System.out.println("Vehicle Name:"+ vehicleName);
        System.out.println("Number of Wheels:"+ wheels);
        if(powerSteer==true)
    }
}

```

```

        System.out.println("Power Steering:Yes");
    else
        System.out.println("Power Steering:No");
    }
}

/**
 * Overridden method
 * Displays the acceleration details of the vehicle
 *
 * @return void
 */
@Override
public void accelerate(int speed) {
    // Invoke the super class accelerate() method
    super.accelerate(speed);
    System.out.println("Maximum acceleration:"+ speed + " kmph");
}

}

/**
 * Define the TestVehicle class
*/
public class TestVehicle {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        FourWheeler objFour = new FourWheeler("LA-09 CS-1406", "Volkswagen", 4,
        true);
        objFour.showDetails();
        objFour.accelerate(200);
    }
}

```

Code Snippet 5 depicts the use of `super()` to call the super class constructor from the child class constructor. Similarly, the `super.accelerate()` statement is used to invoke the super class `accelerate()` method from the child class.

Figure 7.5 shows the output of Code Snippet 5.

```
Output - session7 (run) ×
run:
Vehicle no:LA-09 CS-1406
Vehicle Name:Volkswagen
Number of Wheels:4
Power Steering:Yes
Accelerating at:200 kmph
Maximum acceleration:200 kmph
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 7.5: Output of TestVehicle Class After Using `super` Keyword

Notice that the output displays statements of both `accelerate()` methods, that is, of super class as well as subclass. Also, the arguments for the members inherited from super class were passed to the subclass constructor. However, the values were passed to the super class constructor using `super()` and therefore, super class members were initialized in the super class constructor. Now, every subclass of `Vehicle` class is not required to write code for initializing the common attributes of a vehicle. Instead, it can pass the values to the super class constructor by using `super()`.

7.3 Polymorphism

The word polymorph is a combination of two words namely, 'poly' which means 'many' and 'morph' which means 'forms'. Thus, polymorph refers to an object that can have many different forms. This principle can also be applied to subclasses of a class that can define their own specific behaviors as well as derive some of the similar functionality of the super class. The concept of method overriding is an example of polymorphism in object-oriented programming in which the same method behaves in a different manner in super class and in subclass.

7.3.1 Understanding Static and Dynamic Binding

When the compiler resolves the binding of methods and method calls at compile time, it is called static binding or early binding. If the compiler resolves the method calls and the binding at runtime, it is called dynamic binding or late binding. All static method calls are resolved at compile time and therefore, static binding is done for all static method calls. The instance method calls are always resolved at runtime.

Static methods are class methods and are accessed using the class name itself. The use of static methods is encouraged because object references are not required to access them and therefore, static method calls are resolved during compile time itself. This is also the reason why static methods are not overridden.

Similarly, Java does not allow polymorphic behavior of variables of a class. Therefore, access to all the variables also follows static binding.

Some important differences between static and dynamic binding are listed in Table 7.1.

Static Binding	Dynamic Binding
Static binding occurs at compile time.	Dynamic binding occurs at runtime.
Private, static, and final methods and variables use static binding and are bounded by compiler.	Virtual methods are bounded at runtime based upon the runtime object.
Static binding uses object type information for binding. That is, the type of class.	Dynamic binding uses reference type to resolve binding.
Overloaded methods are bounded using static binding.	Overridden methods are bounded using dynamic binding.

Table 7.1: Static v/s Dynamic Binding

Code Snippet 6 demonstrates an example of static binding.

Code Snippet 6:

```
class Employee {
    String empId; // Variable to store employee ID
    String empName; // Variable to store employee name
    int salary; // Variable to store salary
    float commission; // Variable to store commission
    /**
     * Parameterized constructor to initialize the variables
     *
     * @param id a String variable storing employee id
     * @param name a String variable storing employee name
     * @param sal an integer variable storing salary
     *
     */
    public Employee(String id, String name, int sal) {
        empId=id;
        empName=name;
        salary=sal;
    }
    /**

```

```
* Calculates commission based on sales value
* @param sales a float variable storing sales value
*
* @return void
*/
public void calcCommission(float sales) {
    if(sales > 10000)
        commission = salary * 20 / 100;
    else
        commission=0;
}
/***
* Overloaded method. Calculates commission based on overtime
* @param overtime an integer variable storing overtime hours
*
* @return void
*/
public void calcCommission(int overtime) {
    if(overtime > 8)
        commission = salary/30;
    else
        commission = 0;
}
/***
* Displays employee details
*
* @return void
*/
public void displayDetails() {
    System.out.println("Employee ID:"+empId);
    System.out.println("Employee Name:"+empName);
```

```

        System.out.println("Salary:"+salary);
        System.out.println("Commission:"+commission);
    }
}

/**
 * Define the EmployeeDetails class
 */
public class EmployeeDetails {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        // Instantiate the Employee class object
        Employee objEmp = new Employee ("E001", "Maria Nemeth", 40000);
        // Invoke the calcCommission() with float argument
        objEmp.calcCommission(20000F);
        objEmp.displayDetails(); // Print the employee details
    }
}

```

Code Snippet 6 depicts a class **Employee** with four member variables. The constructor is used to initialize the member variables with the received values. The class consists of two **calcCommission()** methods. The first method calculates commission based on the sales done by the employee and the other based on the number of hours the employee worked overtime. The **displayDetails()** method is used to print the details of the employee.

Another class **EmployeeDetails** is created with the **main()** method. Inside the **main()** method, object **objEmp** of class **Employee** is created and the parameterized constructor is invoked with appropriate arguments. Next, the **calcCommission()** method is invoked with a **float** argument **20000F**.

In the example, when the **calcCommission()** method is executed, the method with **float** argument gets invoked because it was bounded during compile time based on the type of variable, that is, **float**. Lastly, **displayDetails()** method is invoked to print the details of the employee.

Figure 7.6 shows the output of Code Snippet 6.

```
run:  
Employee ID:E001  
Employee Name:Maria Nemeth  
Salary:40000  
Commission:8000.0
```

Figure 7.6: Using Static Binding

Now, consider the class hierarchy shown in Figure 7.7.

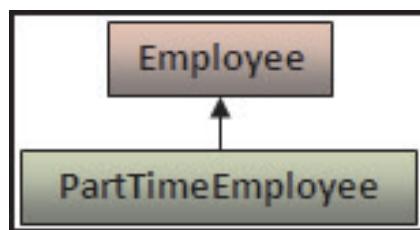


Figure 7.7: Single Inheritance Between Employee and PartTimeEmployee

Code Snippet 7 demonstrates an example of dynamic binding.

Code Snippet 7:

```
class PartTimeEmployee extends Employee{
    // Subclass specific variable
    String shift; // Variable to store shift information
    /**
     * Parameterized constructor to initialize values based on user input
     *
     * @param id a String variable storing employee ID
     * @param name a String variable storing employee name
     * @param sal an integer variable storing salary
     * @param shift a String variable storing shift information
     */
    public PartTimeEmployee(String id, String name, int sal, String shift)
    {
        // Invoke the super class constructor
        super(id, name, sal);
        this.shift=shift;
    }
    /**

```

```

* Overridden method to display employee details
*
* @return void
*/
@Override
public void displayDetails() {
    calcCommission(12); // Invoke the inherited method
    super.displayDetails(); // Invoke the super class display method
    System.out.println("Working shift:"+shift);
}
}

/**
* Modified EmployeeDetails.java
*/
public class EmployeeDetails {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the Employee class object
        Employee objEmp = new Employee("E001", "Maria Nemeth", 40000);
        objEmp.calcCommission(20000F); // Calculate commission
        objEmp.displayDetails(); // Print the details
        System.out.println("-----");
        // Instantiate the Employee object as PartTimeEmployee
        Employee objEmp1 = new PartTimeEmployee("E002", "Rob Smith", 30000,
        "Day");
        objEmp1.displayDetails(); // Print the details
    }
}

```

Code Snippet 7 displays the **PartTimeEmployee** class that is inherited from the **Employee** class created earlier. The class has its own variable called **shift** which indicates the day or night shift for an employee. The constructor of **PartTimeEmployee** class calls the super class constructor using the **super** keyword to initialize the common attributes of an employee. Also, it initializes the **shift** variable.

The subclass overrides the `displayDetails()` method. Within the overridden method, the `calcCommission()` method is invoked with an integer argument. This will calculate commission based on overtime. Next, the super class `displayDetails()` method is invoked to display basic details of the employee as well as the shift details.

The `EmployeeDetails` class is modified to create another object `objEmp1` of class `Employee`. However, the object is assigned the reference of class `PartTimeEmployee` and the constructor is invoked with four arguments. Next, the `displayDetails()` method is invoked to print employee details.

Figure 7.8 shows the output of Code Snippet 7.

```

Employee ID:E001
Employee Name:Maria Nemeth
Salary:40000
Commission:8000.0
-----
Employee ID:E002
Employee Name:Rob Smith
Salary:30000
Commission:1000.0
Working shift:Day

```

Figure 7.8: Using Dynamic Binding

Notice that the output of employee `E002` shows the details of shift variable as well. This indicates that the `displayDetails()` method of the subclass `PartTimeEmployee` was invoked even though the type of object for `objEmp1` was `Employee`. This is because, during creation it stored the reference of `PartTimeEmployee` class.

This is dynamic binding, that is, the method call is bound to the object at runtime based on the reference assigned to the object.

7.3.2 Differentiate Between Type of Reference and Type of Object

In Code Snippet 7, type of object of `objEmp1` is `Employee`. This means that the object will have all characteristics of an `Employee`. However, the reference assigned to the object was of `PartTimeEmployee`. This means that the object will bind with the members of `PartTimeEmployee` class during runtime. That is, object type is `Employee` and reference type is `PartTimeEmployee`. This is possible only when the classes are related with a parent-child relationship.

Java allows casting an instance of a subclass to its parent class. This is known as upcasting.

For example,

```

PartTimeEmployee objPT = new PartTimeEmployee();
Employee objEmp = objPT; // upcasting

```

While upcasting a child object, the child object `objPT` is directly assigned to the parent class object `objEmp`. However, the parent object cannot access members that are specific to the child class and not available in the parent class.

Java also allows casting the parent reference back to the child type. This is because parent references an object of type child. Casting a parent object to child type is called downcasting because an object is being casted to a class lower down in the inheritance hierarchy. However, downcasting requires explicit type casting by specifying the child class name in brackets.

For example,

```
PartTimeEmployee objPT1 = (PartTimeEmployee) objEmp; // downcasting
```

7.3.3 Invocation of Virtual Method

In Code Snippet 7, during execution of the statement `Employee objEmp1 = new PartTimeEmployee(...)`, the runtime type of the `Employee` object is determined. The compiler does not generate error because the `Employee` class has a `displayDetails()` method. At runtime, the method executed is referenced from the `PartTimeEmployee` object. This aspect of polymorphism is called virtual method invocation.

The difference here is between the compiler and the runtime behavior. The compiler checks the accessibility of each method and field based on the class definition whereas the behavior associated with an object is determined at runtime.

This is an important aspect of polymorphism wherein the behavior of the object is determined at runtime based on the reference passed to it.

Since the object created was of `PartTimeEmployee`, the `displayDetails()` method of `PartTimeEmployee` is invoked even though the object type is `Employee`. This is referred to as virtual method invocation and the method is referred to as virtual method.

In Java, all methods behave in this manner, whereby a method overridden in the child class is invoked at runtime irrespective of the type of reference used in the source code at compile time. In other languages such as C++, the same can be achieved by using the keyword `virtual`.

7.4 Using the abstract Keyword

While implementing inheritance, it can be seen that one can create object of parent class as well as child class. However, what if the user wants to restrict direct use of the parent class? That is, in some cases, one might want to define a super class that declares the structure of a given entity without giving a complete implementation of every method. Such a super class serves as a generalized form that will be inherited by all of its subclasses. The methods of the super class serve as a contract or a standard that the subclass can implement in its own way. Java provides the `abstract` keyword to accomplish this task.

Thus, an `abstract` method is one that is declared with the `abstract` keyword and is without an implementation, that is, without any body.

The **abstract** method does not contain any '{}' brackets and ends with a semicolon. The syntax for declaring an **abstract** method is as follows:

Syntax:

```
abstract <return-type> <method-name> (<parameter-list>);
```

where,

abstract: Indicates that the method is an **abstract** method.

For example,

```
public abstract void calculate();
```

An **abstract class** is one that consists of **abstract** methods. **abstract class** serves as a framework that provides certain behavior for other classes. The subclass provides the requirement-specific behavior of the existing framework. **abstract classes** cannot be instantiated and they must be subclassed to use the class members. The subclass provides implementations for the **abstract** methods in its parent class. The syntax for declaring an **abstract class** is as follows:

Syntax:

```
abstract class <class-name>
{
    // declare fields
    // define concrete methods
    [abstract <return-type> <method-name> (<parameter-list>);]
}
```

where,

abstract: Indicates that the class and method are **abstract**.

For example,

```
public abstract Calculator
{
    public float getPI() { // Define a concrete method
        return 3.14F;
    }
    abstract void calculate(); // Declare an abstract method
}
```

Consider the class hierarchy as shown in Figure 7.9.

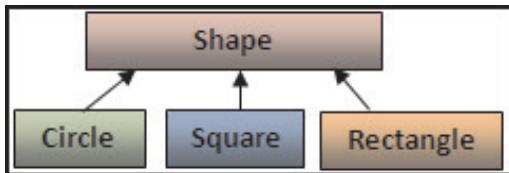


Figure 7.9: Using Abstract Class

Code Snippet 8 demonstrates creation of abstract class and abstract method.

Code Snippet 8:

```

package abstractdemo;

abstract class Shape {

    private final float PI = 3.14F; // Variable to store value of PI

    /**
     * Returns the value of PI
     *
     * @return float
     */

    public float getPI() {
        return PI;
    }

    /**
     * Abstract method
     * @param val a float variable storing the value specified by user
     *
     * @return float
     */

    abstract void calculate(float val);
}
  
```

The class **Shape** is an abstract class with one concrete method **getPI()** and one abstract method **calculate()**.

To use the abstract class, one must create subclasses. Code Snippet 9 demonstrates two subclasses **Circle** and **Rectangle** inheriting the **Shape** class.

Code Snippet 9:

```
package abstractdemo;

/**
 * Define the child class Circle
 */
class Circle extends Shape{
    float area; // Variable to store area of a circle
    /**
     * Implement the abstract method to calculate area of circle
     *
     * @param rad a float variable storing value of radius
     * @return void
     */
    @Override
    void calculate(float rad) {
        area = getPI() * rad * rad;
        System.out.println("Area of circle is:"+ area);
    }
}

/**
 * Define the child class Rectangle
 */
class Rectangle extends Shape{
    float perimeter; // Variable to store perimeter value
    float length=10; // Variable to store length
    /**
     * Implement the abstract method to calculate the perimeter
     *
     * @param width a float variable storing width
     * @return void
     */
    @Override
    void calculate(float width) {
        perimeter = 2 * (length+width);
    }
}
```

```

        System.out.println("Perimeter of the Rectangle is:"+perimeter);

    }

}

```

The class **Circle** implements the abstract method **calculate()** to calculate the area of a circle. Similarly, the class **Rectangle** implements the abstract method **calculate()** to calculate the perimeter of a rectangle.

Code Snippet 10 depicts the code for **Calculator** class that uses the subclasses based on user input.

Code Snippet 10:

```

package abstractdemo;

public class Calculator {

    /**
     * @param args the command line arguments
     */

    public static void main(String[] args)
    {
        Shape objShape; // Declare the Shape object
        String shape; // Variable to store the type of shape
        if(args.length==2) { // Check the number of command line arguments
            //Retrieve the value of shape from args[0]
            shape = args[0].toLowerCase(); // converting to lower case
            switch(shape) {
                // Assign reference to Shape object as per user input
                case "circle": objShape = new Circle();
                objShape.calculate(Float.parseFloat(args[1]));
                break;
                case "rectangle": objShape = new Rectangle();
                objShape.calculate(Float.parseFloat(args[1]));
                break;
            }
        }
        else{
            // Error message to be displayed when arguments are not supplied
        }
    }
}

```

```

System.out.println("Usage: java Calculator <shape-name><value>");

}
}

}

```

The class **Calculator** takes two arguments from the user at command line namely, the shape and the value for the shape. Notice the **Shape** class object **objShape**. It was mentioned earlier that an abstract class cannot be instantiated. That is, one cannot write **Shape objShape = new Shape()**. However, an abstract class can be assigned a reference of its subclasses.

The statement **args.length** checks the number of arguments supplied by the user. If the length is 2, the value for shape is extracted from the first argument **args[0]** and stored in the **shape** variable. Next, the **switch** statement evaluates the value of **shape** variable and accordingly assigns the reference of the appropriate shape to the **objShape** object. For example, if shape is circle, it assigns **new Circle()** as the reference. Then, using the object **objShape**, the **calculate()** method is invoked for the referenced subclass.

To execute the example at command line, write following command:

```
java Calculator Rectangle 12
```

Note that the word **Circle** can be in any case. Within the code, it will be converted to lowercase.

To execute the example in NetBeans IDE, type the arguments in the **Arguments** box of **Run** property as shown in Figure 7.10.

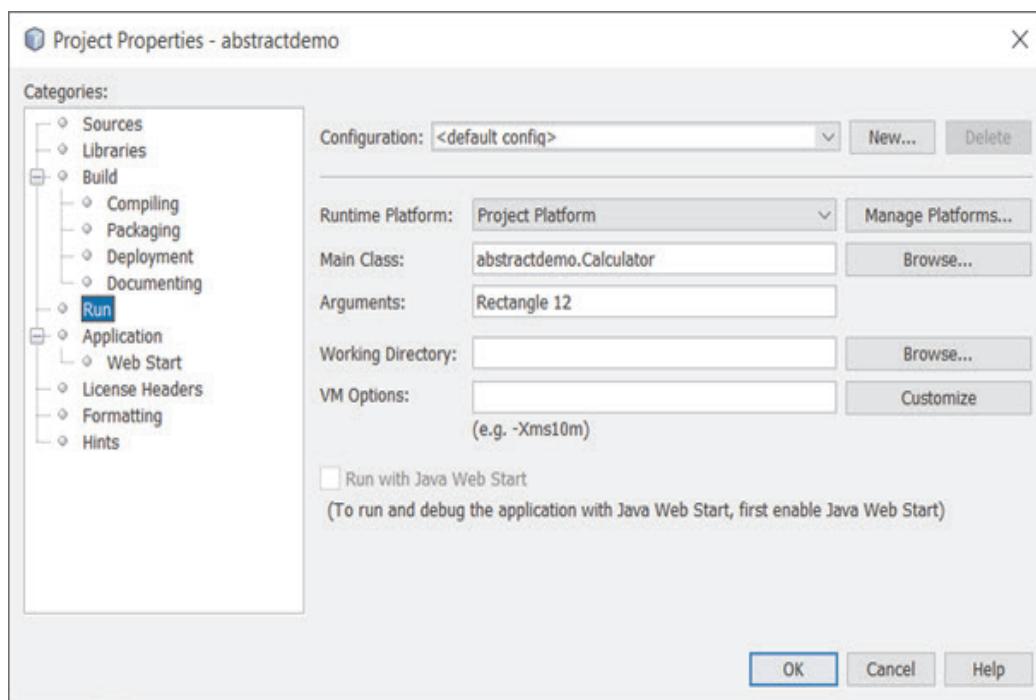
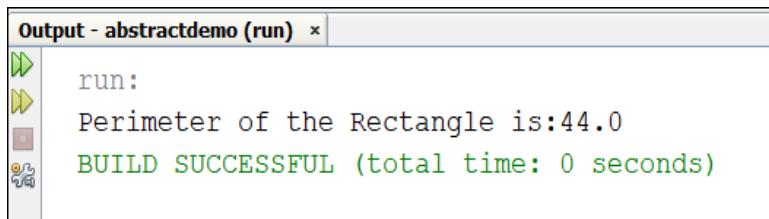


Figure 7.10: Setting Command Line Arguments

Figure 7.11 shows the output of the code after execution.



```
Output - abstractdemo (run) ×
run:
Perimeter of the Rectangle is:44.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 7.11: Output of Calculator Class

Notice that the output shows the perimeter of a rectangle. This is because the first command line argument was `Rectangle`. Therefore, within the `main()` method, the switch case for rectangle got executed.

Similarly, one can inherit several other classes from the abstract class `Shape` and implement the `calculate()` method as required.

7.5 Check Your Progress

1. _____ keyword is used to access parent class constructor and methods from child class.

(A) virtual	(C) super
(B) extends	(D) base

2. Which of the following statements about the static and dynamic binding are true?

a.	Overridden methods are bounded using static binding	c.	Private, static, and final methods and variables use static binding and are bounded by compiler
b.	Static binding occurs at compile time and dynamic binding occurs at runtime	d.	Virtual methods are bounded at compile time based upon the runtime object

(A)	a, c	(C)	b, d
(B)	b, c	(D)	a, d

3. Match the following inheritance type with the corresponding description.

	Inheritance Type		Description
a.	Hierarchical	1.	A child class derives from more than one parent class
b.	Single	2.	A parent class has more than one child classes at different levels
c.	Multiple	3.	A child class derives from a parent that itself is a child of another class
d.	Multilevel	4.	A child class inherits from one and only one parent class

(A)	a-4, b-3, c-2, d-1	(C)	a-2, b-4, c-1, d-3
(B)	a-2, b-3, c-4, d-1	(D)	a-4, b-1, c-3, d-2

4. Consider following code:

```

class Furniture {
    String ID;
    float price;
    public Furniture(String ID, float price) {
        this.ID = ID;
        this.price = price;
    }
    public void displayDetails() {
        System.out.println("ID: " + ID);
        System.out.println("Price: " + price);
    }
}
public class Table extends Furniture
{
    String type;
    public Table(String ID, float price, String type) {
        super(ID, price);
        this.type = type;
    }
    public void displayDetails() {
        super.displayDetails();
        System.out.println("Type: " + type);
    }
    public static void main(String[] args) {
        Furniture obj = new Table("F001", 2000F, "Wooden");
        obj.displayDetails();
    }
}

```

What will be the output of the code?

(A)	ID: F001 Price: 2000.0	(C)	ID: F001 Price: 2000.0 Type: Wooden
(B)	Compilation Error	(D)	Runtime Error

5. Consider following code: (Assumption: Super class – Vehicle and Subclass – Car)

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

Which concept of object-oriented programming is used in the code?

(A)	Downcasting	(C)	Abstraction
(B)	Overriding	(D)	Upcasting

6. Consider following code:

```
class Animal
{
    ...
}

class Herbivore extends Animal
{
    ...
}

class Deer extends Herbivore
{
    ...
}
```

Which type of inheritance is used in the code?

(A)	Single	(C)	Multiple
(B)	Multilevel	(D)	Hierarchical

7.5.1 Answers

1.	C
2.	B
3.	C
4.	C
5.	D
6.	B

Summary

- Inheritance is a feature in Java through which classes can be derived from other classes and inherit fields and methods from classes it is inheriting.
- The class that is derived from another class is called a subclass, derived class, child class, or extended class. The class from which the subclass is derived is called a super class.
- Creation of an instance method in a subclass having the same signature and return type as an instance method of the super class is called method overriding.
- Polymorphism refers to an object that can have many different forms.
- When the compiler resolves the binding of methods and method calls at compile time, it is called static binding or early binding. If the compiler resolves the method calls and the binding at runtime, it is called dynamic binding or late binding.
- An abstract method is one that is declared with the abstract keyword without an implementation, that is, without any body.
- An abstract class is one that consists of abstract methods and serves as a framework that provides certain pre-defined behavior for other classes that can be modified later as per the requirement of the inheriting class.

Try It Yourself

1. **ITQuest.com** is a well-known online IT training company based in **Los Angeles, USA**. The company organizes online code competitions every year. Thousands of IT whiz kids participate in the competition to test their IT skills. The winner(s) is awarded with a one year scholarship and free training in the subject of his choice in the IT institutes affiliated with the company. This year, the competition is on Java and the following code has been posted online for the participants.

```
package session7;

public class TwoWheeler {

    String vehicleId;
    String type;
    int wheels;
    float price;

    public TwoWheeler(String vId, String vType, int tyres, float rate) {
        vehicleId=vId;
        type=vType;
        wheels=tyres;
        price=rate;
    }

    public void printDetails() {
        System.out.println("Bicycle Id: "+vehicleId);
        System.out.println("Bicycle Type: "+type);
        System.out.println("Wheels: "+wheels);
        System.out.println("Price: $" +price);
    }
}
```

Try It Yourself

```

public class Bicycle{
    boolean gear;
    public Bicycle(String vId, String vName, int tyres, float price, boolean gear) {
        super(vId, vName, tyres, price);
        gear=gear;
    }
    @Override
    public void printDetails() {
        if(gear==true)
            System.out.println("Geared: Yes");
        else
            System.out.println("Geared: No");
    }
    public static void main(String[] args) {
        TwoWheeler obj = new Bicycle(args[0], args[1],
            Integer.parseInt(args[2]), args[3],args[4]);
        obj.printDetails();
    }
}

```

The code is not functioning properly. The user passes following arguments at command line:

B001 Mountain-Bicycle 2 200 true

Fix the code to get following output:

Bicycle Id: B001

Bicycle Type: Mountain-Bicycle

Wheels: 2

Price: \$200.0

Geared: Yes

(Hint: Use concepts of inheritance, super, wrapper class, and virtual method invocation)

Session - 8

Interfaces and Nested Classes

Welcome to the Session, **Interfaces and Nested Classes**.

This session explains the concept of interfaces and the purpose of using interfaces as well as implementation of multiple interfaces. Further, the session describes the concept of abstraction, nested class, member class, and local class. Lastly, the session explains the use of anonymous class and static nested class.

In this Session, you will learn to:

- Describe interfaces
- Illustrate the purpose of interfaces
- Explain implementation of multiple interfaces
- Describe private methods in interfaces
- Define Abstraction
- Explain Nested class
- Explain Member class
- Explain Local class
- Elaborate Anonymous class
- Outline static nested class



8.1 Introduction

Java does not support multiple inheritance. However, there are several cases when it becomes mandatory for an object to inherit properties from multiple classes to avoid redundancy and complexity in code. For this purpose, Java provides a workaround in the form of interfaces.

Also, Java provides the concept of nested classes to make certain types of programs easy to manage, more secure, and less complex.

8.2 Interfaces

An interface in Java is a contract that specifies the standards to be followed by the types that implement it. The classes that accept the contract must abide by it.

An interface and a class are similar in following ways:

- An interface can contain multiple methods.
- An interface is saved with a **.java** extension and the name of the file must match with the name of the interface just as a Java class.
- The bytecode of an interface is also saved in a **.class** file.
- Interfaces are stored in packages and the bytecode file is stored in a directory structure that matches the package name.

However, an interface and a class differ in several ways as follows:

- An interface cannot be instantiated.
- An interface cannot have constructors.
- All the methods of an interface are implicitly `abstract`.
- The fields declared in an interface must be both `static` and `final`. Interface cannot have instance fields.
- An interface is not extended but implemented by a class.
- An interface can extend multiple interfaces.

8.2.1 Purpose of Interfaces

Objects in Java interact with the outside world with the help of methods exposed by them. Thus, it can be said that, methods serve as the object's interface with the outside world.

This is similar to the buttons in front of a television set. These buttons acts as an interface between the user and the electrical circuit and wiring on the other side of the plastic casing. When the 'power' button is pressed, the television set is turned ON and OFF.

In Java, an interface is a collection of related methods without any body. These methods form the contract that the implementing class must agree with. When a class implements an interface, it becomes more formal about the behavior it promises to provide. This contract is enforced at build time by the compiler. If a class implements an interface, all the methods declared by that interface must appear in

the implementing class for the class to compile successfully.

Thus, in Java, an interface is a reference type that is similar to a class.

However, it can contain only method signatures, constants, and nested types. There is no method definition but only declaration. Also, unlike a class, interfaces cannot be instantiated and must be implemented by classes or extended by other interfaces in order to use them.

There are several situations in software engineering when it becomes necessary for different groups of developers to agree to a 'contract' that specifies how their software interacts. However, each group should have the liberty to write their code in their desired manner without having the knowledge of how other groups are writing their code. Java interfaces can be used for defining such contracts.

Interfaces do not belong to any class hierarchy, even though they work in conjunction with classes. Java does not permit multiple inheritance for which interfaces provide an alternative. In Java, a class can inherit from only one class, but it can implement multiple interfaces. Therefore, objects of a class can have multiple types such as the type of their own class as well as the types of the interfaces that the class implements. The syntax for declaring an interface is as follows:

Syntax:

```
<visibility> interface <interface-name> extends <other-interfaces, ... >
{
    // declare constants
    // declare abstract methods
}
```

where,

<visibility>: Indicates the access rights to the interface. Visibility of an interface is always public.

<interface-name>: Indicates the name of the interface.

<other-interfaces>: List of interfaces that the current interface inherits from.

For example,

```
public interface Sample extends Interface1{
    static final int someInteger;
    public void someMethod();
}
```

In Java, interface names are written in CamelCase, that is, first letter of each word is capitalized. Also, the name of the interface describes an operation that a class can perform. For example,

```
interface Enumerable
interface Comparable
```

Some programmers prefix the letter 'I' with the interface name to distinguish interfaces from classes. For example,

```
interface IComparable
interface IComparable
```

Notice that the method declaration does not have any braces and is terminated with a semicolon. Also, the body of the interface contains only abstract methods and no concrete method. However, since all methods in an interface are implicitly abstract, the `abstract` keyword is not explicitly specified with the method signature.

When a class implements an interface, it must implement all its methods. If the class does not implement all its methods, it must be marked `abstract`. Also, if the class implementing the interface is `abstract`, one of its subclasses must implement the unimplemented methods. Again, if any of the `abstract` class subclasses does not implement all the interface methods, the subclass must be marked `abstract` as well.

The data members of an interface are implicitly `static`, `final`, and `public`.

Consider the hierarchy of vehicles where `IVehicle` is the interface that declares methods which can be defined by implementing classes such as `TwoWheeler`, `FourWheeler`, and so on. To create a new interface in NetBeans IDE, right-click the package name and select **New → Java Interface** as shown in Figure 8.1.

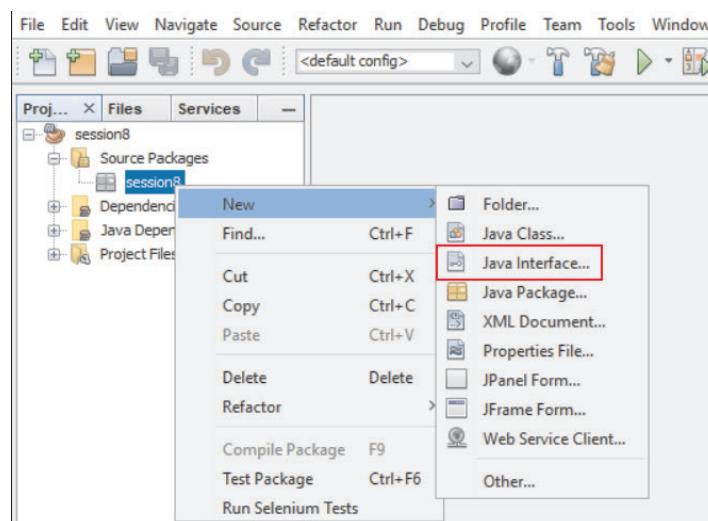


Figure 8.1: Creating a New Java Interface

A dialog box appears where the user must provide a name for the interface and then, click **OK**. This will create an interface with the specified name.

Code Snippet 1 defines the interface, `IVehicle`.

Code Snippet 1:

```
package session8;

public interface IVehicle {
    // Declare and initialize constant
    static final String STATEID="LA-09"; // variable to store state ID
    /**
     * Abstract method to start a vehicle
    */
}
```

```

* @return void
*/
public void start();
/***
 * Abstract method to accelerate a vehicle
 * @param speed an integer variable storing speed
 * @return void
*/
public void accelerate(int speed);
/***
 * Abstract method to apply a brake
 * @return void
*/
public void brake();
/***
 * Abstract method to stop a vehicle
 * @return void
*/
public void stop();
}

```

Code Snippet 1 defines an interface **IVehicle** with a static constant String variable **STATEID**. Also, it declares several abstract methods such as **start()**, **accelerate(int)**, **brake()**, and **stop()**. To use the interface, a class is required to implement the interface. The instantiating class implementing the interfaces must define all these methods.

The syntax to implement an interface is as follows:

Syntax:

```

class <class-name> implements <Interface1>, ...
{
    // class members
    // overridden abstract methods of the interface(s)
}

```

Code Snippet 2 defines the class **TwoWheeler** that implements the **IVehicle** interface.

Code Snippet 2:

```

package session8;
class TwoWheeler implements IVehicle {
    String ID; // variable to store vehicle ID
    String type; // variable to store vehicle type
    /**

```

```
* Parameterized constructor to initialize values based on user input
* @param ID a String variable storing vehicle ID
* @param type a String variable storing vehicle type
*/
public TwoWheeler(String ID, String type) {
    this.ID = ID;
    this.type = type;
}
/**
 * Overridden method, starts a vehicle
 *
 * @return void
 */
@Override
public void start() {
    System.out.println("Starting the "+ type);
}
/**
 * Overridden method, accelerates a vehicle
 * @param speed an integer storing the speed
 * @return void
 */
@Override
public void accelerate(int speed) {
    System.out.println("Accelerating at speed:"+speed+ " kmph");
}
/**
 * Overridden method, applies brake to a vehicle
 *
 * @return void
 */
@Override
public void brake() {
    System.out.println("Applying brakes");
}
/**
 * Overridden method, stops a vehicle
 *
 * @return void
 */
```

```

@Override
public void stop() {
    System.out.println("Stopping the "+ type);
}

/**
 * Displays vehicle details
 *
 * @return void
 */
public void displayDetails() {
    System.out.println("Vehicle No.: "+ STATEID+ " "+ ID);
    System.out.println("Vehicle Type.: "+ type);
}

/**
 * Define the class TestVehicle and save the entire code as TestVehicle.java
 *
 */
public class TestVehicle {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Verify the number of command line arguments
        if(args.length==3) {
            // Instantiate the TwoWheeler class
            TwoWheeler objBike = new TwoWheeler(args[0], args[1]);
            // Invoke the class methods
            objBike.displayDetails();
            objBike.start();
            objBike.accelerate(Integer.parseInt(args[2]));
            objBike.brake();
            objBike.stop();
        }
        else {
            System.out.println("Usage: java TwoWheeler <ID><Type><Speed>");
        }
    }
}

```

Code Snippet 2 defines the class **TwoWheeler** that implements the **IVehicle** interface. The class consists of some instance variables and a constructor to initialize the variables. Notice, that the class implements all the methods of the interface **IVehicle**. The **displayDetails()** method is used to display the details of the specified vehicle.

The **main()** method is defined in another class **TestVehicle**. Within the **main()** method, the number of command line arguments specified is verified and accordingly the object of class **TwoWheeler** is created. Next, the object is used to invoke various methods of the class.

Figure 8.2 shows the output of the code when the user passes **CS-2723 Bike 80** as command line arguments.

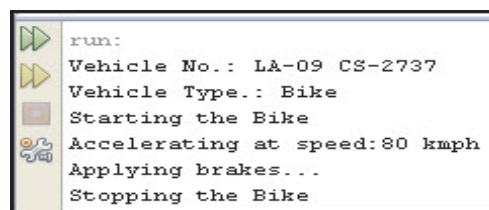


Figure 8.2: Output of **TwoWheeler.java**

8.2.2 Implementing Multiple Interfaces

Java does not support multiple inheritance of classes, but allows implementing multiple interfaces to simulate multiple inheritance. To implement multiple interfaces, write the interface names after the **implements** keyword separated by a comma. For example,

```
public class Sample implements Interface1, Interface2{  
}
```

Code Snippet 3 defines the interface **IManufacturer**.

Code Snippet 3:

```
package session8;  
  
public interface IManufacturer {  
    /**  
     * Abstract method to add contact details  
     * @param detail a String variable storing manufacturer detail  
     * @return void  
     */  
    public void addContact(String detail);  
    /**  
     * Abstract method to call the manufacturer  
     * @param phone a String variable storing phone number  
     * @return void  
     */
```

```

*/
public void callManufacturer(String phone);
/**
 * Abstract method to make payment
 * @param amount a float variable storing amount
 * @return void
 */
public void makePayment(float amount);
}

```

The interface **IManufacturer** declares three abstract methods namely, **addContact(String)**, **callManufacturer(String)**, and **makePayment(float)** that must be defined by the implementing classes.

The modified class, **TwoWheeler** implementing both the **IVehicle** and **IManufacturer** interfaces is displayed in Code Snippet 4.

Code Snippet 4:

```

package session8;
class TwoWheeler implements IVehicle, IManufacturer {
    String ID; // variable to store vehicle ID
    String type; // variable to store vehicle type
    /**
     * Parameterized constructor to initialize values based on user input
     * @param ID a String variable storing vehicle ID
     * @param type a String variable storing vehicle type
     */
    public TwoWheeler(String ID, String type) {
        this.ID = ID;
        this.type = type;
    }
    /**
     * Overridden method, starts a vehicle
     *
     * @return void
     */
    @Override
    public void start() {
        System.out.println("Starting the "+ type);
    }
}

```

```
/**  
 * Overridden method, accelerates a vehicle  
 * @param speed an integer storing the speed  
 * @return void  
 */  
@Override  
public void accelerate(int speed) {  
    System.out.println("Accelerating at speed:"+speed+ " kmph");  
}  
/**  
 * Overridden method, applies brake to a vehicle  
 *  
 * @return void  
 */  
@Override  
public void brake() {  
    System.out.println("Applying brakes...");  
}  
/**  
 * Overridden method, stops a vehicle  
 *  
 * @return void  
 */  
@Override  
public void stop() {  
    System.out.println("Stopping the "+ type);  
}  
/**  
 * Displays vehicle details  
 *  
 * @return void  
 */  
public void displayDetails()  
{  
    System.out.println("Vehicle No.:"+ STATEID+ " "+ ID);  
    System.out.println("Vehicle Type.: "+ type);  
}
```

```
// Implement the IManufacturer interface methods
/**
 * Overridden method, adds manufacturer details
 * @param detail a String variable storing manufacturer detail
 * @return void
 */
@Override
public void addContact(String detail) {
    System.out.println("Manufacturer: "+detail);
}

/**
 * Overridden method, calls the manufacturer
 * @param phone a String variable storing phone number
 * @return void
 */
@Override
public void callManufacturer(String phone) {
    System.out.println("Calling Manufacturer @: "+phone);
}

/**
 * Overridden method, makes payment
 * @param amount a String variable storing the amount
 * @return void
 */
@Override
public void makePayment(float amount) {
    System.out.println("Payable Amount: $" +amount);
}

}
*/
*/
* Define the class TestVehicleNew and save the file as TestVehicleNew.java
*
*/
public class TestVehicleNew {
/*
 * @param args the command line arguments

```

```

*/
public static void main(String[] args) {
    // Verify number of command line arguments
    if(args.length==6) {
        // Instantiate the class
        TwoWheeler objBike = new TwoWheeler(args[0], args[1]);
        objBike.displayDetails();
        objBike.start();
        objBike.accelerate(Integer.parseInt(args[2]));
        objBike.brake();
        objBike.stop();
        objBike.addContact(args[3]);
        objBike.callManufacturer(args[4]);
        objBike.makePayment(Float.parseFloat(args[5]));
    }
    else{
        // Display an error message
        System.out.println("Usage: java TwoWheeler <ID> <Type> <Speed>
<Manufacturer> <Phone> <Amount>");
    }
}
}
}

```

The class **TwoWheeler** now implements both the interfaces; **IVehicle** and **IManufacturer**. Also, it implements all the methods of both the interfaces.

Figure 8.3 shows the output of the code. The user passes **CS-2737 Bike 80 BN-Bikes 808-283-2828 300** as command line arguments.

```

run:
Vehicle No.: LA-09 CS-2737
Vehicle Type.: Bike
Starting the Bike
Accelerating at speed:80 kmph
Applying brakes...
Stopping the Bike
Manufacturer: BN-Bikes
Calling Manufacturer @: 080-283-2828
Payable Amount: $300.0

```

Figure 8.3: Output of **TestVehicleNew.java** Class

Notice that the interface **IManufacturer** can be implemented by other classes such as **FourWheeler**, **Furniture**, **Jewelry**, and so on, that require manufacturer information.

8.2.3 Private Methods in Interfaces

Private methods are visible only inside the interface that they are declared in, so it is recommended to use them for sensitive code.

There are several guidelines for using private methods in interfaces, some of which are as follows:

- Private interface methods cannot be abstract and you cannot use private and abstract modifiers together.
- Private methods can be used only inside an interface and other static and non-static interface methods.
- Private non-static methods cannot be used inside private static methods.

Consider Code Snippet 5:

Code Snippet 5:

```
package session8;
interface Person{
    default void display1(String msg) {
        msg+=" from display1";
        printMessage(msg);
    }
    default void display2(String msg) {
        msg+=" from display2";
        printMessage(msg);
    }
    private void printMessage(String msg) {
        System.out.println(msg);
    }
}
public class Employee implements Person {
    public void printInterface() {
        display1("Hello there");
        display2("Hi there");
    }
    public static void main(String[] args) {
        Employee objEmployee = new Employee();
        objEmployee.printInterface();
    }
}
```

Here, **Person** is an interface that declares two default methods, **display1** and **display2** respectively and one private method, **printMessage**. This private method can only be called within **display1** or **display2** and not outside of the interface. In class **Employee**, which implements **Person**, a method **printInterface** is defined that in turn calls **display1** and **display2** with appropriate String parameters. Thus, the method **printMessage** can be made to contain some action and at the same time be restricted to the interface only. Key benefits of private methods in interfaces are that you can implement code reusability and also restrict the access of specific methods implementations to other classes or interfaces.

8.2.4 Understanding the Concept of Abstraction

Abstraction is an essential element of object-oriented programming. In Java, it is defined as the process of hiding the unnecessary details and revealing only the essential features of an object to the user. Abstraction is a concept that is used by classes that consist of attributes and methods that perform operations on these attributes.

Abstraction can also be achieved through composition. For example, a class **Vehicle** is composed of an engine, tires, ignition key, and many other components. To construct the class **Vehicle**, one is not required to know about the internal working of different components, but only know how to interact or interface with them. That is, send and receive messages to and from them as well as make different objects that compose the **Vehicle** class interact with each other.

In Java, abstract classes and interfaces are used to implement the concept of abstraction. An abstract class or interface is not concrete. In other words, it is incomplete. To use an abstract class or interface one requires to extend or implement abstract methods with concrete behavior according to the context in which it is used.

Abstraction is used to define an object based on its attributes, functionality, and interface.

Differences between an abstract class and an interface are listed in Table 8.1.

Abstract Class	Interface
An abstract class may have non-final variables.	Variables declared in an interface are implicitly final.
An abstract class may have members with different access specifiers such as <code>private</code> , <code>protected</code> , and so on.	Members of an interface are <code>public</code> by default.
An abstract class is inherited using <code>extends</code> keyword.	An interface is implemented using <code>implements</code> keyword.
An abstract class can inherit from another class and implement multiple interfaces.	An interface can extend from one or more interfaces.

Table 8.1: Abstract Class versus Interfaces

Abstraction and Encapsulation are two important object-oriented programming concepts in Java. Both concepts are completely different from each other. Abstraction refers to bringing out the behavior from 'How exactly' it is implemented. Whereas Encapsulation refers to hiding details of implementation from the outside world so as to ensure that any change to a class does not affect the dependent classes. Some of the differences between the two concepts are as follows:

- Abstraction is implemented using an interface and an abstract class whereas Encapsulation is implemented using `private`, `default` or `package-private`, and `protected` access modifier.
- Encapsulation is also referred to as data hiding.
- The basis of the design principle 'programming for interface than implementation' is abstraction and that of 'encapsulate whatever changes' is encapsulation.

8.3 Nested Class

Java allows defining a class within another class. Such a class is called a nested class as shown in Figure 8.4.

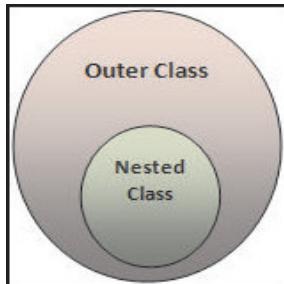


Figure 8.4: Nested Class

Code Snippet 6 defines a nested class.

Code Snippet 6:

```
class Outer{
...
class Nested{
...
}
```

The class **Outer** is the external enclosing class and the class **Nested** is the class defined within the class **Outer**.

Nested classes are classified as static and non-static. Nested classes that are declared `static` are simply termed as static nested classes whereas non-static nested classes are termed as inner classes. This has been demonstrated in Code Snippet 7.

Code Snippet 7:

```
class Outer{
...
static class StaticNested{
...
}
class Inner{
...
}
```

Notice that the class **StaticNested** is a nested class that has been declared as `static` whereas the non-static nested class, **Inner**, is declared without the keyword `static`.

A nested class is a member of its enclosing class. Note that non-static nested classes or inner classes can access the members of the enclosing class even when they are declared as `private`. On the other hand, `static` nested classes cannot access any other member of the enclosing class. As a member of the outer class, a nested class can have any access specifier such as `public`, `private`, `protected`, or `default` (package private).

8.3.1 Benefits of Using Nested Class

The reasons for introducing this advantageous feature of defining nested class in Java are as follows:

- **Creates logical grouping of classes:** If a class is of use to only one class, then it can be embedded within that class and the two classes can be kept together. In other words, it helps in grouping the related functionality together. Nesting of such 'helper classes' helps to make the package more efficient and streamlined.
- **Increases encapsulation:** In case of two top level classes such as class A and B, when B wants access to members of A that are `private`, class B can be nested within class A so that B can access the members declared as `private`. Also, this will hide class B from the outside world. Thus, it helps to access all the members of the top-level enclosing class even if they are declared as `private`.
- **Increased readability and maintainability of code:** Nesting of small classes within larger top-level classes helps to place the code closer to where it will be used.

Different types of nested classes are as follows:

- Member classes or non-static nested classes
- Local classes
- Anonymous classes
- Static Nested classes

8.3.2 Member Classes

A member class is a non-static inner class. It is declared as a member of the outer or enclosing class. The member class cannot have `static` modifier since, it is associated with instances of the outer class. An inner class can directly access all members that is, fields and methods of the outer class including the `private` ones. However, the reverse is not true. That is, the outer class cannot access members of the inner class directly even if they are declared as `public`. This is because members of an inner class are declared within the scope of inner class.

An inner class can be declared as `public`, `private`, `protected`, `abstract`, or `final`. Instances of an inner class exist within an instance of the outer class. To instantiate an inner class, one must create an instance of the outer class.

Then, one can access the inner class object within the outer class object using the statement defined in Code Snippet 8.

Code Snippet 8:

```
// accessing inner class using outer class object
Outer.Inner objInner = objOuter.new Inner();
```

Code Snippet 9 describes an example of non-static inner class.

Code Snippet 9:

```
package session8;
class Server {
    String port; // variable to store port number
    /**
     * Connects to specified server
     * @param IP a String variable storing IP address of server
     * @param port a String variable storing port number of server
     * @return void
     */
    public void connectServer(String IP, String port) {
        System.out.println("Connecting to Server at:" + IP + ":" + port);
    }
    /**
     * Define an inner class
     */
    class IPAddress
    {
        /**
         * Returns the IP address of a server
         * @return String
         */
        String getIP() {
            return "101.232.28.12";
        }
    }
}
/**
 * Define the class TestConnection and save the code as TestConnection.java
 */
public class TestConnection {
```

```

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    // Check the number of command line arguments
    if(args.length==1) {
        // Instantiate the outer class
        Server objServer1 = new Server();
        // Instantiate the inner class using outer class object
        Server.IPAddress objIP = objServer1.new IPAddress();
        // Invoke connectServer() method with the IP returned from getIP() method
        // of the inner class
        objServer1.connectServer(objIP.getIP(),args[0]);
    }
    else {
        System.out.println("Usage: java Server <port-no>");
    }
}
}

```

The class **Server** is an outer class that consists of a variable **port** that represents the port at which the server will be connected. Also, the **connectServer(String, String)** method accepts the IP address and port number as a parameter. The inner class **IPAddress** consists of the **getIP()** method that returns the IP address of the server.

The **main()** method is defined in the class **TestConnection**. Within **main()** method, the number of arguments is verified and accordingly the object of **Server** class is created. Next, the object **objServer1** is used to create the object, **objIP**, of inner class **IPAddress**. Lastly, the outer class object is used to invoke the **connectServer()** method. The IP address is retrieved using **objIP.getIP()** statement. Figure 8.5 shows the output of the code when user provides '8080' as the port number at command line.

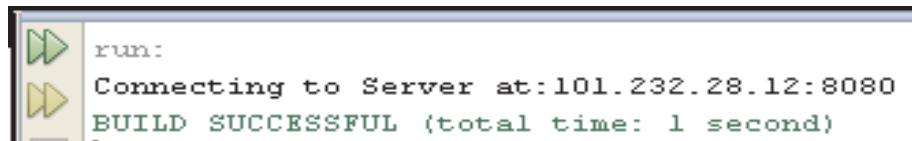


Figure 8.5: Output of `Server.java` Class Using Inner Class

8.3.3 Local Class

An inner class defined within a code block such as the body of a method, constructor, or initializer is termed as a local inner class. The scope of a local inner class is only within that particular block. Unlike an inner class, a local inner class is not a member of the outer class and therefore, it cannot have any access

specifier. That is, it cannot use modifiers such as `public`, `protected`, `private`, or `static`. However, it can access all members of the outer class as well as `final` variables declared within the scope in which it is defined. Figure 8.6 displays a local inner class.

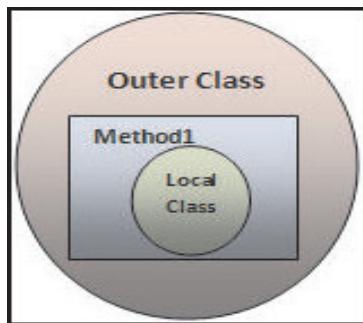


Figure 8.6: Local Inner Class

Local inner class has following features:

- It is associated with an instance of the enclosing class.
- It can access any members, including private members, of the enclosing class.
- It 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`.

Code Snippet 10 demonstrates an example of local inner class.

Code Snippet 10:

```
package session8;
class Employee {
    /**
     * Evaluates employee status
     * @param empID a String variable storing employee ID
     * @param empAge an integer variable storing employee age
     * @return void
     */
    public void evaluateStatus(String empID, int empAge) {
        // local final variable
        final int age=40;
        /**
         * Local inner class Rank
         *
         */
        class Rank{
        /**
         * Returns the rank of an employee
         */
    }
}
```

```

* @param empID a String variable that stores the employee ID
 * @return char
 */
public char getRank(String empID) {
    System.out.println("Getting Rank of employee: "+ empID);
    // assuming that rank 'A' was returned from server
    return 'A';
}

// Check the specified age
if(empAge>=age) {
    // Instantiate the Rank class
    Rank objRank = new Rank();
    // Retrieve the employee's rank
    char rank = objRank.getRank(empID);
    // Verify the rank value
    if(rank == 'A') {
        System.out.println("Employee rank is:"+ rank);
        System.out.println("Status: Eligible for upgrade");
    }
    else{
        System.out.println("Status: Not Eligible for upgrade");
    }
}
else{
    System.out.println("Status: Not Eligible for upgrade");
}
}

/**
 * Define the class TestEmployee and save the code as TestEmployee.java
 */
public class TestEmployee {
    /**
     * @param args the command line arguments

```

```

*/
public static void main(String[] args)
{
    if(args.length==2) {
        // Object of outer class
        Employee objEmp1 = new Employee();
        // Invoke the evaluateStatus() method
        objEmp1.evaluateStatus(args[0], Integer.parseInt(args[1]));
    }
    else{
        System.out.println("Usage: java Employee <Emp-Id> <Age>");
    }
}
}

```

The class **Employee** is the outer class with a method named **evaluateStatus (String, int)**. The method consists of a local **final** variable named **age**. The class **Rank** is a local inner class within the method. The **Rank** class consists of one method **getRank()** that returns the rank of the specified employee Id.

Next, if the age of the employee is greater than **40**, the object of **Rank** class is created and the rank is retrieved. If the rank is equal to '**A**' then, the employee is eligible for upgrade otherwise the employee is not eligible.

The **main()** method is defined in the **TestEmployee** class. Within the **main()** method, an object **objEmp1** of class **Employee** is created and the method **evaluateStatus ()** is invoked with appropriate arguments.

Figure 8.7 shows the output of the code when user passes '**E001**' as employee Id and **50** for age.

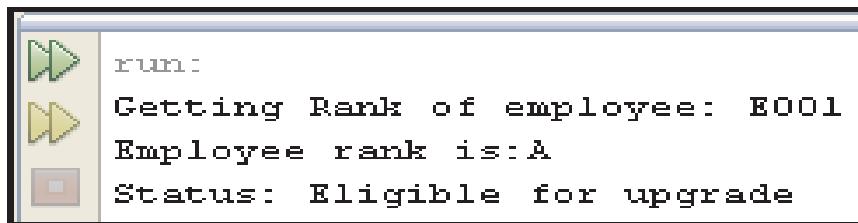


Figure 8.7: Output of **Employee.java** Class Using Local Inner Class

8.3.4 Anonymous Class

An inner class declared without a name within a code block such as the body of a method is called an anonymous inner class. Since, an anonymous class does not have a name associated, it can be accessed only at the point where it is defined.

Anonymous class is a type of local class that cannot use the `extends` and `implements` keywords nor can specify any access modifiers, such as `public`, `private`, `protected`, and `static`. It cannot define a constructor, `static` fields, methods, or classes. Also, it cannot implement anonymous interfaces because an interface cannot be implemented without a name.

Since, an anonymous class does not have a name, it cannot have a named constructor, but it can have an instance initializer. Rules for accessing an anonymous class are the same as that of local inner class.

Usually, anonymous class is an implementation of its super class or interface and contains the implementation of the methods. Anonymous inner classes have a scope limited to the outer class. They can access the internal or `private` members and methods of the outer class. Anonymous class is useful for controlled access to the internal details of another class. Also, it is useful when a user wants only one instance of a special class.

Figure 8.8 displays an anonymous class.

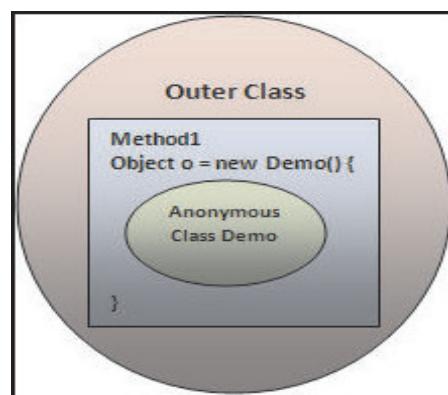


Figure 8.8: Anonymous Inner Class

Code Snippet 11 describes an example of anonymous class.

Code Snippet 11:

```
package session8;
class Authenticate {
    /**
     * Define an anonymous class
     *
     */
    Account objAcc = new Account () {
        /**
         * Displays balance
         *
         * @param accNo a String variable storing balance
         * @return void
    }
}
```

```
/*
@Override
public void displayBalance(String accNo) {
    System.out.println("Retrieving balance. Please wait..."); 
    // Assume that the server returns 40000
    System.out.println("Balance of account number " + accNo.toUpperCase() + " 
is $40000");
}
} ; // End of anonymous class
}

/**
 * Define the Account class
 *
 */
class Account {
    /**
     * Displays balance
     *
     * @param accNo a String variable storing balance
     * @return void
     */
    public void displayBalance(String accNo) {
    }
}

/**
 * Define the TestAuthentication class
 *
 */
public class TestAuthentication {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the Authenticate class
    }
}
```

```
Authenticate objUser = new Authenticate();
// Check the number of command line arguments
if (args.length == 3) {
    if (args[0].equals("admin") && args[1].equals("abc@123")) {
        // Invoke the displayBalance() method
        objUser.objAcc.displayBalance(args[2]);
    }
} else{
    System.out.println("Unauthorized user");
}
else {
    System.out.println("Usage: java Authenticate <user-name>
<password><account-no>");
}
}
```

The class **Authenticate** consists of an anonymous object of type **Account**. The **displayBalance(String)** method is used to retrieve and display the balance of the specified account number. The **main()** method is defined in the **TestAuthentication** class. Within **main()** method, an object of class **Authenticate** is created. The number of command line arguments is verified. If the number of arguments are correct, the username and password are verified. The object of anonymous class is used to invoke the **displayBalance()** method with account number as the argument.

Figure 8.9 shows the output of the code when user passes 'admin', 'abc@123', and 'akdle26152', as arguments.

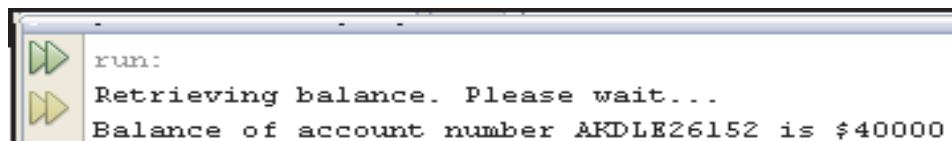


Figure 8.9: Output of Authenticate.java That Uses Anonymous Class

8.3.5 Static Nested Class

A static nested class is associated with the outer class just like variables and methods. A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but can access only through an object reference. A static nested class, by behavior, is a top-level class that has been nested in another top-level class for packaging convenience. Static nested classes are accessed using the fully qualified class name, that is, `OuterClass.StaticNestedClass`.

A static nested class can have public, protected, private, default or package private, final, and abstract access specifiers. Code Snippet 12 demonstrates the use of static nested class.

Code Snippet 12:

```
package session8;
import java.util.Calendar;
class AtmMachine {
    /**
     * Define the static nested class
     *
     */
    static class BankDetails {
        // Instantiate the Calendar class of java.util package
        static Calendar objNow = Calendar.getInstance();
        /**
         * Displays the bank and transaction details
         *
         * @return void
         */
        public static void printDetails() {
            System.out.println("State Bank of America");
            System.out.println("Branch: New York");
            System.out.println("Code: K3983LKSIE");
            // retrieving current date and time using Calendar object
            System.out.println("Date-Time:" + objNow.getTime());
        }
    }
    /**
     * Displays balance
     * @param accNo a String variable that stores the account number
     * @return void
     */
    public void displayBalance(String accNo) {
        // Assume that the server returns 200000
        System.out.println("Balance of account number " + accNo.toUpperCase() +
            " is $200000");
    }
}
/**
```

```

* Define the TestATM class
*
*/
public class TestATM {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        if(args.length==1) { // verifying number of command line arguments
            // Instantiate the outer class
            AtmMachine objAtm = new AtmMachine();
            // Invoke the static nested class method using outer class object
            AtmMachine.BankDetails.printDetails();
            // Invoke the instance method of outer class
            objAtm.displayBalance(args[0]);
        }
        else{
            System.out.println("Usage: java AtmMachine <account-no>");
        }
    }
}

```

The class **AtmMachine** consists of a static nested class named **BankDetails**. The static nested class creates an object of the **Calendar** class of **java.util** package. The **Calendar** class consists of built-in methods to set or retrieve the system date and time. The **printDetails()** method is used to print the bank details along with the current date and time using the **getTime()** method of **Calendar** class.

The **main()** method is defined in the **TestATM** class. Within **main()** method, the number of command line arguments is verified and accordingly an object of class **AtmMachine** is created. Next, the static method **printDetails()** of the nested class **BankDetails** is invoked and accessed directly using the class name of the outer class **AtmMachine**. Lastly, the object **objAtm** of outer class is used to invoke **displayBalance()** method of the outer class with account number as the argument.

Figure 8.10 shows the output of the code when user passes '**akdle26152**' as account number.

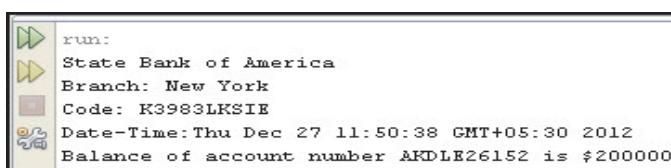


Figure 8.10: Output of **AtmMachine.java** That Uses Static Nested Class

Notice that the output of date and time shows the default format as specified in the implementation of the `getTime()` method. `SimpleDateFormat` from the `java.text` package formats and parses dates in a locale-specific way, adaptable to user requirements. `SimpleDateFormat` class allows specifying user-defined patterns for date-time formatting. The modified `BankDetails` class using `SimpleDateFormat` class is displayed in Code Snippet 13.

Code Snippet 13:

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
class AtmMachine {
    /**
     * Define the static nested class
     *
     */
    static class BankDetails
    {
        // Instantiate the Calendar class of java.util package
        static Calendar objNow = Calendar.getInstance();
        /**
         * Displays the bank and transaction details
         *
         * @return void
         */
        public static void printDetails()
        {
            System.out.println("State Bank of America");
            System.out.println("Branch: New York");
            System.out.println("Code: K3983LKSIE");
            // Format the output of date-time using SimpleDateFormat class
            SimpleDateFormat objFormat = new SimpleDateFormat ("dd/MM/yyyy HH:mm:ss");
            // Retrieve the current date and time using Calendar object
            System.out.println("Date-Time:" + objFormat.format(objNow.getTime()));
        }
    }
    public void displayBalance(String accNo)
    {
        // Assume that the server returns 200000
    }
}
```

```

        System.out.println("Balance of account number " + accNo.toUpperCase() +
        " is $200000");
    }
}

```

The `SimpleDateFormat` class constructor takes the date pattern as a `String`. In Code Snippet 13, the pattern '`dd/MM/yyyy HH:mm:ss`' uses several symbols that are pattern letters recognized by the `SimpleDateFormat` class. Table 8.2 lists the pattern letters used in the code with their description.

Pattern Letter	Description
d	Day of the month
M	Month of the year
Y	Year
H	Hour of a day (0-23)
m	Minute of an hour
s	Second of a minute

Table 8.2: Date Pattern Letters

Figure 8.11 shows the output of the modified code.

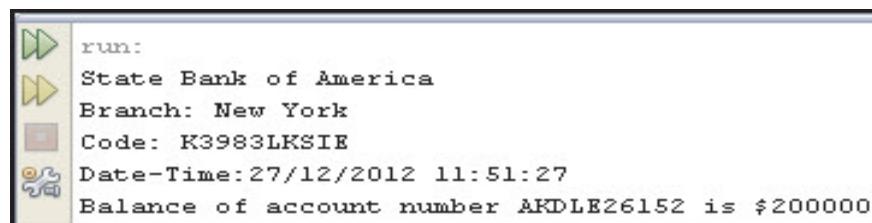


Figure 8.11: Output of Modified `AtmMachine` Class Using `SimpleDateFormat`

Notice that the date and time are now displayed in the specified format that is more understandable to the user.

8.4 Check Your Progress

1. An interface is saved with a _____ extension.

(A)	.interface	(C)	.jar
(B)	.exe	(D)	.java

2. Which of the following statements about an interface are true?

a.	An interface can contain multiple methods	c.	All the methods of an interface are implicitly abstract
b.	An interface can be instantiated	d.	An interface can have constructors

(A)	b, c	(C)	b, d
(B)	a, d	(D)	a, c

3. Match following description with the corresponding nested class type.

	Description		Nested Class Type
a.	A non-static inner class declared as a member of the outer or enclosing class	1.	Local Inner Class
b.	An inner class declared without a name within a code block such as the body of a method	2.	Member Class
c.	An inner class defined within a code block such as the body of a method, constructor, or an initialize	3.	Static Nested Class
d.	Cannot directly refer to instance variables or methods of the outer class but only through an object reference	4.	Anonymous Class

(A)	a-3, b-4, c-1, d-2	(C)	a-4, b-3, c-2, d-1
(B)	a-2, b-4, c-1, d-3	(D)	a-2, b-3, c-4, d-1

4. Consider following code:

```

public interface Publisher {
    public void addContact();
    public void getRoyaltyAmt();
}

public class Book implements Publisher {
    String ID;
    String type;
    public Book(String ID, String type) {
        this.ID = ID;
        this.type = type;
    }
    @Override
    public void addContact() {
        System.out.println("Storing contact .....");
    }
    @Override
    public void getRoyaltyAmt() {
        System.out.println("Royalty amount is $2000");
    }
    public void displayDetails() {
        System.out.println("Book ID: " + ID);
        System.out.println("Book Type.: " + type);
    }
    public static void main(String[] args) {
        if (args.length == 2) {
            Book objBook1 = new Book(args[0], args[1]);
            objBook1.displayDetails();
            objBook1.addContact();
            objBook1.getRoyaltyAmt();
        } else {
            System.out.println("Usage: java Book <ID> <Type>");
        }
    }
}

```

What will be the output of the code when user passes 'B001' as the command line argument?

(A)	Compilation Error	(C)	Usage: java Book <ID> <Type>
(B)	Book ID: B001 Book Type: Storing contact Royalty amount is \$2000	(D)	Runtime Error

5. Identify the correct syntax to implement multiple interfaces in a class.

(A)	class <class-name> implements <Interface1> <Interface2> ...	(C)	class <class-name> implements <Interface1>: <Interface2>:...
(B)	class <class-name> implements <Interface1>, <Interface2>, ...	(D)	class <class-name> implements <Interface1>& <Interface2>& ...

6. Consider following code:

```
class Bird
{
    public void moveBird()
    {
        Object o = new Fly() {
            public void flightSpeed(int speed)
            {
                System.out.println("Flying at "+ speed + "kmph");
            }
        };
    }

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

class Fly
{
    public void flightSpeed(int speed)
    {}
}
```

Which type of nested class is used in the code?

(A)	Anonymous class	(C)	Member class
(B)	Local Inner class	(D)	Static Nested class

8.4.1 Answers

1.	D
2.	D
3.	B
4.	C
5.	B
6.	A

```
g package;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Summary

- An interface in Java is a contract that specifies the standards to be followed by the types that implement it.
- To implement multiple interfaces, write the interfaces after the implements keyword separated by a comma.
- Abstraction, in Java, is defined as the process of hiding the unnecessary details and revealing only the necessary details of an object.
- Java allows defining a class within another class. Such a class is called a nested class.
- A member class is a non-static inner class. It is declared as a member of the outer or enclosing class.
- An inner class defined within a code block such as the body of a method, constructor, or an initializer, is termed as a local inner class.
- An inner class declared without a name within a code block such as the body of a method is called an anonymous inner class.
- A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but only through an object reference.

Try It Yourself

1. **Imaginations Pvt. Ltd.** is a famous graphics design and animation company located in **New York, USA**. The company has started a new IT division wherein they recruit programmers to create customized graphic software. Currently, the company is working on an animation software. The project manager has assigned various modules to different programmers according to their skill.

Jack, one of the Java programmers, has been assigned the task to create a code for drawing and animating shapes according to specifications given by user. Jack has written following sample code to accomplish the task.

```
public abstract class Shape {
    public abstract void drawShape();
    public void calcArea(float val) {
        System.out.println("Area of Shape");
    }
}

public abstract class IAnimate {
    void rotateObject(int degree);
    void flipObject(int direction);
    void moveObject(int distance);
}

public class Circle extends Shape, IAnimate{
    @Override
    public void calcArea(float rad) {
        System.out.println("Area of Circle is: "+ (3.14*rad*rad));
    }
    @Override
    public void rotateObject(int degree) {
        System.out.println("Rotating Circle by "+ degree +" degrees");
    }
    @Override
    public void flipObject(int direction) {
```

Try It Yourself

```

        g.fillRect(100, 100, 200, 200);
        g.drawString("Hello", 150, 150);
    }
}

public class Circle {
    public void drawShape() {
        System.out.println("Drawing Circle ...");
    }

    public void calcArea(String args[]) {
        double radius = Double.parseDouble(args[0]);
        double area = 3.14 * radius * radius;
        System.out.println("Area of Circle is: " + area);
    }

    public void rotateObject(String args[]) {
        System.out.println("Rotating Circle by " + args[0] + " degrees");
    }
}

```

Jack wishes to use the characteristics of both the **Shape** and **Animate** classes into the **Circle** class. However, the code is generating compilation errors and not functioning properly.

Fix the code to get following output:

Drawing Circle ...
 Area of Circle is: 1256.0
 Rotating Circle by 30 degrees

Also, if the user does not specify the required number of arguments, appropriate message must be displayed.

Session - 9

Exceptions

Welcome to the Session, **Exceptions**.

This session explains the concept of exception and types of errors and exceptions. Further, this session describes the Exception class and process of exception handling. The session also explains try-catch and finally blocks. Finally, the session explains execution flow of exceptions and guidelines for exception handling.

In this Session, you will learn to:

- Describe exceptions
- Explain types of errors and exceptions
- Elaborate the Exception class
- Describe exception handling
- Explain try-catch block
- Explain finally block
- Explain execution flow of exceptions
- Summarize guidelines for exception handling



9.1 Introduction

Java is a very robust and efficient programming language. Features such as classes, objects, inheritance, and so on make Java a strong, versatile, and secure language. However, no matter how well a code is written, it is prone to failure or behaves erroneously in certain conditions. These situations may be expected or unexpected. In either case, the user would be confused with such unexpected behavior of code.

To avoid such a situation, Java provides the concept of exception handling using which, a programmer can display appropriate message to the user in case such unexpected behavior of code occurs.

9.2 Exceptions

An exception is an event or an abnormal condition in a program occurring during execution of a program that leads to disruption of normal flow of program instructions. Exceptions may arise due to various factors, including user inputting invalid data, missing essential files, disrupted network connections, or JVM memory exhaustion. When an error occurs inside a method, it creates an exception object and passes it to the runtime system. This object holds information about the type of error and state of the program when the error occurred.

This process of creating an exception object and passing it to the runtime system is termed as throwing an exception. After an exception is thrown by a method, the runtime system tries to find some code block to handle it. The possible code blocks where the exception can be handled are a series of methods that were invoked in order to reach the method where the error has actually occurred. This list or series of methods is called the call stack. In other words, the stack trace shows the sequence of method invocations that led up to the exception.

Figure 9.1 shows an example of method call stack.

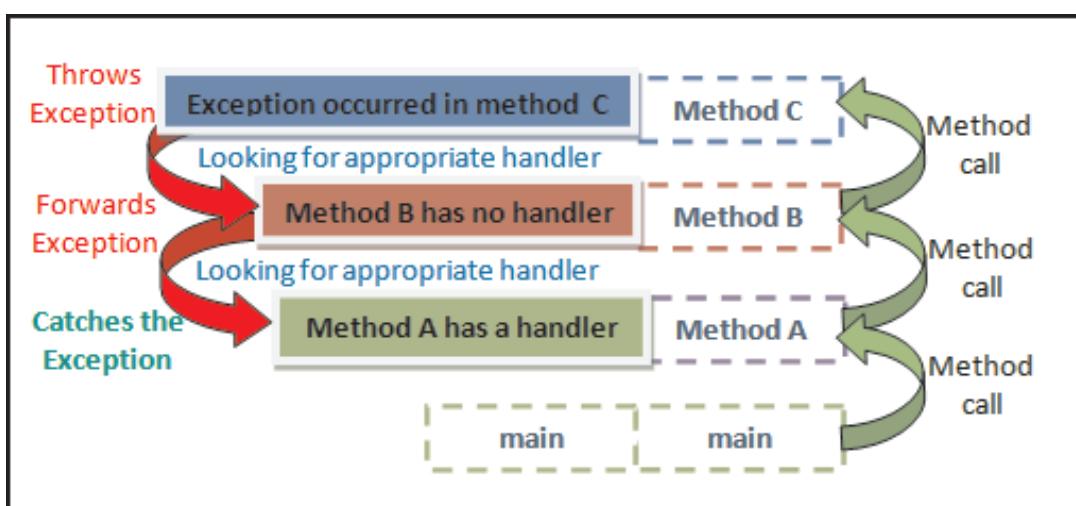


Figure 9.1: Runtime Call Stack and Exception Handling

Figure 9.1 shows the method call from **main** → **Method A** → **Method B** → **Method C**. When an exception occurs in method C, it throws the exception object to the runtime environment. The runtime environment then searches the entire call stack for a method that consists of a code block that can handle the exception. This block of code is called an exception handler.

The runtime environment first searches the method in which the error occurred. If a handler is not found, it proceeds through the call stack in the reverse order in which the methods were invoked. When an appropriate handler is found, the runtime environment passes the exception to the handler.

An appropriate exception handler is one that handles the same type of exception as the one thrown by the method. In this case, the exception handler is said to 'catch' the exception. If while searching the call stack, the runtime environment fails to find an appropriate exception handler, the runtime environment will consequently terminate the program.

An exception is thrown for following reasons:

- A throw statement within a method was executed.
- An abnormality in execution was detected by the JVM, such as:
 - Violation of normal semantics of Java while evaluating an expression such as an integer divided by zero.
 - Error occurring while linking, loading, or initializing part of the program that will throw an instance of a subclass of `LinkageError`.
 - The JVM is prevented from executing the code due to an internal error or resource limitation that will throw an instance of a subclass of `VirtualMachineError`.

These exceptions are not thrown arbitrarily, but at a point where they have been specified as a possible outcome of an expression evaluation or statement execution.

- An asynchronous exception occurred.

The use of exceptions to handle errors offers some advantages as follows:

- **Separate Error-Handling Code from Normal Code:** Exceptions help separating details of what must be done when something wrong happens from the main logic of a program. In conventional programming, error detection, error reporting, and error handling code were written in the same area which often led to confusing and cluttered code. Exceptions on the other hand allow a user to write the main flow of the code while dealing with the exceptional cases elsewhere.
- **Propagate Errors Higher Up in the Call Stack:** Exceptions allow propagation of error reporting up in the call stack of methods. A method that is part of the call stack can evade any exceptions that are thrown within it so that a method further up the call stack can catch it. Thus, only the concerned methods have to worry about detecting specific types of errors.

- **Group the Similar Error Types:** All exceptions that are thrown within a program are objects. Thus, grouping of similar exceptions is an obvious outcome of the class hierarchy. For example, the group of related exception classes defined in `java.io` package such as `IOException` and its descendants. The `IOException` class is the most general exception that represents any type of error occurring while performing Input/Output (I/O) operations. The subclasses of `IOException` represent more specific errors. For example, the subclass `FileNotFoundException` handles the exception when the user attempts to open a file that does not exist.

9.2.1 Types of Errors and Exceptions

As stated earlier, an exception is an abnormal condition arising during program execution. There are several causes for an exception to occur such as invalid data entered by user, and trying to open a file that does not exist. Also, loss of network connection in the middle of a transaction, or the JVM runs out of memory while performing a task may cause an exception. It is clear that some of the exceptions are caused by user error, some by programming error, and others by system resources that failed in some manner.

Based on this observation, Java provides following two types of exceptions:

- **Checked Exceptions:** These are exceptions that a well-written application must anticipate and provide methods to recover from. For example, suppose an application prompts the user to specify the name of a file to be opened and the user specifies the name of a nonexistent file. In such a case, the `java.io.FileNotFoundException` is thrown. However, a well-written program will have the code block to catch this exception and inform the user of the mistake by displaying an appropriate message. In Java, all exceptions are checked exceptions, except those indicated by `Error`, `RuntimeException`, and their subclasses.
- **Unchecked Exceptions:** The unchecked exceptions are as follows:
- **Error:** These are exceptions that are external to the application. The application usually cannot anticipate or recover from errors. For example, suppose the user specified correct file name for the file to be opened and the file exists on the system. However, the runtime fails to read the file due to some hardware or system malfunction. Such a condition of unsuccessful read throws the `java.io.IOException` exception. In this case, the application may catch this exception and display an appropriate message to the user or leave it to the program to print a stack trace and exit. Errors are exceptions generated by `Error` class and its subclasses.
 - **Runtime Exception:** These exceptions are internal to the application and usually the application cannot anticipate or recover from such exceptions. These exceptions usually indicate programming errors, such as logical errors or improper use of an API. For example, suppose a user specified the file name of the file to be opened. However, due to some logical error a `null` is passed to the application, then the application will throw a `NullPointerException`. The application can choose to catch this exception and display appropriate message to the user or eliminate the error that caused the exception to occur. Runtime exceptions are indicated by `RuntimeException` class and its subclasses.

Errors and runtime exceptions are collectively known as unchecked exceptions.

In Java, `Object` class is the base class of the entire class hierarchy. `Throwable` class is the base class of all the exception classes. `Object` class is the base class of `Throwable`. `Throwable` class has two direct subclasses namely, `Exception` and `Error`. The `Throwable` class hierarchy is shown in Figure 9.2.

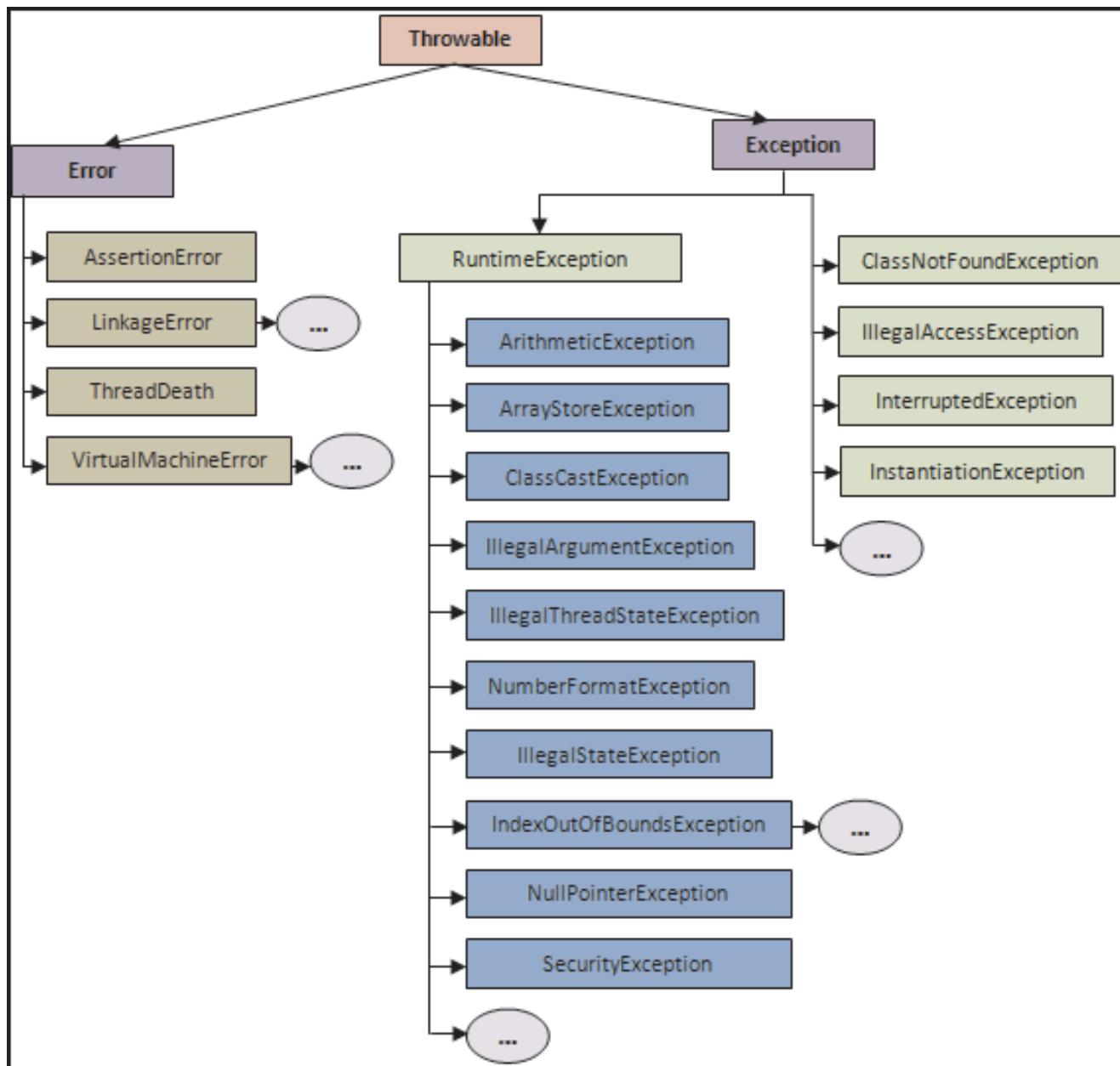


Figure 9.2: `Throwable` Class Hierarchy

Table 9.1 lists some of the checked exceptions.

Exception	Description
<code>InstantiationException</code>	Occurs upon an attempt to create instance of an abstract class.

Exception	Description
InterruptedException	Occurs when a thread is interrupted.
NoSuchMethodException	Occurs when JVM is unable to resolve which method to be invoked.

Table 9.1: Checked Exceptions

Table 9.2 lists some of the commonly observed unchecked exceptions.

Exception	Description
ArithmaticException	Indicates an arithmetic error condition.
ArrayIndexOutOfBoundsException	Occurs if an array index is less than zero or greater than the actual size of the array.
IllegalArgumentException	Occurs if method receives an illegal argument.
NegativeArraySizeException	Occurs if array size is less than zero.
NullPointerException	Occurs on access to a <code>null</code> object member.
NumberFormatException	Occurs if unable to convert the string to a number.
StringIndexOutOfBoundsException	Occurs if index is negative or greater than the size of the string.

Table 9.2: Unchecked Exceptions

9.2.2 Exception Class

The class `Exception` and its subclasses indicate conditions that an application might attempt to handle. The `Exception` class and all its subclasses except `RuntimeException` and its subclasses, are checked exceptions. The checked exceptions must be declared in a method or constructor's `throws` clause if the method or constructor is liable to throw the exception during its execution and propagate it further in the call stack. Code Snippet 1 displays the structure of the `Exception` class.

Code Snippet 1:

```
public class Exception extends Throwable
{
    ...
}
```

Table 9.3 lists the constructors of `Exception` class.

Exception Class Constructor	Description
<code>Exception()</code>	Constructs a new exception with error message set to <code>null</code> .
<code>Exception(String message)</code>	Constructs a new exception with error message set to the specified string <code>message</code> .
<code>Exception(String message, Throwable cause)</code>	Constructs a new exception with error message set to the specified strings <code>message</code> and <code>cause</code> .
<code>Exception(Throwable cause)</code>	Constructs a new exception with the specified <code>cause</code> . The error message is set as per the evaluation of <code>cause == null?null:cause.toString()</code> . That is, if <code>cause</code> is <code>null</code> , it will return <code>null</code> , else it will return the <code>String</code> representation of the message. The message is usually the class name and detail message of <code>cause</code> .

Table 9.3: Exception Class Constructors

`Exception` class provides several methods to get the details of an exception. Table 9.4 lists some of the methods of `Exception` class.

Exception Class Method	Description
<code>public String getMessage()</code>	Returns the details about the exception that has occurred.
<code>public Throwable getCause()</code>	Returns the cause of the exception that is represented by a <code>Throwable</code> object.
<code>public String toString()</code>	If the <code>Throwable</code> object is created with a message string that is not <code>null</code> , it returns the result of <code>getMessage()</code> along with the name of the exception class concatenated to it. If the <code>Throwable</code> object is created with a <code>null</code> message string, it returns the name of the actual class of the object.
<code>public void printStackTrace()</code>	Prints the result of the method, <code>toString()</code> and the stack trace to <code>System.err</code> , that is, the error output stream.
<code>public StackTraceElement [] getStackTrace()</code>	Returns an array where each element contains a frame of the stack trace. The index 0 represents the method at the top of the call stack and the last element represents the method at the bottom of the call stack.
<code>public Throwable fillInStackTrace()</code>	Fills the stack trace of this <code>Throwable</code> object with the current stack trace, adding to any previous information in the stack trace.

Table 9.4: Exception Class Methods

Most of the methods throw and catch objects that derive from the `Exception` class.

An exception indicates that a problem has occurred, but it is not a serious system problem. Majority of the programs are capable of throwing and catching exceptions as opposed to errors.

The Java platform defines many subclasses of the `Exception` class. These subclasses indicate various types of exceptions that can occur. For example, an `IllegalAccessException` indicates that a method that was called could not be found. Similarly, a `NegativeArraySizeException` indicates that a program attempted to create an array with a negative size.

The `RuntimeException` subclass of `Exception` class is reserved for those exceptions that indicate incorrect use of an API. For example, the `NullPointerException` occurs when a method tries to access a member of an object through a `null` reference.

9.3 Handling Exceptions in Java

Any exception that a method is liable to throw is considered to be as much a part of that method's programming interface as its parameters and return value. The code that calls a method must be aware about the exceptions that a method may throw. This helps the caller to decide how to handle them if and when they occur.

More than one runtime exceptions can occur anywhere in a program. Having to add code to handle runtime exceptions in every method declaration may reduce a program's clarity. Thus, the compiler does not require that a user must catch or specify runtime exceptions, although it does not object it either.

A common situation where a user can throw a `RuntimeException` is when the user calls a method incorrectly. For example, a method can check beforehand if one of its arguments is incorrectly specified as `null`. In that case, the method may throw a `NullPointerException`, which is an unchecked exception.

Thus, if a client is capable of reasonably recovering from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

9.3.1 try-catch Block

The first step in creating an exception handler is to identify the code that may throw an exception and enclose it within the `try` block. The syntax for declaring a `try` block is as follows:

Syntax:

```
try{
    // statement 1
    // statement 2
}
```

The statements within the `try` block may throw an exception. Now, when the exception occurs, it is trapped by the `try` block and the runtime looks for a suitable handler to handle the exception. To handle the exception, the user must specify a `catch` block within the method that raised the exception or

somewhere higher in the method call stack.

The syntax for declaring a `try-catch` block is as follows:

Syntax:

```
try{
    // statements that may raise exception
    // statement 1
    // statement 2
}
catch(<exception-type><object-name>) {
    // handling exception
    // error message
}
```

where,

`exception-type`: Indicates the type of exception that can be handled.

`object-name`: Object representing the type of exception.

The `catch` block handles exceptions derived from `Throwable` class.

Code Snippet 2 demonstrates an example of `try` with a single `catch` block.

Code Snippet 2:

```
package session9;
class Mathematics {
    /**
     * Divides two integers
     * @param num1 an integer variable storing value of first number
     * @param num2 an integer variable storing value of second number
     * @return void
     */
    public void divide(int num1, int num2) {
        // Create the try block
        try {
```

```

// Statement that can cause exception
System.out.println("Division is: " + (num1/num2));
}

catch(ArithmaticException e) { //catch block for ArithmaticException
    // Display an error message to the user
    System.out.println("Error: "+ e.getMessage());
}

// Rest of the method
System.out.println("Method execution completed");
}

}

/***
 * Define the TestMath.java class
 */
public class TestMath {

/**
 * @param args the command line arguments
 */
public static void main(String[] args)
{
    // Check the number of command line arguments
    if(args.length==2) {
        // Instantiate the Mathematics class
        Mathematics objMath = new Mathematics();
        // Invoke the divide(int,int) method
        objMath.divide(Integer.parseInt(args[0]), Integer.
parseInt(args[1]));
    }
    else {
        System.out.println("Usage: java TestMath <number1> <number2>");
    }
}
}

```

The class **Mathematics** consists of one method named **divide()** that accepts two integers as parameters and prints the value after division on the screen. However, it is clear that the statement **num1/num2** might raise an error if the user specifies zero for the denominator **num2**. Therefore, the

statement is enclosed within the `try` block. Division being an arithmetic operation, the user can create an appropriate `catch` block with `ArithmeticException` class object.

Within the `catch` block, the `ArithmeticException` class object `e` is used to invoke the `getMessage()` method that will print the detail about the error. The `main()` method is defined in the `TestMath` class. Within the `main()` method, the object of `Mathematics` class is created to invoke the `divide()` method with the parameters specified by the user at command line.

The output of the program when user specifies 12 as numerator and 0 as denominator is shown in Figure 9.3.

```
Output - session9 (run) x
run:
Error: / by zero
Method execution completed
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 9.3: Execution of Catch Block

Figure 9.3 shows that upon execution, the `try` block raised an error as `num2` was specified as 0. The `catch` block was executed and the result of `getMessage()` is displayed to the user.

9.3.2 Execution Flow of Exceptions

In Code Snippet 2, divide-by-zero exception occurs on execution of the statement `num1/num2`. If `try-catch` block is not provided, any code after this statement is not executed as an exception object is automatically created. Since, no `try-catch` block is present, JVM handles the exception, prints the stack trace, and the program is terminated.

Figure 9.4 shows the execution of the code when `try-catch` block is not provided.

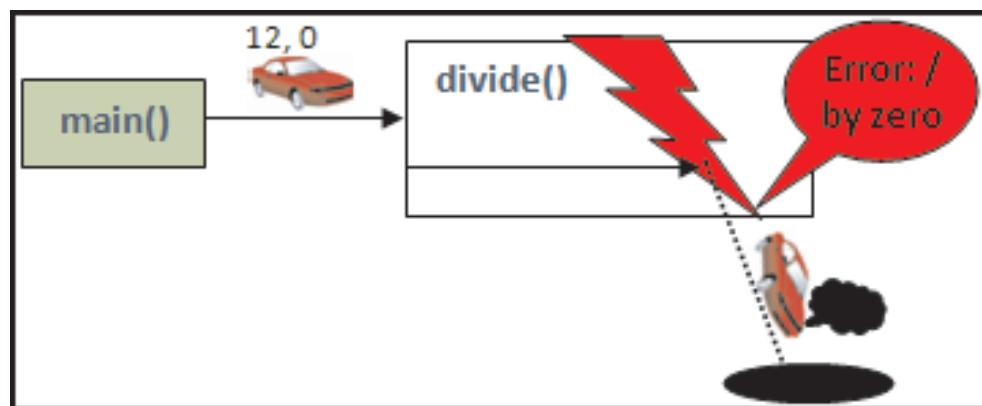


Figure 9.4: Execution Without `try-catch` Block

However, when the `try-catch` block is provided, the divide-by-zero exception occurring in the code is handled by the `try-catch` block and an exception message is displayed. Also, the rest of the code gets executed normally.

Figure 9.5 shows the execution of the code when try-catch block is provided.

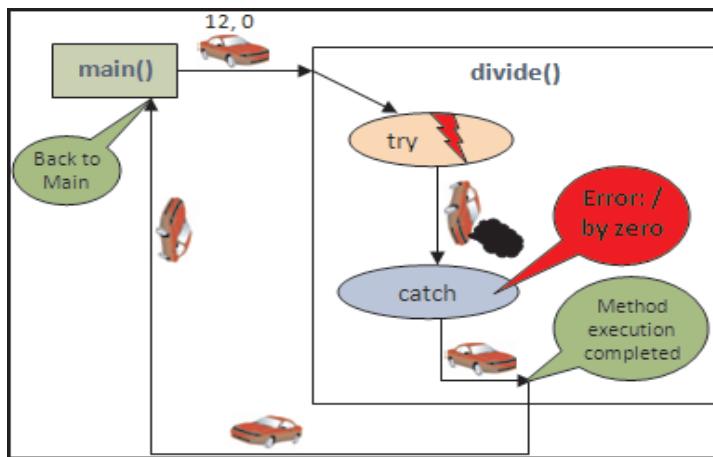


Figure 9.5: Execution of try-catch Block

9.3.3 throw and throws Keywords

Java provides the `throw` and `throws` keywords to explicitly raise an exception in the `main()` method. The `throw` keyword throws the exception in a method. The `throws` keyword indicates the exception that a method may throw.

The `throw` clause requires an argument of `Throwable` instance and raises checked or unchecked exceptions in a method. Code Snippet 3 demonstrates the modified class **Mathematics** now using `throw` and `throws` keywords for handling exceptions.

Code Snippet 3:

```
package session9;

class Mathematics {

    /**
     * Divides two integers, throws ArithmeticException
     * @param num1 an integer variable storing value of first number
     * @param num2 an integer variable storing value of second number
     * @return void
     */

    public void divide(int num1, int num2) throws ArithmeticException {
        // Check the value of num2
        if(num2==0) {
            // Throw the exception
            throw new ArithmeticException("/ by zero");
        }
    }
}
```

```
    }
    else {
        System.out.println("Division is: " + (num1/num2));
    }
    // Rest of the method
    System.out.println("Method execution completed");
}
}

/**
 * Define the TestMathNew class
 */
public class TestMathNew{
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Check the number of command line arguments
        if(args.length==2) {
            // Instantiate the Mathematics class
            Mathematics objMath = new Mathematics();
            try {
                // Invoke the divide(int,int) method
                objMath.divide(Integer.parseInt(args[0]), Integer.
                    parseInt(args[1]));
            }
            catch(ArithmeticException e) {
                // Display an error message to the user
                System.out.println("Error: "+e.getMessage());
            }
        }
        else{
            System.out.println("Usage: java Mathematics <number1> <number2>");
        }
        System.out.println("Back to Main");
    }
}
```

The method `divide(int, int)` now includes the `throws` clause that informs the calling method that `divide(int, int)` may throw an `ArithmeticException`. Within `divide(int, int)`, the code checks for the value of `num2`. If it is equal to zero, it creates an instance of `ArithmeticException` using the `new` keyword with the error message as an argument. The `throw` keyword throws the instance to the caller.

Within the `main()` method, an instance, `objMath` is used to invoke the `divide(int, int)` method. However, this time, the code is written within the `try` block since `divide(int, int)` may throw an `ArithmeticException` that the `main()` method will have to handle within its `catch` block.

Figure 9.6 depicts the output of the program when user specifies 12 as numerator and 0 as denominator.

```
Output - session9 (run) ×
run:
Error: / by zero
Back to Main
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 9.6: Using `throw` and `throws` Keywords

Figure 9.6 shows that upon execution of the code, the `if` condition becomes true and it throws the `ArithmeticException`. The control returns back to the caller, that is, the `main()` method where it is finally handled. The `catch` block was executed and the result of `getMessage()` is displayed to the user. Notice, that the remaining statement of the `divide(int, int)` method is not executed in this case.

Figure 9.7 depicts the execution of the code when `throw` and `throws` clauses are used.

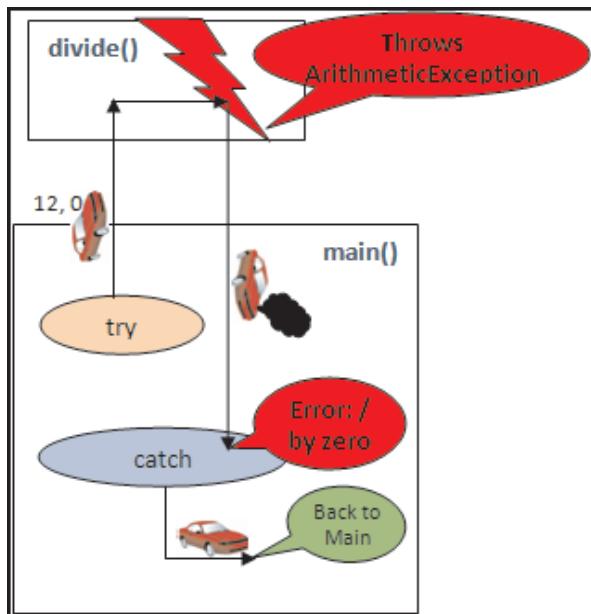


Figure 9.7: Execution of Code Using `throw` and `throws` Keywords

Figure 9.7 shows that the method `divide(int, int)` is invoked within the `try` block in `main()` method. Inside the method, the `ArithmeticException` exception is thrown. The control is then transferred to

the `catch` block of the `main()` method that prints the output of `getMessage()` method and executes the rest of the code of the `main()` method. However, remaining statements of the `divide(int, int)` method are not executed.

9.3.4 Multiple catch Blocks

The user can associate multiple exception handlers with a `try` block by providing more than one `catch` blocks directly after the `try` block. The syntax for declaring a `try` block with multiple `catch` blocks is as follows:

Syntax:

```
try
{ ... }
catch (<exception-type> <object-name>)
{ ... }
catch (<exception-type> <object-name>)
{ ... }
```

In this case, each `catch` block is an exception handler that handles a specific type of exception indicated by its argument `exception-type`. The `catch` block consists of the code that must be executed if and when the exception handler is invoked.

The runtime system invokes the handler in the call stack whose `exception-type` matches the type of the exception thrown. Code Snippet 4 demonstrates the use of multiple `catch` blocks.

Code Snippet 4:

```
package session9;

public class Calculate {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Check the number of command line arguments
        if (args.length == 2) {
            try {
                // Perform the division operation
                int num3 = Integer.parseInt(args[0]) / Integer.parseInt(args[1]);
            } catch (ArithmaticException e) {
                System.out.println("An error occurred: " + e.getMessage());
            }
        }
    }
}
```

```
        System.out.println("Division is: "+num3);
    }

    catch (ArithmetcException e) { // Catch the ArithmetcException
        System.out.println("Error: " + e.getMessage());
    }

    catch (NumberFormatException e) { // Catch the NumberFormatException
        System.out.println("Error: Required Integer found String:" +
e.getMessage());
    }

    catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }

}

else {

    System.out.println("Usage: java Calculate <number1> <number2>");
}

}
```

The class **Calculate** consists of the `main()` method. Within `main()` method, the `try` block performs division of two values specified by the user. As seen earlier, the division may lead to a divide-by-zero error. Therefore, a corresponding `catch` block that handles `ArithmaticException` has been created.

However, one can identify another type of exception also that might be raised, that is, `NumberFormatException`. This may happen if the user specifies a string instead of a number that cannot be converted to integer using `Integer.parseInt()` method. Therefore, another catch block to handle the `NumberFormatException` has been specified.

Notice that the last catch block with Exception class handles any other exception that might occur in the code. Since, Exception class is the parent of all exceptions, it must be used in the last catch block. If Exception class is used with the first catch block, it will handle all the exceptions and other catch blocks, that is, ArithmeticException and NumberFormatException blocks will never be invoked. In other words, when multiple catch blocks are used, catch blocks with exception subclasses must be placed before the super classes, otherwise the super class exception will catch exceptions of the same class as well as its subclasses. Consequently, catch blocks with exception subclasses will never be reached.

Figure 9.8 shows the output of the code when user specifies 12 and 0 as arguments.

```
Output - session9 (run) ×
run:
Error: / by zero
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 9.8: Output with 12 and 0 as Arguments

Figure 9.9 shows the output of the code when user specifies 12 and 'zero' as arguments.

```
Output - session9 (run) ×
run:
Error: Required Integer found String:For input string: "zero"
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 9.9: Output with 12 and zero as Arguments

Figure 9.10 depicts the execution of the code when multiple `catch` blocks are used.

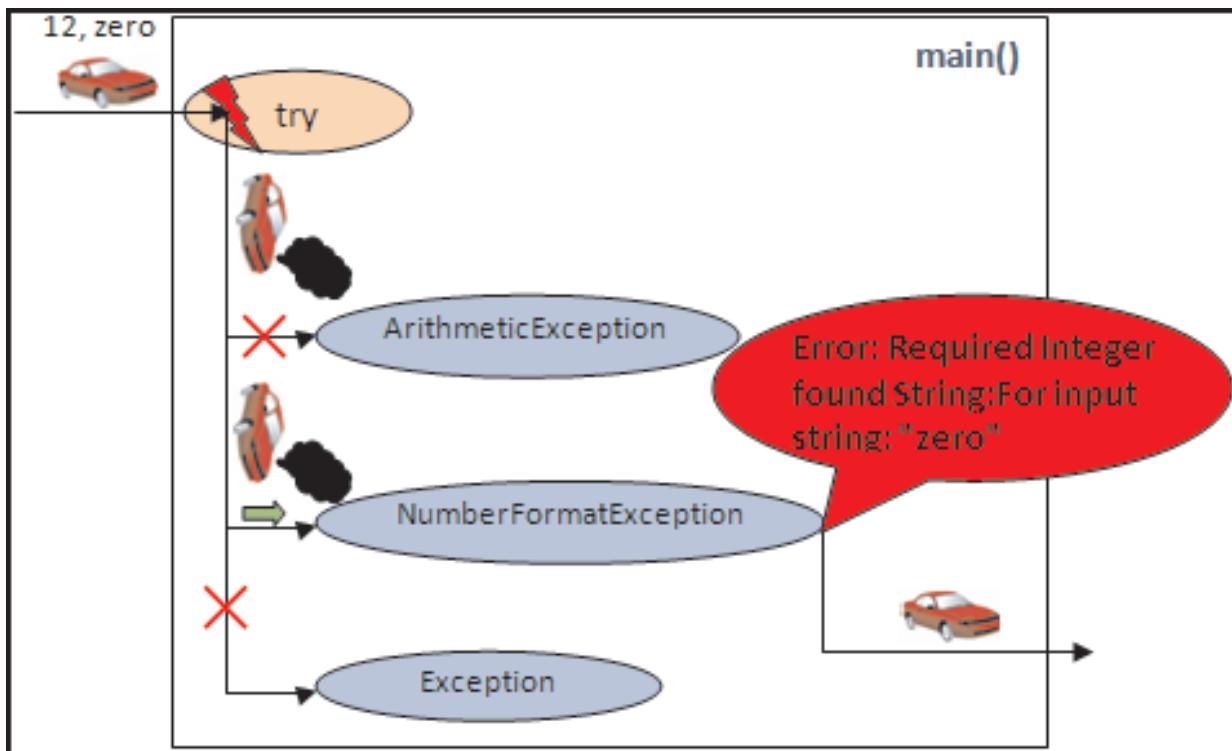


Figure 9.10: Execution of Code Using Multiple `catch` Blocks

Figure 9.10 shows that the first `catch` block is not capable of handling the exception. Therefore, the runtime looks further into another `catch` block. This is also known as bubbling of exception. That is, the exception is propagated further like a bubble.

Here, it finds a matching catch block that handles NumberFormatException. The catch block is executed and appropriate message is displayed to the user. Once a match is found, the remaining catch blocks are ignored, and execution proceeds after the last catch block.

9.3.5 finally Block

Java provides the finally block to ensure execution of certain statements even when an exception occurs. The finally block is always executed irrespective of whether or not an exception occurs in the try block. This ensures that the cleanup code is not accidentally bypassed by a return, break, or continue statement. Thus, writing cleanup code in a finally block is considered a good practice even when no exceptions are anticipated.

The finally block is mainly used as a tool to prevent resource leaks. Tasks such as closing a file and network connection, closing input-output streams, or recovering resources, must be done in a finally block to ensure that a resource is recovered even if an exception occurs.

However, if due to some reason, the JVM exits while executing the try or catch block, then the finally block may not execute. Similarly, if a thread executing the try or catch block gets interrupted or killed, the finally block may not execute even though the application continues to execute.

The syntax for declaring try-catch blocks with a finally block is as follows:

Syntax:

```
try
{
    // statements that may raise exception
    // statement 1
    // statement 2
}
catch(<exception-type> <object-name>)
{
    // handling exception
    // error message
}
finally
{
    // clean-up code
    // statement 1
    // statement 2
}
```

Code Snippet 5 demonstrates the modified class **Calculate** using the **finally** block.

Code Snippet 5:

```
package session9;
public class Calculate {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Check the number of command line arguments
        if (args.length == 2) {
            try {
                int num3 = Integer.parseInt(args[0]) / Integer.parseInt(args[1]);
                System.out.println("Division is: " + num3);
            }
            catch (ArithmaticException e) {
                System.out.println("Error: " + e.getMessage());
            }
            catch (NumberFormatException e) {
                System.out.println("Error: Required Integer found String:" +
                    e.getMessage());
            }
            catch (Exception e) {
                System.out.println("Error: " + e.getMessage());
            }
            finally {
                // Write the clean-up code for closing files, streams, and network
                // connections
                System.out.println("Executing Cleanup Code. Please Wait...");  

                System.out.println("All resources closed.");
            }
        }
        else {
            System.out.println("Usage: java Calculate <number1> <number2>");
        }
    }
}
```

Within the class **Calculate**, finally block is included after the last catch block. In this case, even if an exception occurs in the code, the finally block statements will be executed. Figure 9.11 shows the output of Code Snippet 5 after using finally block when user passes 12 and 'zero' as command line arguments.

```
Output - session9 (run) ×
run:
Error: Required Integer found String:For input string: "zero"
Executing Cleanup Code. Please Wait...
All resources closed.
BUILD SUCCESSFUL (total time: 8 seconds)
```

Figure 9.11: Output After Using finally Block

Figure 9.12 shows the execution of code when finally block is used.

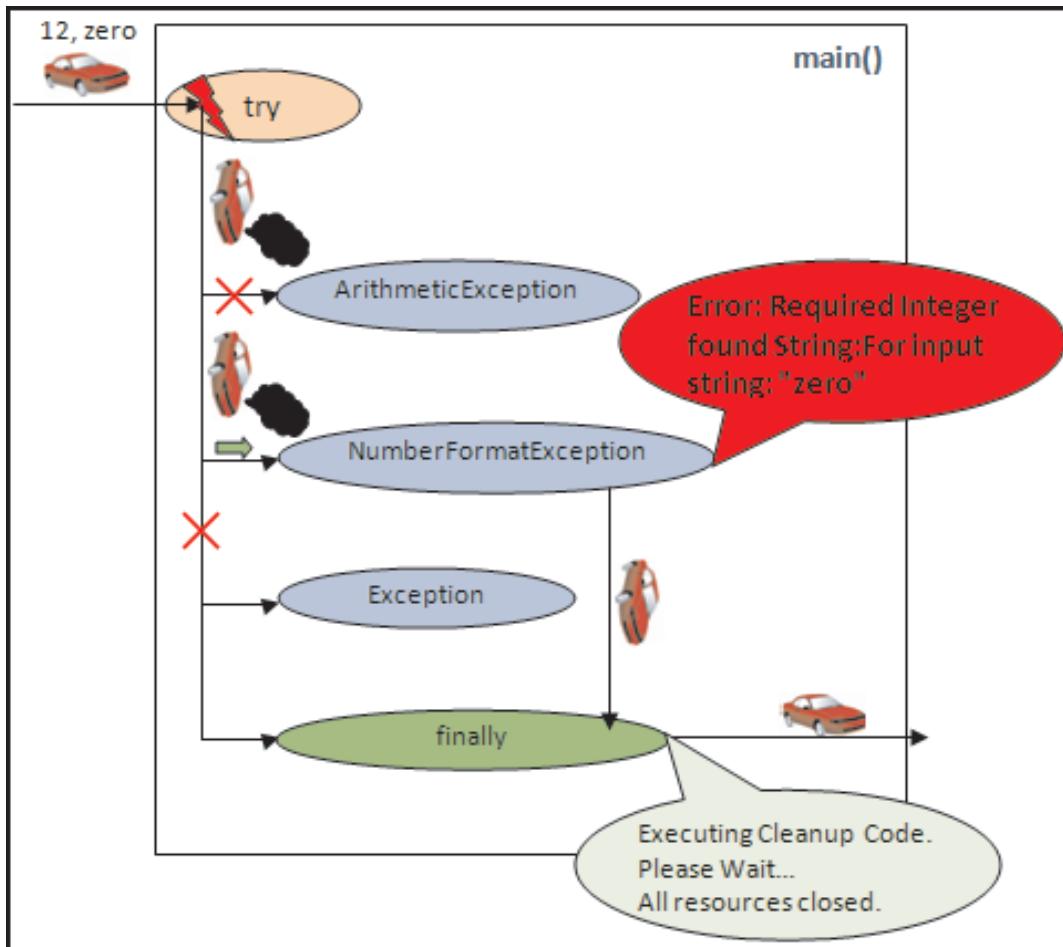


Figure 9.12: Execution of Code Using finally Block

Figure 9.12 shows that after handling the exception in the second catch block, the control is transferred to the finally block. The statements of the finally block get executed and the program execution is completed.

9.4 Guidelines for Handling Exceptions

Guidelines to be followed for handling exceptions are as follows:

- The `try` statement must be followed by at least one `catch` or a `finally` block.
- Use the `throw` statement to throw an exception that a method does not handle by itself along with the `throws` clause in the method declaration.
- The `finally` block must be used to write clean up code.
- The `Exception` subclasses should be used when the caller of the method is expected to handle the exception. The compiler will raise an error message if the caller does not handle the exception.
- Subclasses of `RuntimeException` class can be used to indicate programming errors such as `IllegalArgumentException`, `UnsupportedOperationException`, and so on.
- Avoid using the `java.lang.Exception` or `java.lang.Throwable` class to catch exceptions that cannot be handled. Since, `Error` and `Exception` class can catch all exception of its subclasses including `RuntimeException`, the runtime behavior of such a code often becomes vague when global exception classes are caught. For example, one would not want to catch the `OutOfMemoryError`. How can one possibly handle such an exception?
- Provide appropriate message along with the default message when an exception occurs. All necessary data must be passed to the constructor of the exception class which can be helpful to understand and solve the problem.
- Try to handle the exception as near to the source code as possible. If the caller can perform the corrective action, the condition must be rectified there itself. Propagating the exception further away from the source leads to difficulty in tracing the source of the exception.
- Exceptions should not be used to indicate normal branching conditions that may alter the flow of code invocation. For example, a method that is designed to return a zero, one, or an object can be modified to return `null` instead of raising an exception when it does not return any of the specified values. However, a disconnected database is a critical situation for which no alternative can be provided. In such a case, exception must be raised.
- Repeated re-throwing of the same exception must be avoided as it may slow down programs that are known for frequently raising exceptions.
- Avoid writing an empty `catch` block as it will not inform anything to the user and it gives the impression that the program failed for unknown reasons.

9.5 Check Your Progress

1. _____ are exceptions that are external to the application that it cannot anticipate nor recover from.

(A)	Exception	(C)	Interrupt
(B)	Error	(D)	Fault

2. Which of the following statements about exceptions are true?

a.	An exception can occur due to programming errors, client code errors, or errors that are beyond the control of a program	c.	An appropriate exception handler is one that handles all types of exceptions
b.	The exception object holds information about the type of error and state of the program when the error occurred	d.	If a handler is not found in the method in which the error occurred, the runtime proceeds through the call stack in the same order in which the methods were invoked

(A)	a, d	(C)	b, c
(B)	a, b	(D)	b, d

3. Match following exception types with their corresponding description.

	Exception		Description
a.	ArrayIndexOutOfBoundsException	1.	Occurs on access to a null object member
b.	IllegalArgumentException	2.	Occurs upon an attempt to create instance of an abstract class
c.	NullPointerException	3.	Occurs if an array index is less than zero or greater than the actual size of the array
d.	InstantiationException	4.	Occurs if method receives an illegal argument

(A)	a-2, b-4, c-1, d-3	(C)	a-2, b-3, c-4, d-1
(B)	a-4, b-1, c-3, d-2	(D)	a-3, b-4, c-1, d-2

4. Consider following code:

```
public class Book {
    String bookId;
    String type;
    String author;
    public Book(String bookId, String type, String author) {
        this.bookId = bookId;
        this.type = type;
        this.author = author;
    }
    public void displayDetails() {
        System.out.println("Book Id: "+bookId);
        System.out.println("Book Type: "+type);
        System.out.println("Author: "+author);
    }
    public static void main(String[] args) {
        Book objBook1= new Book(args[1],args[2],args[3]);
        objBook1.displayDetails();
    }
}
```

What will be the output of the code when user passes 'B001', 'Thriller', and 'James-Hadley' as the command line arguments?

(A)	Compilation Error	(C)	java.lang. ArrayIndexOutOfBoundsException
(B)	Book Id: B001 Book Type: Thriller Author: James-Hadley	(D)	Book Id: null Book Type: null Author: null

5. Identify the method of `Exception` class that returns the result of `getMessage()` along with the name of the exception class concatenated to it.

(A)	<code>public Throwable getCause()</code>	(C)	<code>public String getMessage()</code>
(B)	<code>public void printStackTrace()</code>	(D)	<code>public String toString()</code>

6. Consider following code:

```
public class Cart {
    public static void main(String[] args)
    {
        String[] shopCart = new String[4];
        System.out.println(shopCart[1].charAt(1));
    }
}
```

Which type of exception will be raised when the code is executed?

(A)	<code>java.lang.NullPointerException</code>	(C)	<code>java.lang.StringIndexOutOfBoundsException</code>
(B)	<code>java.lang.ArrayIndexOutOfBoundsException</code>	(D)	<code>java.lang.IllegalArgumentException</code>

9.5.1 Answers

1.	B
2.	B
3.	D
4.	C
5.	D
6.	A

```
g package;
import java.util.*;
public class Main {
    public static void main() {
        String pac
        String pr
        String str
        String s
        try {
            m
        } catch (Exception e) {
            e
        }
    }
}
```

Summary

- An exception is an event or an abnormal condition in a program occurring during execution of a program that leads to disruption of the normal flow of the program instructions.
- The process of creating an exception object and passing it to the runtime system is termed as throwing an exception.
- An appropriate exception handler is one that handles the same type of exception as the one thrown by the method.
- Checked exceptions are exceptions that a well-written application must anticipate and provide methods to recover from.
- Errors are exceptions that are external to the application and the application usually cannot anticipate or recover from errors.
- Runtime Exceptions are exceptions that are internal to the application from which the application usually cannot anticipate or recover from.
- Throwable class is the base class of all exception classes and has two direct subclasses namely, Exception and Error.
- The try block is a block of code which might raise an exception and catch block is a block of code used to handle a particular type of exception.
- The user can associate multiple exception handlers with a try block by providing more than one catch blocks directly after the try block.
- Java provides the throw and throws keywords to explicitly raise an exception in the main() method.
- Java provides the finally block to ensure execution of cleanup code even when an exception occurs.

Try It Yourself

1. **Data Informatics Ltd.** is a well-known business outsourcing company located in **New Jersey, USA**. The company hires workforce for data entry projects.

Recently, the company has created its own data entry software. However, the software is not functioning properly. The software does not check the number of values the user is entering nor provides appropriate message to the user if some functionality is not working and simply terminates the application. The company has hired a developer to fix the issue. The developer has created following sample code to handle `ArithmaticException` and `ArrayIndexOutOfBoundsException` exceptions.

```
public class Tester {
    public static void main(String[] args) {
        int sum = 0;
        final int grace = 20;
        try{
            for(int i = 0;i < 5;i++){
                sum = sum + Integer.parseInt(args[i]);
            }
            int perGrace=grace*100/sum;
            System.out.println("Sum is:"+sum);
            System.out.println("Percentage grace is:"+perGrace);
        }
        catch(Exception ex) {
            System.out.println("Error in code");
        }
        catch(ArithmaticException ex) {
            System.out.println("Division by zero");
        }
        catch(ArrayIndexOutOfBoundsException ex) {
            System.out.println("Unreachable array index");
        }
    }
}
```

However, the code is not functioning properly and showing the error 'java.lang.ArithmaticException has already been caught'.

Fix the code to handle the `ArrayIndexOutOfBoundsException` when user passes 3, 4, and 5 as command line arguments and display following message:

Unreachable array index

Onlinevarsity

LEARN @

**YOUR TIME
AND SPACE**



Session - 10

Date and Time API

Welcome to the Session, **Date and Time API**.

This session explains the Date and Time (also called Date-Time) API available in Java since version 8 and the classes introduced in this API. The session also outlines the role of time-zones in Java. Finally, the session describes support for backward compatibility in the API.

In this Session, you will learn to:

- Explain classes of the Date and Time API
- Explain Enum and Clock types
- Describe the role of time-zones
- Explain support for backward compatibility in the new API
- Explain about Stream of Dates



10.1 Introduction

Our day-to-day tasks often require us to work with time constraints. Whether it is catching a train or bus for your commute or performing tasks at home, many everyday activities necessitate time monitoring. This may include going to work or school, delivering orders to customers, and so on. Programmers too require various coding routines to define and track date and time in software applications. Each programming language has a unique set of built-in routines to work with date and time.

A long-standing drawback for Java developers is the lack of strong support in date and time use cases. The existing date-time classes proved to have several issues, which often forced developers to look for other libraries to solve their requirements.

The Date-Time API is intended to address following issues faced by the earlier date and time library:

- **Thread-safe issue** – As `java.util` is not thread-safe, developers had a tough time in dealing with concurrency issues while using date related data. The new Date-Time API is immutable and does not have setter procedures. It provides thread-safety.
- **Poor design** – Default 'date' in earlier versions of Java starts from 1900; 'month' starts from one and 'day' starts from zero, hence, there is no uniformity. The earlier Date-Time API lacks indirect methods that are used for date operations. The new API has many utility methods for such operations.
- **Time-zone handling issue** – Earlier, developers had to write a lot of code to deal with Time-zone issues. The new API has been developed keeping a domain-specific design in mind.

10.2 Classes in the New Date-Time API

All classes of the new Date-Time API are located within the `java.time` package.

Table 10.1 shows a complete list of classes in this API.

Class
Clock
Duration
Instant
LocalDate
LocalDateTime
LocalTime
MonthDay
OffsetDateTime
OffsetTime
Period
Year

Class
YearMonth
ZonedDateTime
Zonelid
ZoneOffset

Table 10.1: Classes in the java.time Package

10.2.1 Clock Class

Clock can be used to get the current instant, date, and time using time-zone. Developers can use Clock in place of System.currentTimeMillis() and TimeZone.getDefault().

Code Snippet 1 displays an example of using Clock. An instance of Clock represents a clock providing access to the current instant, date, and time using a time-zone.

Code Snippet 1:

```
import java.time.*; // import the package for Date-Time API classes
public class ClockDemo {
    public static void main(String[] args) {
        // Creates a new Clock instance based on UTC.
        Clock defaultClock = Clock.systemUTC();
        System.out.println("Clock :" + defaultClock);
        // Creates a Clock instance based on system clock zone
        Clock defaultClock2 = Clock.systemDefaultZone();
        System.out.println("Clock :" + defaultClock2);
    }
}
```

A given date can be verified against the Clock object as shown in Code Snippet 2.

Code Snippet 2:

```
import java.time.*; // import the package for Date-Time API classes
public class ClockDemo {
    private Clock clock;
    public static void main(String[] args) {
        // Creates a new Clock instance based on UTC.
        Clock defaultClock = Clock.systemUTC();
        System.out.println("Clock :" + defaultClock);
```

```
//Creates a clock instance based on system clock zone
Clock defaultClock2=Clock.systemDefaultZone();
System.out.println("Clock:"+defaultClock2);
ClockDemo objClockDemo=new ClockDemo();
LocalDate eventDate=LocalDate.of(2021,02,14);
Clock clock=Clock.systemUTC();
if(eventDate.isBefore(LocalDate.now(clock))) {
    System.out.println("yes");
}
}
```

Duration Calculations

The Duration class comprises a set of methods that can be used to perform calculations based on a Duration object. For example, plusSeconds() method adds seconds in a calculation and minusSeconds() method subtracts seconds in a calculation.

Following are the plus or minus methods:

- plusNanos()
- minusNanos()
- plusMillis()
- minusMillis()
- plusSeconds()
- minusSeconds()
- plusMinutes()
- minusMinutes()
- plusHours()
- minusHours()
- plusDays()
- minusDays()

All the methods work similarly.

Accessing the Time of a Duration

A Duration instance comprises two components:

- Nanoseconds part of duration
- Seconds part of duration

Nanoseconds part here represents the part of Duration that is smaller than a second. Seconds part represents the part of Duration which is larger than one second.

You can use following methods to retrieve these values:

- getNano()
- getSeconds()

Duration can be converted to time units using these conversion methods:

- toNanos()
- toMillis()
- toMinutes()
- toHours()
- toDays()

Each of these methods convert full time interval represented by the Duration to nanoseconds, milliseconds, minutes, hours, or days.

The `getNano()` method returns a portion of the Duration which is less than one second. The `toNanos()` method returns the full time interval converted to nanoseconds.

Code Snippet 3 shows usage of `plusDays()` and `minusDays()` methods.

Code Snippet 3:

```
Duration present=... // assume code is written to get a present duration
Duration samplePlusA=present.plusDays(3);
Duration sampleMinusA=present.minusDays(3);
```

Here, the first line of code produces a Duration variable named `present` that will be used as the base of calculations. It is assumed that code to create the Duration object is added.

Code Snippet 3 then produces two new Duration objects based on the `present` object. The second line generates a Duration, which is equivalent to `present` plus three days. The third line builds Duration that is equivalent to `present` minus three days.

All the calculation methods return new Duration objects representing duration ensuring from the calculation. This is done to keep the Duration object immutable.

10.2.2 Instant Class (`java.time.Instant`)

Instant class is another addition in the new Date-Time API. It denotes a specific moment in time. One Instant is defined as the offset from the origin or the starting point, which is 1/1/1970 - 00:00 - Greenwich Mean Time (GMT). In other words, the Instant class is used for time stamp creation.

This can be used in real-world applications such as shipping, stock updates, and so on.

Generating an Instant

An instance of an Instant can be generated using one of the Instant class factory methods. Code Snippet 4 shows an Instant object which represents the exact moment of now, the method call `Instant.now()`.

Code Snippet 4:

```
Instant sampleNow=Instant.now();
```

Access - Time of an Instant

Following fields contain the time denoted by an Instant object:

- Seconds
- Nanoseconds

The seconds value points to the number of seconds since the origin (1/1/1970 00:00 GMT) and the nanoseconds value points to the part of Instant which is less than one second.

Both seconds and nanoseconds can be accessed via following methods respectively:

- `getEpochSecond()`
- `getNano()`

Instant Calculations

Various Date-Time calculations can be performed on the Instant class with plus or minus methods. For example, the methods `plusSeconds()` and `minusSeconds()` add and subtract seconds in a calculation respectively. Similarly, the calculations can be performed in nanoseconds and milliseconds.

Code Snippet 5 displays the use of Instant.

Code Snippet 5:

```
Instant sampleFuture=sampleNow.plusNanos(4);
//four nanoseconds in the future

Instant samplePast=sampleNow.minusNanos(4);
//four nanoseconds in the past
```

10.2.3 LocalDate Class (`java.time.LocalDate`)

`LocalDate` class in Date-Time API denotes local date that is a date without time-zone information. An example of a local date could be a birthday or official holiday such as Independence Day, which relates to a specific day of the year and not the exact time of that day (the moment the day starts).

`LocalDate` class is bundled with the `java.time` package. `LocalDate` instances are immutable, thus, all the calculations on a `LocalDate` class produce a new `LocalDate`.

Creating a LocalDate

`LocalDate` objects can be created using several approaches. The first approach is to get a `LocalDate` equivalent to the local date of today. Code Snippet 6 shows creating a `LocalDate` object using `now()`.

Code Snippet 6:

```
LocalDate sampleLocDaA=LocalDate.now();
```

Next approach to obtain a `LocalDate` is to create it from a specific year, month, and day information.

Code Snippet 7 shows creating `LocalDate` using `of()`.

Code Snippet 7:

```
LocalDate sampleLocDaB=LocalDate.of(2016, 07, 04);
```

The `LocalDate of()` method generates a `LocalDate` instance, signifying a specific day of a specific month of a specific year, however, without time-zone data.

Accessing the Date Information of a LocalDate

Date information of a `LocalDate` can be accessed using following methods:

- `getYear()`
- `getMonth()`
- `getDayOfMonth()`
- `getDayOfWeek()`
- `getDayOfYear()`

Code Snippet 8 illustrates the use of these methods.

Code Snippet 8:

```
int year=localDate.getYear();
int dayOfMonth=localDate.getDayOfMonth();
Month month=localDate.getMonth();
int dayOfYear=localDate.getDayOfYear();
DayOfWeek dayOfWeek=localDate.getDayOfWeek();
```

```
int monthValue=month.getValue();
```

Notice how `getMonth()` and `getDayOfWeek()` methods return an enum instead of an `int`. These enums can provide their data as `int` values by calling their `getValue()` methods.

LocalDate Calculations

A set of date calculations can be achieved with the `LocalDate` class using one or more of following methods:

- ➔ `plusDays()`
- ➔ `minusDays()`
- ➔ `plusWeeks()`
- ➔ `minusWeeks()`
- ➔ `plusMonths()`
- ➔ `minusMonths()`
- ➔ `plusYears()`
- ➔ `minusYears()`

Code Snippet 9 displays how a `LocalDate` calculation methods works.

Code Snippet 9:

```
LocalDate sampleLocDa=LocalDate.of(2016, 04, 30);
LocalDate sampleLocDaA=sampleLocDa.plusYears(4);
LocalDate sampleLocDaB=sampleLocDa.minusYears(4);
```

In the code, a new instance of `LocalDate` named `sampleLocDa` is created using `of()` method. Then, the code builds a new `LocalDate` instance that represents the date four years later from the earlier specified date. Finally, the code generates a new `LocalDate` instance that denotes the date, four years earlier from the earlier specified date.

10.2.4 `LocalDateTime` Class (`java.time.LocalDateTime`)

`LocalDateTime` class in Date-Time API represents a local date and time without any time-zone data. The `LocalDateTime` can be viewed as a combination of `LocalDate` and `LocalTime` classes of Date-Time API.

`LocalDateTime` is immutable, so all methods that execute calculations on the `LocalDateTime` display a new `LocalDateTime` instance.

Creating a `LocalDateTime`

`LocalDateTime` object can be created by using one of its static factory methods. Here is a statement of code shown in Code Snippet 10 that showcases how to create a `LocalDateTime` object via the `now()` method.

Code Snippet 10:

```
LocalDateTime sampleLocDaTiA=LocalDateTime.now();
```

Here, the variable `sampleLocDaTiA` will be assigned a value representing the current local date and time of the place.

Another approach to create a `LocalDateTime` object is based on a specific year, month, and day as shown in Code Snippet 11.

Code Snippet 11:

```
LocalDateTime sampleLocDaTiB=
LocalDateTime.of(2016, 05, 07, 12, 06, 16, 054);
```

Parameters passed to the `of()` method are year, month, day (of month), hours, minutes, seconds, and nanoseconds respectively.

Access - Time of a LocalDateTime

Date-Time information of a `LocalDateTime` object can be accessed using `getValue()` method. For example, `getDayOfYear()` displays a specific day of the year and `getDayOfWeek()` displays a specific day of the week in a calculation.

Following are the `getvalue()` methods:

- `getYear()`
- `getMonth()`
- `getDayOfMonth()`
- `getDayOfWeek()`
- `getDayOfYear()`
- `getHour()`
- `getMinute()`
- `getSecond()`
- `getNano()`

Some of these methods get the result as an `int` value and some of them display an `enum`. The `int` representation of `enum` can be called using the `getValue()` of `enum`.

Date-Time Calculations

Various date and time calculations can be performed on `LocalDateTime` object with plus or minus methods. For example, `plusYears()` method adds years in a calculation and `minusYears()` method

subtracts years in a calculation. Following are the plus or minus methods:

- ➔ plusYears()
- ➔ plusMonths()
- ➔ plusDays()
- ➔ plusHours()
- ➔ plusMinutes()
- ➔ plusSeconds()
- ➔ plusNanos()
- ➔ minusYears()
- ➔ minusMonths()
- ➔ minusDays()
- ➔ minusHours()
- ➔ minusMinutes()
- ➔ minusSeconds()
- ➔ minusNanos()

Code Snippet 12 illustrates how this calculation methods work.

Code Snippet 12:

```
LocalDateTime sampleLocDaTi=LocalDateTime.now();
LocalDateTime sampleLocDaTiA=sampleLocDaTi.plusYears(4);
LocalDateTime sampleLocDaTiB=sampleLocDaTi.minusYears(4);
```

The code first creates a `LocalDateTime` instance `sampleLocDaTi` signifying the current moment. Then, the code creates a `LocalDateTime` object that denotes a date and time four years later. Finally, the code builds a `LocalDateTime` object that denotes a date and time four years prior.

10.2.5 LocalTime Class

`LocalTime` class in Date-Time API signifies exact time of day without any time-zone data. The `LocalTime` instance can be used to describe real world scenarios such as, the time when the school or work starts in various countries. It helps in analyzing the interest of different zonal people in the Universal Time Coordinated (UTC) time, with concern to the time-zone of respective countries.

The `LocalTime` class is absolute, so all calculations on `LocalTime` objects produce a new `LocalTime` instance.

Creating a LocalTime Object

A `LocalTime` instance can be generated using several approaches. The foremost approach is to create a `LocalTime` instance that denotes the exact time of now. Code Snippet 13 shows the `now()` method.

Code Snippet 13:

```
LocalTime sampleLocTiA=LocalTime.now();
```

Another approach to produce a `LocalTime` object is to create it from specific hours, minutes, seconds, and nanoseconds. Code Snippet 14 displays the `of()` method.

Code Snippet 14:

```
LocalTime sampleLocTiB=LocalTime.of(12, 24, 33, 00135);
```

There are other versions of the `of()` method which only take hours and minutes or hours, minutes, and seconds as factors.

Retrieving the Time of a `LocalTime` Object.

The hours, minutes, seconds, and nanosecond of a `LocalTime` object can be read by using following methods:

- ➔ `getHour()`
- ➔ `getMinute()`
- ➔ `getSecond()`
- ➔ `getNano()`

LocalTime Calculations

`LocalTime` class consists of a set of methods that can perform local time calculations. For example, `plusMinutes()` method adds minutes and `minusMinutes()` subtracts minutes from a given value in a calculation. Plus or minus methods are in `LocalDateTime` object.

Code Snippet 15 shows how these methods work.

Code Snippet 15:

```
LocalTime sampleLocTi=LocalTime.of(12, 24, 33, 00135);
//currentlocaltime

LocalTime sampleLocTiFuture=sampleLocTi.plusHours(4); //future

LocalTime sampleLocTiPast=sampleLocTi.minusHours(4); //past
```

10.2.6 MonthDay Class

`MonthDay` is an immutable Date-Time object that represents month as well as day-of-month. For example, a birthday or banking holiday can be derived from a month and day object.

Code Snippet 16 depicts how `MonthDay` class can be used for checking recurring date-time events, such as a birthday by checking month and day, regardless of the year.

Code Snippet 16:

```
import java.time.*; // import the package for Date-Time API classes
public class DateDemo {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        LocalDate dateOfBirth = LocalDate.of(1988, 02, 13);
        // Code to retrieve the birthday month and day
        MonthDay bday = MonthDay.of(dateOfBirth.getMonth(), dateOfBirth.getDayOfMonth());
        // Code to retrieve the current month and day
        MonthDay currentMonthDay = MonthDay.from(today);
        if (currentMonthDay.equals(bday)) {
            System.out.println("**Colorful Joyful Birthday Buddy**");
        } else {
            System.out.println("Nope, today is not your B'day");
        }
    }
}
```

The code retrieves the birthday month and day, based on a given date of birth. It then retrieves the current month and day based on current date and then, compares the two and displays appropriate messages.

10.2.7 OffsetDateTime Class

`OffsetDateTime` is an immutable illustration of date and time with an offset. This class stores all date and time fields, to an accuracy of nanoseconds, as well as the offset from UTC/Greenwich. For example, the value '23rd November 2016 at 11:34.21.278965143 +05:00' can be stored in an `OffsetDateTime`.

Code Snippet 17 displays an example stating California is GMT or UTC – 07:00 and to get a similar time-zone, static method `ZoneOffset.of()` can be used. After fetching the offset value, `OffsetDateTime` can be shaped by passing a `LocalDateTime` and an offset to it.

Code Snippet 17:

```
LocalDateTime datetime = LocalDateTime.of(2016, Month.FEBRUARY, 15, 18, 20);
// to display the result using Offset
ZoneOffset sampleOffset = ZoneOffset.of("-07:00");
OffsetDateTime date = OffsetDateTime.of(datetime, sampleOffset);
```

```
System.out.println("Sample display of Date and Time using time-zone offset :" + date);
```

10.2.8 OffsetTime Class

`OffsetTime` is an immutable Date-Time object that denotes a time, frequently observed as hour-minute-second-offset. This class stores all time fields, to an exactness of nanoseconds, along with a zone offset. For instance, the value '13:45.30.123456789+02:00' can be secured in an `OffsetTime`.

Using identity-sensitive processes on `OffsetTime` can lead to random results and hence, is not recommended. You should use equals method for comparisons.

Code Snippet 18 shows the complete program to fetch the seconds using the `OffsetTime` class.

Code Snippet 18:

```
import java.time.OffsetTime; // Class to show the result by using
// OffsetTime class method
public class MinuteOffset {
    public static void main(String[] args) {
        OffsetTime d = OffsetTime.now();
        int e = d.getMinute();
        System.out.println("Minutes: " + e);
    }
}
```

Output:

Minutes: 49

Code Snippet 19 shows the complete program to demonstrate `ofInstant()` method with `OffsetTime` class.

Code Snippet 19:

```
import java.time.Instant;
import java.time.OffsetTime;
import java.time.ZoneId;
public class InstCl {
    public static void main(String[] args) {
        OffsetTime d = OffsetTime.ofInstant(Instant.now(), ZoneId.
        systemDefault());
        System.out.println(d);
    }
}
```

Output:

17:11:10.710-07:00

10.2.9 Period Class

`Period` class (`java.time.Period`) represents an amount of time in terms of days, months, and years.

`Duration` and `Period` are somewhat similar; however, the difference between the two can be seen in their approach towards Daylight Savings Time (DST) when they are added to `ZonedDateTime`. `Duration` will add an exact number of seconds, which means that duration of one day is exactly 24 hours. Whereas, a `Period` will add a theoretical day and not actual 24 hours, trying to maintain the local time.

Consider an example to understand this. Add a period of one day and duration of one day to 22:00 at evening before a DST gap. `Period` calculates theoretical day and results in a `ZonedDateTime` at 22:00 the upcoming day. On the other hand, the `Duration` adds exactly 24 hours and results in a `ZonedDateTime` at 23:00 the upcoming day (approximately 60 minutes DST gap).

Code Snippet 20 displays an example to calculate the span of time from today until a birthday, assuming the birthday is on May 22nd.

Code Snippet 20:

```
import java.time.LocalDate; // Class to get the present day
import java.time.Month; // Class to get month related calculations
import java.time.Period; // Class to calculate the time period between two
// time instances
import java.time.temporal.ChronoUnit;
public class NextBday {
    public static void main(String[] args) {
        LocalDate presentday = LocalDate.now();
        LocalDate bday = LocalDate.of(1983, Month.MAY, 22);
        LocalDate comingBDay = bday.withYear(presentday.getYear());
        // To address the belated b'day celebration.
        if (comingBDay.isBefore(presentday) || comingBDay.isEqual(presentday)) {
            comingBDay = comingBDay.plusYears(1);
        }
        Period waitA = Period.between(presentday, comingBDay);
        long waitB = ChronoUnit.DAYS.between(presentday, comingBDay);
        System.out.println("Totally, I must wait for " + waitA.getMonths() + " months,
and " +
                waitA.getDays() + " days to celebrate my next B'day. (" +

```

```
    waitB + " days in total)"; // to display the waiting time for B' day Bash  
}  
}  
}
```

Code Snippet 20 shows the usage of `Period` class. The code also makes use of `ChronoUnit` enumeration.

Output:

Totally, I must wait for 0 months and 22 days to celebrate the next B'day. (16 days in total)

Code Snippet 21 displays another example to depict between() and ofDays() methods of Period class.

Code Snippet 21:

```
import java.time.LocalDate; // Class to get the present day

import java.time.Period; // Class to calculate the time period between two
// time instances

public class JavaPeriodSample {

    public static void main(String[] args) {

        LocalDate h1 = LocalDate.now();

        // To display the time period results

        System.out.println("Time Period between current date and Maximum
no. of days:" + Period.between(h1, LocalDate.MAX).getDays());

        System.out.println("Time Period in Days:" +
Period.ofDays(7).getDays());
    }
}
```

Output:

Time Period between current date and Maximum no. of days:22

Time Period in Days: 7

10 2 10 Year Class

A `Year` (`java.time.Year`) object is an immutable Date-Time object that denotes a year. Every field that is a resultant from a year can be attained. Note that years in International Standard Organization (ISO) chronology only associate with years in the Gregorian system for modern years. Parts of Russia adapted to Gregorian/ISO rules only after 1920. Thus, a cautious approach is required for historical years.

This class does not store or represent a specific month, day, time, or time-zone. Code Snippet 22 displays the calculations using `Year` class.

Code Snippet 22:

```
import java.time.Year; // Class to use Year values in calculations
public class SampleYear {
public static void main(String[] args) {
    System.out.println("The Present Year():" + Year.now());
    System.out.println("The year 2022 is a Leap year :" + Year.isLeap(2022)); // to display whether year 2002 is a leap year or not
    System.out.println("The year 2024 is a Leap year :" + Year.isLeap(2024));
    // to display whether the year 2024 is a leap year or not
}
}
```

Output:

The Present Year (): 2023
 The year 2022 is a Leap year: false
 The year 2024 is a Leap year: true

10.2.11 YearMonth Class

`YearMonth` (`java.time.YearMonth`) is a stable Date-Time object that denotes the combination of year and month. Any field that can be consequent from year and month, such as quarter-of-year, can be acquired. This class does not store or denote a day, time, or time-zone. For example, the value 'November 2011' can be stored in a `YearMonth`.

`YearMonth` can be used to denote things such as credit card expiry, Fixed Deposit maturity date, Stock Futures, Stock options expiry dates, or determining if the year is a leap year or not. Code Snippet 23 displays one such example.

Code Snippet 23:

```
import java.time.YearMonth; // to use the Year and Month info
public class YearMonth {
    public static void main(String[] args) {
        System.out.println("The Present Year Month:" + YearMonth.now());
        // To display present year and month
        System.out.println("Month alone:" + YearMonth.parse("2023-02")
            .getMonthValue()); // To display only the month value
        System.out.println("Year alone:" + YearMonth.parse("2023-02"
            .getYear())); // to display the year value alone
    }
}
```

```

        System.out.println("This year is a Leap year:"+
+YearMonth.parse("2023-02").isLeapYear()); // leap year check
    }
}

```

Output:

The Present Year Month: 2023-11

Month alone: 2

Year alone: 2023

This year is a Leap year: false

10.2.12 ZonedDateTime

`ZonedDateTime` (`java.time.ZonedDateTime`) is an immutable class that represents date and time in addition to a time-zone. This class stores all the date and time fields and can store nanosecond values with time-zone information. For example, the value '15th November 2011 at 21:32.30.34192576 -04:00 in the America/New York time-zone' can be stored in a `ZonedDateTime`. The `ZonedDateTime` class in Date-Time API represents a date and time with time-zone data. This could be the start of specific event somewhere in the world, such as a conference or a rocket launch.

The `ZonedDateTime` class is immutable. This means that all methods executing calculations on a `ZonedDateTime` object yields a new `ZonedDateTime` instance.

You can convert between timelines using offset calculation based on `ZoneId` rules.

Obtaining the offset for a local Date-Time is not as easy as obtaining one for an instant.

Following are three cases of offsets:

- **Normal:** This is applicable for all seasons of the year; normal case concerns a single valid offset for the local Date-Time.
- **Gap:** This is when clock jump forward normally due to the summer DST change from 'spring' to 'autumn'. Gap concerns no legal offset in local Date-Time values.
- **Overlap:** This is when clocks are set back naturally due to the winter DST changes from 'autumn' to 'spring'. Overlap concerns two valid offsets in local Date-Time values.

Creating a ZonedDateTime Object

`ZonedDateTime` object can be created in various ways. The easiest way is to call the `now()` method of `ZonedDateTime` class. Code Snippet 24 illustrates forming a `ZonedDateTime` object using the `now()` method.

Code Snippet 24:

```
ZonedDateTime zoDaTi=ZonedDateTime.now();
```

There is another way to form a `ZonedDateTime` object using `of()` method. This method helps to create a `ZonedDateTime` object from a concrete date and time. Code Snippet 25 illustrates forming a `ZonedDateTime` object using the `of()` method.

Code Snippet 25:

```
ZoneId sampleZoneId=ZoneId.of("UTC+1");
ZonedDateTime zoDaTi2=ZonedDateTime.of(2016,11,30,23,45,59,5682,
sampleZoneId);
```

Accessing Date and Time of a ZonedDateTime

Time-zone based date and time information of a `ZonedDateTime` object can be accessed using `getValue()` method. For example, `getDayOfMonth()` displays a specific day of the month and `getDayOfWeek()` displays a specific day of the week in a calculation.

Code Snippet 26 depicts accessing the year of a `ZonedDateTime`.

Code Snippet 26:

```
int sampleYear=ZonedDateTime.now().getYear();
```

Some of these methods return an `enum` and others return an `int`. An example is shown in Code Snippet 27.

Code Snippet 27:

```
int sampleMonth=ZonedDateTime.now().getMonth().getValue();
```

Date and Time Calculations

The `ZonedDateTime` object contains a set of methods that perform local time calculations. For example, `plusHours()` method adds hours and `minusHour()` subtracts hours from a specified value in a calculation. The `plus()` and `minus()` methods already described in Date-Time class.

Period instance method is shown in Code Snippet 28.

Code Snippet 28:

```
ZonedDateTime newZoneDateTime=previousDateTime.plus(Period.ofDays(4));
```

Here, the variable `newZoneDateTime` represents the value that is four days before from current date and time. This results in a more accurate calculation.

Time-zones

Time-zones are signified by the `ZoneId` class. `ZoneId` object can be created using the `ZoneId.of()` method as shown in Code Snippet 29.

Code Snippet 29:

```
ZoneId sampleZoneId=ZoneId.of("UTC+1");
```

The parameter passed to the `of()` method is the ID of time-zone to create a `ZoneId`. In Code Snippet 29, the ID is 'UTC+1' that is an offset from UTC (Greenwich) time. UTC offset for the desired time-zone can be identified and form an ID matching it by merging 'UTC' with the offset (for example '+1' or '-5').

Another type of time-zone id consisting of a name of the location where the time-zone is active can also be utilized as shown in Code Snippet 30.

Code Snippet 30:

```
ZoneIdsampleZoneIdA=ZoneId.of("America/New_York");
ZoneIdsampleZoneIdB=ZoneId.of("Europe/Paris");
```

Code Snippet 31 displays an example for this class to depict the usage of methods to get year, month, day, hour, minute, seconds, and zone offset.

Code Snippet 31:

```
import java.time.ZonedDateTime; // to access Zoned Date Time
public class ZoneDT { //Class ZoneDT refers to ZonedDateTime
    public static void main(String[] args) {
        System.out.println(ZonedDateTime.now());
        ZonedDateTime sampleZoDT = ZonedDateTime.parse("2023-11-
03T10:15:30+08:00[Asia/Singapore]");
        System.out.println("Present day of the year:"+sampleZoDT.
            getDayOfYear());
        System.out.println("Present year:"+sampleZoDT.getYear());
    }
}
```

Output:

2023-11-06T06:03:51.787+08:00[Etc/UTC]

Present day of the year: 307

Present year: 2023

10.2.13 `ZoneId` Class

A `ZoneId` class is used to recognize rules used to convert between an `Instant` and a `LocalDateTime`.

Two different ID types are as follows:

- **Fixed offsets** - The unchangeable offset since UTC/Greenwich that provides the same offset for every local Date-Times.
- **Geographical regions** - Consist of definite set of rules for finding the offset from UTC/Greenwich pertaining to that zone.

Most static offsets are denoted by `ZoneOffset`. Calling `normalized()` on any `ZoneId` will ensure that a fixed offset ID will be formed as a `ZoneOffset`.

10.2.14 ZoneOffset Class

A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC. This is fixed in most cases.

Time-zone offsets differ from place to place across the planet. The rules for how offsets vary by place and time of year are specified in the `ZoneId` class.

For example, Berlin is two hours ahead of Greenwich/UTC in Spring and four hours ahead during Autumn. The `ZoneId` instance for Berlin will reference two `ZoneOffset` instances - a `+02:00` instance for Spring and a `+04:00` instance for Autumn.

Code Snippet 32 illustrates using this class.

Code Snippet 32:

```
ZoneOffset sampleOffset=ZoneOffset.of("+05:00");
```

10.3 Enums

An enumeration or enum is a type in Java that helps to denote the fixed number of well-known values in Java. This type is defined using the `enum` keyword. For example, it can be used to store number of days in a week or number of planets in the Solar system.

Benefits of using Enums in Java

- Enum is type-safe and cannot be assigned with any other items in addition to the predefined Enum constants to an Enum variable. It is a compiler error to allocate something else to an Enum.
- Enum type has its own namespace.
- The best feature of Enum is that it can be used in Java inside `switch` statements similar to an `int` or `char` primitive data type.
- Adding new constants by extending an Enum in Java is easy and new constants can be added without breaking the existing code.

`ChronoUnit` is one such enumeration. It defines a standard set of date periods units.

Code Snippet 32 demonstrates the usage of this enumeration. It is defined in the `java.time.temporal` package.

Code Snippet 32:

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class EnumDateCalculation {
```

```

public static void main(String args[]) {
    EnumDateCalculation javaenum = new EnumDateCalculation ();
        javaenum.enumChromoUnits ();
    }

    public void enumChromoUnits () {
        // To display the current date
        LocalDate today = LocalDate.now ();
        System.out.println ("Current date: " + today);
        // To display the result 2 weeks addition to the current date
        LocalDate nextWeek = today.plus (2, ChronoUnit.WEEKS);
        System.out.println ("After 2 weeks: " + nextWeek);
        // To display the result 2 months addition to the current date
        LocalDate nextMonth = today.plus (2, ChronoUnit.MONTHS);
        System.out.println ("After 2 months: " + nextMonth);
        // To display the result 2 years addition to the current date
        LocalDate nextYear = today.plus (2, ChronoUnit.YEARS);
        System.out.println ("After 2 years: " + nextYear);
        // To display the result 20 years addition to the current date
        LocalDate nextDecade = today.plus (2, ChronoUnit.DECADES);
        System.out.println ("Date after twenty years: " + nextDecade);
    }
}

```

Output:

Current date: 2023-11-09
 After 2 weeks: 2023-11-23
 After 2 months: 2024-01-09
 After 2 years: 2025-11-09
 Date after twenty years: 2043-11-09

10.4 Temporal Adjusters

`TemporalAdjuster` is a functional interface and a key tool for modifying a temporal object. This is an execution of the strategy design pattern using which the procedure of adjusting a value is externalized. This interface has a method `adjustInto(Temporal)` and it can be accurately called by passing the `Temporal` object. It accepts input as the temporal value and returns the altered value. It can also be raised using with method of the temporal object to be accustomed. The interface is defined in the `java.time.temporal` package.

A TemporalAdjuster can be used to perform complicated date 'math' that is popular in business applications.

For example, it can be used to find 'first Thursday of the month' or 'next Tuesday'.

The TemporalAdjusters class comprises a set of methods for creating the TemporalAdjusters.

Following are some of the methods:

- FirstDayOfMonth()
- FirstDayOfNextMonth()
- FirstInMonth(DayOfWeek)
- LastDayOfMonth()
- Next(DayOfWeek)
- NextOrSame(DayOfWeek)
- Previous(DayOfWeek)
- PreviousOrSame(DayOfWeek)

TemporalAdjusters class

TemporalAdjusters offers many TemporalAdjuster implementations. These can be used to adjust Date-Time objects. Based on a specified date, you can find the first day of that month.

Code Snippet 33 shows the code to find the first day of a month using a specified date.

Code Snippet 33:

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;
public class TemporalAdj {
    public static void main(String args[]) {
        TemporalAdj TemporalAdj = new TemporalAdj();
        TemporalAdj.sampleAdj();
    }
    public void sampleAdj() {
        // To display the current date
        LocalDate sampledateA = LocalDate.now();
        System.out.println("Current date: " + sampledateA);
        // To display the next Wednesday from current date
        LocalDate nextWednesday = sampledateA.with(TemporalAdjusters.
        next(DayOfWeek.WEDNESDAY));
        System.out.println("Next Wednesday on : " + nextWednesday);
        LocalDate firstInYear = LocalDate.of(sampledateA.getYear(),sampledateA.
        getMonth(), 1);
```

```

    LocalDate secondSunday = firstInYear.with(TemporalAdjusters.
        nextOrSame(DayOfWeek.SUNDAY)).with(TemporalAdjusters.next(DayOfWeek.
        SUNDAY));
    System.out.println("Second Sunday of the month on : " + secondSunday);
}
}

```

Output:

Current date: 2023-11-09

Next Wednesday on: 2023-11-15

Second Sunday of the month on: 2023-11-12

Custom TemporalAdjuster

Related to the `TemporalAdjusters` class provided in the API, it can produce the custom implementation of the `TemporalAdjuster` having the business logic and use it in the Java application. It should be useful when there are recapping patterns of adjustment in the use cases. Those patterns can be externalized using a `TemporalAdjuster` application.

A custom `TemporalAdjuster` implementation is shown in Code Snippet 34, where a date is assumed to adjust and the next odd date is returned.

Code Snippet 34:

```

import java.time.LocalDate; // To use local date in java
import java.time.Month; // To include month
import java.time.temporal.Temporal; // To initiate temporal
import java.time.temporal.TemporalAdjuster;
public class CustomTempAdjSample implements TemporalAdjuster {
    public Temporal adjustInto(Temporal temporalInput) {
        LocalDate loDate = LocalDate.from(temporalInput);
        int day = loDate.getDayOfMonth(); // Present day
        if (day % 2 == 0) { // to find out odd days
            loDate = loDate.plusDays(1);
        } else {
            loDate = loDate.plusDays(2);
        }
        return temporalInput.with(loDate);
    }
    public static void main(String args[]) {
        LocalDate randomDateA = LocalDate.of(2023, Month.MAY, 5);
        LocalDate randomDateB = LocalDate.of(2023, Month.MAY, 7);
        CustomTempAdjSample nextOddDay = new CustomTempAdjSample();
    }
}

```

```

LocalDate upcomOddDayA = randomDateA.with(nextOddDay);
LocalDate upcomOddDayB = randomDateB.with(nextOddDay);
System.out.println("Upcoming Odd Day for " + randomDateA + " is "
+ upcomOddDayA); // to display the upcoming odd day
System.out.println("Upcoming Odd Day for " + randomDateB + " is "
+ upcomOddDayB);
} // final result
}

```

Output:

Upcoming Odd Day for 2023-05-05 is 2021-05-07

Upcoming Odd Day for 2023-05-07 is 2021-05-09

10.5 Backward Compatibility with Older Versions

The original `Date` and `Calendar` objects contains the `toInstant()` method to convert them to the new Java Date-Time API. It can then use an `ofInstant(Instant, ZoneId)` method to return a `LocalDateTime` or `ZonedDateTime` object as shown in Code Snippet 35.

Code Snippet 35:

```

...
Date sampleDate = new Date();
Instant sampleNow = sampleDate.toInstant();
LocalDateTime dateTime = LocalDateTime.ofInstant(sampleNow, myZone);
ZonedDateTime zdt = ZonedDateTime.ofInstant(sampleNow, myZone);

```

In the given code, `toInstant()` method is additional to the original `Date` and `Calendar` objects, which can be used to convert them to new Date-Time API. Here, `ofInstant(Instant, ZoneId)` method is used to get a `LocalDateTime` or `ZonedDateTime` object as shown in Code Snippet 36.

Code Snippet 36:

```

import java.time.LocalDateTime; // to initiate local date and time
import java.time.ZonedDateTime; // to initiate zoned time
import java.util.Date;
import java.time.Instant;
import java.time.ZoneId;
public class BWCompatibility {
    public static void main(String args[]) {
        BWCompatibility bwcompatibility = new BWCompatibility();
    }
}

```

```

        bwcompatibility.sampleBW();

    }

public void sampleBW() {
    // To display the current date
    Date sampleCurDay = new Date();
    System.out.println(" Desired Current date= " + sampleCurDay);
    // to display result
    // To display the instant of current date
    Instant samplenow = sampleCurDay.toInstant();
    ZoneId samplecurZone = ZoneId.systemDefault();
    // To display the current local date
    LocalDateTime sampleLoDaTi = LocalDateTime.ofInstant(samplenow,
    samplecurZone);
    System.out.println(" Desired Current Local date= " + sampleLoDaTi);
    // To display result
    // To display the desired current zoned date
    ZonedDateTime sampleZoDaTi = ZonedDateTime.ofInstant(samplenow,
    samplecurZone);
    System.out.println(" Desired Current Zoned date= " + sampleZoDaTi);
    // To display result
}
}

```

Output:

Desired Current date= Thu Nov 09 12:04:34 EDT 2023

Desired Current Local date= 2023-11-09T12:04:34.208

Desired Current Zoned date= 2023-11-09T12:04:34.208+05:30 [America/New_York]

10.6 Parsing and Formatting Dates

Parsing dates from strings and formatting dates to strings are possible with the `java.text.SimpleDateFormat` class.

Code Snippet 37 displays two examples of how the `SimpleDateFormat` class works on `java.util.Date` instances.

Code Snippet 37:

```

SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
String dateString = format.format( new Date() );
Date sampleDate = format.parse ("2011-03-25");

```

10.7 TimeZone (`java.util.TimeZone`)

`TimeZone` class in Java represents the time-zones. This class is used in time-zone bound calculations.

Retrieving a Time-Zone from a Calendar

Code Snippet 38 displays a simple example of how to get the time-zone from a Calendar.

Code Snippet 38:

```
Calendar cal = new GregorianCalendar();
TimeZone tizo = cal.getTimeZone();
```

Code Snippet 39 displays how to set time-zone.

Code Snippet 39:

```
cal.setTimeZone(tizo);
```

Creating a `TimeZone` Instance

There are two ways to obtain a `TimeZone` instance as shown in Code Snippet 40.

Code Snippet 40:

```
TimeZone tizo = TimeZone.getDefault();
OR
TimeZone tizo = TimeZone.getTimeZone("Europe/Paris");
```

The first method (`TimeZone.getDefault()`) displays the default time-zone for the system (personal computer or server) this program is streaming on.

The second method (`TimeZone.getTimeZone ("Europe/Paris")`) returns the `TimeZone` instance corresponding to the given `timezoneID` (which in this case is Europe/Paris).

Time-zone Names, IDs, and Offsets

Code Snippet 41 shows how to retrieve display name, ID, and time offset of a given time-zone.

Code Snippet 41:

```
timeZone.getDisplayName();
timeZone.getID();
timeZone.getOffset( System.currentTimeMillis() );
```

Code Snippet 42 denotes a sample of time-zone.

Code Snippet 42:

```
import java.time.ZonedDateTime;
import java.time.ZoneId;
public class Java8CurTZone {
    public static void main(String args[]) {
        Java8CurTZone java8curtzone = new Java8CurTZone();
```

```

java8curtzone.sampleZDTime();
}

public void sampleZDTime() {
    // To display the current date and time
    ZonedDateTime dateA = ZonedDateTime.parse("2023-02-16T10:15:30+08:00[Asia/
    Singapore]");
    System.out.println("dateA: " + dateA);
    // To display the zoneId
    ZoneId sampleIdA = ZoneId.of("Asia/Singapore");
    System.out.println("ZoneId: " + sampleIdA);
    // To display the current Zone
    ZoneId sampleCurrentZoneA = ZoneId.systemDefault();
    System.out.println("CurrentZone: " + sampleCurrentZoneA);
}
}

```

Output:

dateA: 2023-02-16T10:15:30+08:00[Asia/Singapore]

ZoneId: Asia/Singapore

CurrentZone: Etc/UTC

10.8 Stream of Dates

Java 9 introduced a new method `LocalDate.datesUntil()` which returns an ordered sequential stream of dates. The returned stream begins from the specified date (inclusive) up to the end (exclusive) by an incremental step of one day. Using `datesUntil()` makes it easy to create dates streams with fixed offset.

Code Snippet 43 shows the code to print all the dates between today and 2023-11-30.

Code Snippet 43:

```

package session10;
import java.time.LocalDate;
import java.time.Period;
import java.time.Month;
import java.util.stream.Stream;
public class DatesUntilMethodDemo {
    public static void main(String args[]) {

```

```
// Print the days between today and 2023-11-30
Stream<LocalDate> dates = LocalDate.now().datesUntil(LocalDate.
parse("2023-11-30"));
dates.forEach(System.out::println);
}
}
```

The code prints all the dates between current date and 2023-11-30. Current date is retrieved using `LocalDate.now()` method and the dates between the two are retrieved using `datesUntil()` method.

The output of the code will be:

2023-11-08
2023-11-09
2023-11-10
2023-11-11
2023-11-12
2023-11-13
2023-11-14
2023-11-15
2023-11-16
2023-11-17
2023-11-18
2023-11-19
2023-11-20
2023-11-21
2023-11-22
2023-11-23
2023-11-24
2023-11-25
2023-11-26
2023-11-27
2023-11-28
2023-11-29

10.9 Check Your Progress

1. Which of the following APIs represents a specialized Date-Time API to deal with various time-zones?

(A)	Local	(C)	International
(B)	Zoned	(D)	Continental

2. Which of following options does the Instant class represent?

(A)	Day time	(C)	Approximate time
(B)	Specific time	(D)	Fixed time

3. Which of these two methods can be used to perform Duration calculations?

(A)	plusNanos()	(C)	toMillis()
(B)	toNanos()	(D)	minusMillis()

4. Which of the following are three types of offsets?

(A)	Simple, Odd, and Even	(C)	plusDays(), plusNanos(), and plusMillis()
(B)	Normal, Gap, and Overlap	(D)	toNanos(), toMillis(), and toSeconds()

5. Which among the following methods comes under the class `java.time.temporal.TemporalAdjusters`?

(A)	plusDays()	(C)	getDayOfMonth()
(B)	ofInstant()	(D)	toNanos()

10.9.1 Answers

1.	B
2.	B
3.	A and D
4.	A
5.	B

Summary

- The new Date-Time API introduced from Java 8 onwards is a solution for many unaddressed drawbacks of the previous API.
- Date-Time API contains many classes to reduce coding complexity and provides various additional features to work date and time.
- Enum in Java is a keyword, a feature that is used to denote the fixed number of well-known values in Java.
- TemporalAdjuster is a functional interface and a key tool for altering a temporal object.
- Java TimeZone class is a class that denotes time-zones and is helpful when doing calendar arithmetic across time-zones.
- A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC. This is fixed in most cases.
- TemporalAdjuster is a functional interface and a key tool for modifying a temporal object.

Try It Yourself

Victoria runs a cosmetic business which has become very popular. She maintains a business register that consists of the complete record of her customers such as customer name, address, customer age, product, purchase date, and so on. She has now decided to automate the tasks so that she can concentrate more on growing the business.

As a software developer, you have to perform following tasks to implement customer profiling:

1. Create a class named `DemoDateTime.java` and set the variables `cname`, `dob`, and `address` with appropriate data type.
2. Use the `of()` method to set the date of birth of one of the customers as 9th December, 1960.
3. Calculate the age of the customer using the appropriate class of Date and Time API and the `now()` method.
4. Use the `now()` method to get the difference of current date and purchase taken as 12th April, 2020.

Session - 11

Additional Features of Java

Welcome to the Session, **Additional Features of Java**.

This session explains some of the sophisticated aspects of Java programming. These features are designed to help in writing efficient, maintainable, and robust code and they are often employed in real-world applications. This session explains the concept of generics, lambda expressions, annotations, streams, modules, records, switch expressions, text blocks, and pattern matching by using instanceof.

In this Session, you will learn to:

- Explain Lambda Expressions
- Explain the use of Generics
- Elaborate the use of annotations
- Describe streams
- Define modules
- Explain switch expressions
- Describe text blocks
- Outline the use of pattern matching for instanceof



11.1 Overview of Additional Features of Java

Java is a popular and widely used programming language that has been evolving and adding new features since its inception in the year 1995. Java has introduced many features that make it more powerful, expressive, and convenient for developers. Some of the additional features of Java are as follows:

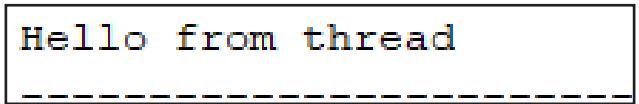
- **Lambda expressions:** Lambda expressions are a way of writing anonymous functions in Java. They allow us to pass a block of code as an argument to another method, without having to define a separate method or class. Lambda expressions can make the code more concise and readable, especially when using functional interfaces or streams. Lambda expressions were first introduced in Java 8. An example of using lambda expressions to create and run a thread is shown in Code Snippet 1.

Code Snippet 1:

```
@FunctionalInterface
interface Sayable {
    public String say(String name);
}

public class LambdaExample2 {
    public static void main(String[] args) {
        // A lambda expression that implements the Sayable interface
        Sayable s = (name) -> {
            return "Hello, " + name;
        };
        System.out.println(s.say("Alex")); // Invoking the say() method
    }
}
```

The output of Code Snippet 1 is as shown in Figure 11.1.



Hello from thread

Figure 11.1: Output of Using Lambda Expression

In Code Snippet 1, the lambda expression `(name) -> { return "Hello, " + name; }` provides the implementation of the `say(String name)` method of the `Sayable` interface. The parameter list has one parameter `name` and the lambda body is a single expression that returns a string. The parameter type can be omitted, as it can be inferred by the compiler.

The lambda expression can access the local variable `width`, which is effectively `final` (meaning it does not change its value after initialization).

- **Generics:** Generics are a way of writing generic classes and methods in Java. They allow to parameterize different types of the classes and methods, so that they can work with different types of objects, without having to write duplicate code or cast the objects. Generics can make the code more type-safe and reusable, especially when using collections or algorithms. Generics were first introduced in Java 5. An example of using generics to create and use a generic stack class is shown in Code Snippet 2.

Code Snippet 2:

```
// Generic stack class
public class Stack<E> {
    private E[] elements;
    private int size;
    public Stack(int capacity) {
        elements = (E[]) new Object[capacity];
    }
    public void push(E element) {
        elements[size++] = element;
    }
    public E pop() {
        return elements[--size];
    }
    public boolean isEmpty() {
        return size == 0;
    }
}
// Using the generic stack class
public class StackDemo {
    public static void main(String[] args) {
        // Create a stack of strings
        Stack<String> stack = new Stack<>(10);
        // Push some strings into the stack
        stack.push("Java");
        stack.push("Python");
```

```

stack.push ("C++") ;

// Pop and print the strings from the stack

while (!stack.isEmpty ()) {
    System.out.println (stack.pop ()) ;
}
}
}

```

A stack adheres to the Last-In-First-Out (LIFO) principle, where the last added element is the first to be removed. It can store various objects, including numbers, strings, and other data structures. A generic class operates with diverse object types without requiring explicit type declaration. It utilizes a placeholder symbol, such as <E>, to denote the handled object types.

Code Snippet 2 uses a data structure to hold stack elements, and an integer to maintain the count of elements in the stack. A constructor `public Stack(int capacity)` accepting an integer parameter creates an array of objects with the specified capacity, casting it to the generic type E. The method `public void push(E element)` takes an object of type E as a parameter. It is added to the stack's top and assigned to the next available position in the array. Simultaneously, it increases the size of the stack by one. There is also a method `public E pop()` that retrieves and removes the top element of the stack. Also, it decrements the size by one and returns the element at that array position. Finally, a method checks if the stack is empty by comparing the size with zero, returning true if it is and false if it is not. The code also defines a demo class that uses the generic stack class to create and manipulates a stack of strings.

The Generic class `Stack` should be saved as a .java file and compiled before executing (for example, save and compile as `StackDemo.java`). The output of Code Snippet 2 is as shown in Figure 11.2.

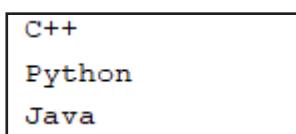


Figure 11.2: Output Using Generic Class

The output shows that the last string that was pushed into the stack was C++ and was the first one that was popped out, followed by Python and then, Java.

- **Annotations:** Annotations are a way of adding metadata to Java code. They can provide information to the compiler, the runtime, or other tools, such as frameworks or libraries. Annotations can help improve the readability, maintainability, and functionality of Java code. Annotations were first introduced in Java 5.

An example of using annotations to mark a method as deprecated and override a method from a superclass is shown in Code Snippet 3.

Code Snippet 3:

```
public class AnnotationDemo {  
    // Annotate a method as deprecated  
    @Deprecated  
    public void oldMethod() {  
        System.out.println("This method is deprecated");  
    }  
    // Annotate a method as overriding a method from a //superclass  
    @Override  
    public String toString() {  
        return "This is an annotation demo";  
    }  
}
```

Code Snippet 3 defines a public class named `AnnotationDemo`. The class has two methods namely, `oldMethod()` and `toString()`. Both methods are annotated with built-in Java annotations. The `@Deprecated` annotation marks the `oldMethod()` as deprecated, which means that it should not be used anymore and may be removed in the future versions of the program. The `@Override` annotation indicates that the `toString()` method overrides the method with the same name and signature from the superclass `Object`. The `toString()` method returns a string representation of the `AnnotationDemo` object, which is “This is an annotation demo”.

Annotations can be used for various purposes, such as:

- **Documenting the code:** Annotations can be used to generate documentation from the source code, such as Javadoc comments. For example, the `@param` annotation can be used to describe the parameters of a method.
- **Validating the code:** Annotations can be used to check the correctness of the code at compile time or runtime, such as syntax, semantics, or constraints. For example, the `@Override` annotation can be used to ensure that a method is overriding a method from a superclass.
- **Enhancing the code:** Annotations can be used to modify or extend the behavior of the code at runtime, such as adding new features, functionalities, or aspects.
- For example, the `@Test` annotation can be used to mark a method as a test case for a testing framework.

Annotations in Java have a specific syntax and structure. They start with an @ symbol, followed by the name of the annotation. They can also have optional parameters, which are specified as name-value pairs within parentheses. For example, consider Code Snippet 4.

Code Snippet 4:

```
// An annotation with no parameters
@Override

// An annotation with one parameter
@SuppressWarnings("unchecked")

// An annotation with multiple parameters
@Test(timeout = 1000, expected = ArithmeticException.class)
```

Annotations can be applied to different elements of Java code, such as classes, methods, fields, parameters, or packages. They can also be applied to other annotations, to create meta-annotations. The placement and scope of annotations are determined by their target and retention policies, which are specified using the @Target and @Retention meta-annotations.

There are three types of annotations in Java:

Built-in Annotations

- These are predefined annotations that are provided by the Java language or the Java platform. They have a special meaning and effect for the compiler or the runtime. For example, they are used with @Deprecated, @FunctionalInterface, or @SafeVarargs keywords.

Standard Annotations

- These are common annotations that are defined by the Java API or third-party libraries. They provide additional information or functionality for various tools or frameworks. For example, using keywords such as @Test, @Entity, or @Autowired.

Custom Annotations

- These are user-defined annotations that are created by using the @interface keyword. They can have their own attributes and logic, and can be processed by custom tools or frameworks.

An example of custom annotation is as given in Code Snippet 5.

Code Snippet 5:

```
// A custom annotation to mark a method as a benchmark
@interface Benchmark {
    // An attribute to specify the number of iterations
    int value() default 1;
}

// A class that uses the custom annotation
class MyClass {
    // A method that is annotated as a benchmark
    @Benchmark(10)
    public void myMethod() {
        // Some logic
    }
}
```

In Code Snippet 5, the `@interface` keyword is used to declare a custom annotation. The `Benchmark` annotation has one attribute named `value`, which is an `int` that specifies the number of iterations for the `Benchmark`. The default value is 1.

The `MyClass` class uses the `Benchmark` annotation on its `myMethod` method. This means that the method is marked as a benchmark and should be executed 10 times.

11.2 Creating Custom Annotations

To create a custom annotation in Java, you must follow these steps:

Declare the annotation using the `@interface` keyword, followed by the name of the annotation.

Refer to Code Snippet 6.

Code Snippet 6:

```
public @interface MyAnnotation {
    // Annotation body
}
```

Optionally, add meta-annotations to specify the scope and the target of your custom annotation. Meta-annotations are annotations that can be applied to other annotations, such as @Retention, @Target, or @Documented.

Refer to Code Snippet 7.

Code Snippet 7:

```
@Retention(RetentionPolicy.RUNTIME) // Specify the retention
//policy of the annotation
@Target(ElementType.METHOD)
// Specify the element type that can be annotated
public @interface MyAnnotation {
    // Annotation body
}
```

Optionally, add elements to your custom annotation. Elements are methods that can have parameters, return types, and default values. They define the attributes of your custom annotation that can be assigned values when using the annotation.

Consider Code Snippet 8:

Code Snippet 8:

```
package com.example;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) // Specify the retention
//policy of the annotation
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    // An element with a String parameter and a default value
    String value() default "Hello";
    // An element with an int parameter and no default value
    int number();
}
```

Code Snippet 8 defines a custom annotation named `MyAnnotation` that can be applied to methods at runtime. The annotation has two elements `value` and `number`. The `value` element has a default value of "Hello", while the `number` element has no default value and must be specified when using the annotation.

Optionally, add logic to your custom annotation. Logic is code that can process or manipulate the values of the elements or perform some actions based on the annotation.

Logic can be written in a separate class or method that uses reflection or annotation processing to access and handle the annotation.

Refer Code Snippet 9.

Code Snippet 9:

```
package com.example;

import java.lang.reflect.Method;

public class MyAnnotationProcessor {
    public static void process(Object obj) {
        // Get the class of the object
        Class<?> objClass = obj.getClass();
        // Get all the methods of the class
        Method[] methods = objClass.getDeclaredMethods();
        // Loop through the methods
        for (Method method : methods) {
            // Check if the method has MyAnnotation
            if (method.isAnnotationPresent(MyAnnotation.class)) {
                // Get the annotation instance
                MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
                // Get the values of the elements
                String value = annotation.value();
                int number = annotation.number();
                // Do something with the values
                System.out.println("Method: " + method.getName());
                System.out.println("Value: " + value);
                System.out.println("Number: " + number);
            }
        }
    }
}
```

Code Snippet 9 defines a class called `MyAnnotationProcessor` that implements the `Processor` interface, which is the main interface for an annotation processor.

The class has a static method called `process` that takes an `object` as a parameter and performs some actions on it. The method first gets the class of the object using the `getClass` method, which returns a `Class` object that represents the runtime type of the object.

Then, the method gets all the methods of the class using the `getDeclaredMethods` method, which returns an array of `Method` objects. It represents all the declared methods of the class, including private ones.

Next, the method loops through the array of methods using a `for-each` loop and checks if each method has an annotation called `MyAnnotation` using the `isAnnotationPresent` method, which returns a boolean value indicating whether the method has the specified annotation or not.

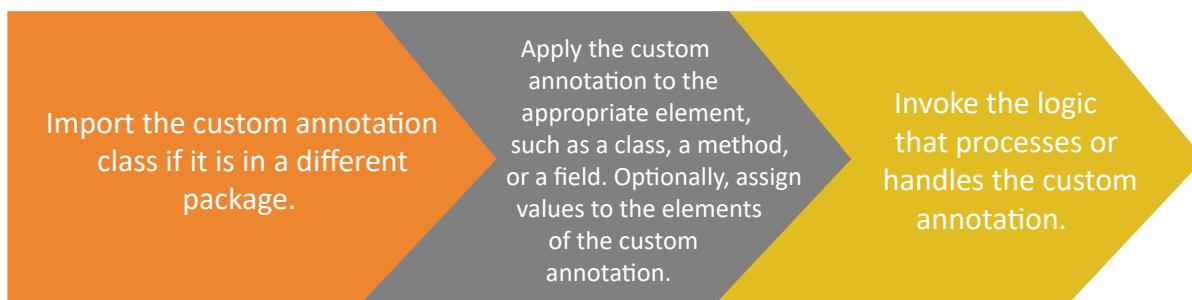
If the method has the annotation, the method gets the annotation instance using the `getAnnotation` method, which returns an object of type `MyAnnotation` that represents the annotation on the method.

Then, the method gets the values of the elements of the annotation using the `value` and `number` methods, which return a `String` and an `int` value respectively. These are defined in the `MyAnnotation` interface defined in Code Snippet 8.

Finally, the method prints out the `name`, `value`, and `number` of each annotated method using the `System.out.println` method, which writes a message to the standard output stream.

Save the code in Code Snippet 8 as `MyAnnotation.java`. Save the code in Code Snippet 9 as `MyAnnotationProcessor.java`.

After a custom annotation has been created in Java, in order to use it, you must follow these steps:



Consider Code Snippet 10.

Code Snippet 10:

```

package com.example;
import com.example.MyAnnotation;
// Import the custom annotation class
public class MyClass {
    @MyAnnotation(value = "World", number = 42) // Apply and assign
    // values to the custom annotation
    public void myMethod() {
        // Some logic
    }
}
  
```

Save the code in Code Snippet 10 as MyClass.java.

An example of invoking logic for custom annotation is shown in Code Snippet 11.

Code Snippet 11:

```
package com.example;

public class Main {
    public static void main(String[] args) {
        // Create an instance of MyClass
        MyClass myClass = new MyClass();
        // Invoke the process method from //MyAnnotationProcessor
        MyAnnotationProcessor.process(myClass);
    }
}
```

Save Code Snippet 11 as Main.java.

Now, compile all the classes (Code Snippets 8-11) and then, run Code Snippet 11.

The output of creating and using custom Annotation is shown in Figure 11.3.

Method: myMethod
Value: World
Number: 42

Figure 11.3: Output of Creating and Using Custom Annotation

11.3 Processing and Using Custom Annotations

To process and use annotations at runtime or compile time in Java, you require to use different APIs and tools depending on your use case. Here are some general steps and guidelines to follow:

- Runtime annotations can be processed by using Reflection API, which allows you to access and manipulate the annotations of classes, methods, fields, parameters, or packages.
Methods such as `getAnnotations()`, `getAnnotation(Class<T>)`, or `isAnnotationPresent(Class<T>)` are used to retrieve the annotations of a given element. Methods such as `invoke(Object, Object...)`, `set(Object, Object)`, or `get(Object)` are also used to perform actions on the annotated elements.

For example, to print the value of a custom annotation named `@MyAnnotation` applied to a method named `myMethod()` is as shown in Code Snippet 12.

Code Snippet 12:

```
// Get the class object of MyClass
Class<?> objClass = MyClass.class;
// Get the method object of myMethod
Method method = objClass.getDeclaredMethod("myMethod");
// Check if the method has MyAnnotation
if (method.isAnnotationPresent(MyAnnotation.class)) {
    // Get the annotation instance
    MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
    // Get the value of the annotation element
    String value = annotation.value();
    // Print the value
    System.out.println("Value: " + value);
}
```

- To process annotations at compile time, the Annotation Processing API is used, which allows to create and register custom annotation processors that can generate code, validate, or modify the annotated elements.

An annotation processor can be created by extending the `javax.annotation.processing.AbstractProcessor` class and overriding methods such as `process(Set<? extends TypeElement>, RoundEnvironment)`, `getSupportedAnnotationTypes()`, or `getSupportedSourceVersion()`. Classes such as `javax.annotation.processing.Filer`, `javax.annotation.processing.Messager`, or `javax.lang.model.util.Elements` are also used to create files, print messages, or access elements. For example, you can use following code as shown in Code Snippet 13. This will create an annotation processor that generates a file named `Hello.txt` with the content "Hello World" for each class annotated with `@MyAnnotation`.

Code Snippet 13:

```
// Import the relevant classes
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Filer;
import javax.annotation.processing.Processor;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
```

```
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.TypeElement;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Set; import
// Annotate the processor class with supported annotation types and
//source version
@SupportedAnnotationTypes ("com.example.MyAnnotation")
@SupportedSourceVersion (SourceVersion.RELEASE_8)
public class MyAnnotationProcessor extends AbstractProcessor {
// Override the process method
@Override
    public boolean process (Set<? extends TypeElement> annotations,
RoundEnvironment roundEnv) {
        // Get the filer instance from the processing environment
        Filer filer = processingEnv.getFiler ();
        // Loop through the annotations
        for (TypeElement annotation : annotations) {
            // Get the annotated elements of type class
            Set<? extends Element> annotatedElements = roundEnv.
getElementsAnnotatedWith(annotation);
            // Loop through the annotated elements
            for (Element element : annotatedElements) {
                // Get the name of the element
                String name = element.getSimpleName ().toString ();
                // Create a file named Hello.txt in the same
//package as the element
                try (PrintWriter writer = new PrintWriter (filer.
createResource (StandardLocation.SOURCE_OUTPUT,
```

```

element.getEnclosingElement().toString(),"Hello.txt").openWriter()))
{
    // Write Hello World to the file
    writer.println("Hello World");
} catch (IOException e) {
    e.printStackTrace();
}
return true;
}
}

```

This code will run successfully when MyClass is also suitably modified.

- To use annotations at runtime or compile time, it is essential to confirm that the appropriate retention policy and target policy are in place. The retention policy determines how long the annotations are retained by Java, and can be one of SOURCE, CLASS, or RUNTIME. The target policy determines which elements can be annotated by specified annotation, and can be one of TYPE, FIELD, METHOD, PARAMETER, and so on. These policies can also be specified using meta-annotations such as @Retention and @Target on the annotation definition. For example, to define a custom annotation named @MyAnnotation that has a runtime retention policy and can be applied to methods is as shown in Code Snippet 14.

Code Snippet 14:

```

// Import the relevant classes
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
// Annotate the annotation definition with retention and target policies
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    // Define an element with a String parameter and a default value
    String value() default "Hello";
}

```

- **Streams:** Streams are a way of processing collections of data in a declarative and functional way. They allow us to perform operations such as filtering, mapping, sorting, or reducing on the elements of a collection, without having to write loops or mutate the original collection. Streams can make the code more concise and parallelizable, especially when using lambda expressions or method references. Streams were first introduced in Java 8. An example of using streams to filter and print the even numbers from a list of integers is as shown in Code Snippet 15.

Code Snippet 15:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
public class StreamDemo {
    public static void main(String[] args) {
        // Create a list of integers
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
        // Create a stream from the list
        Stream<Integer> stream = list.stream();
        // Filter the stream for even numbers and print them
        stream.filter(n -> n % 2 == 0).forEach(System.out::println);
    }
}
```

Code Snippet 15 implements the Java Stream API to process a collection of objects in a functional way. It creates a stream of integers from a list, filters out the odd numbers, and prints the remaining even numbers to the console. The output of Code Snippet 15 is as shown in Figure 11.4.

A small rectangular box containing three even integers: 2, 4, and 6, each on a new line.

Figure 11.4: Output of Streams

To use streams with collections or arrays, you require to use the `stream()` method of the `Collection` interface or the `Arrays` class. This method returns a `Stream` object that represents a sequence of elements from the collection or array. Once a stream is acquired, various methods from the `Stream` interface or its subinterfaces (`IntStream`, `LongStream`, and `DoubleStream`) can be utilized to execute operations on the stream. These methods can be classified into two categories:

1. **Intermediate:** Intermediate methods return another stream as the result and can be chained together to form a pipeline of operations.

2. **Terminal:** Terminal methods return a non-stream value as the result, such as a collection, an array, a primitive value, or an optional value. Terminal methods also consume the stream, meaning that you cannot use the stream after calling a terminal method.
Some examples of intermediate methods include:

```
filter (Predicate<T>)
```

Returns a stream consisting of the elements that match the given predicate.

```
map (Function<T, R>)
```

Returns a stream consisting of the elements that match the given predicate.

```
filter (Predicate<T>)
```

Returns a stream consisting of the elements sorted according to the given comparator.

Some examples of terminal methods include:

```
collect (Collector<T, A, R>)
```

Returns a collection or another value that is the result of applying the given collector to the elements of this stream.

```
toArray ()
```

Returns an array containing the elements of this stream.

`sum()`

Returns the sum of the elements in this stream.

Some other methods of the `Stream` interface include:

`flatMap (Function<T, Stream<R>>)`

- Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. This is a way of flattening nested streams into a single stream.

`limit (long maxSize)`

- Returns a stream consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.

`skip (long n)`

- Returns a stream consisting of the remaining elements of this stream after discarding the first `n` elements of the stream.

`peek (Consumer<T>)`

- Returns a stream consisting of the remaining elements of this stream after discarding the first `n` elements of the stream.

`reduce (BinaryOperator<T>)`

- Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. This is mainly useful for debugging purposes.

- **Modules:** Modules are a way of organizing and encapsulating Java code. They allow us to define and enforce the dependencies and visibility of the packages and classes within a module, as well as between different modules. Modules can help improve the modularity, security, and performance of Java applications. Modules were introduced in Java 9.

To begin with, create a new Modular Project as shown in Figure 11.5.

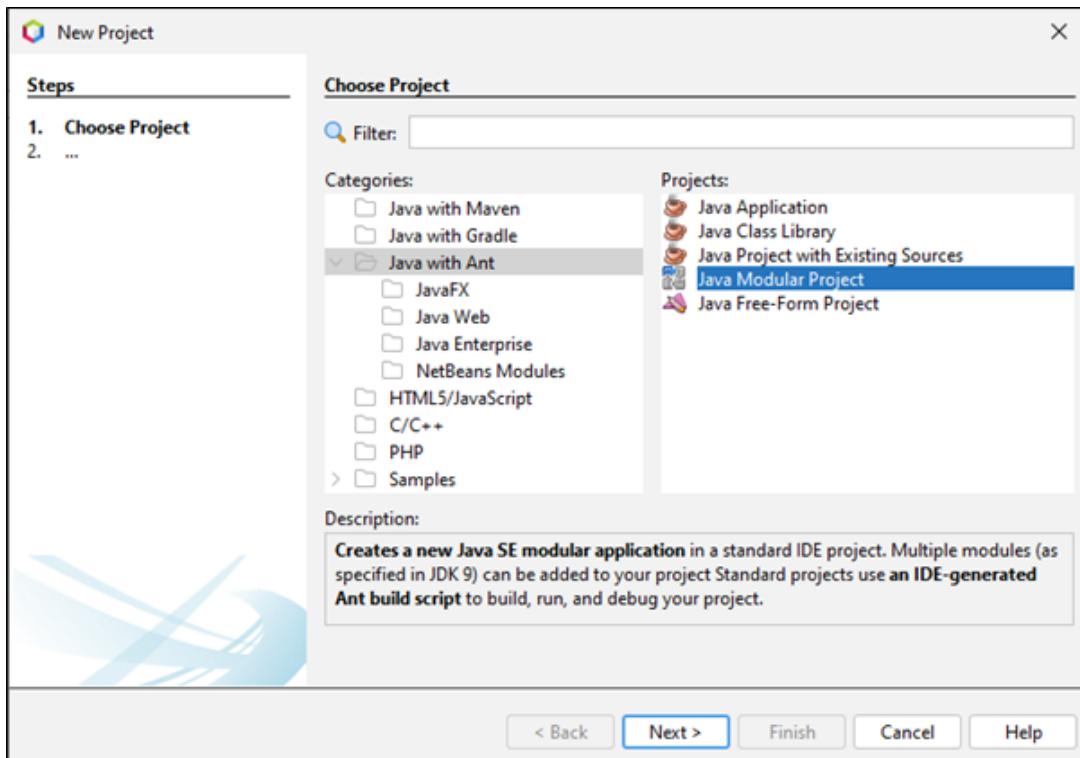


Figure 11.5: Creating a Modular Project

Name it as `JavaModuleDemo` and click **Finish**. It will be created.

Right-click the project name and click **New → Module**.

Specify a name **hello**.

Following code will be autogenerated inside the module, in a file named `module-info.java`.

```
module hello {  
}
```

Similarly, create another module named **test**.

Now, create a class `Hello` in the first module as shown in Code Snippet 16.

Code Snippet 16:

```
// Create a class named Hello in the com.example.hello package  
package com.example.hello;  
public class Hello {  
    public static void sayHello() {  
        System.out.println("Hello from module");  
    }  
}
```

Create another class named `TestHello` with a main method as shown in Code Snippet 17.

Code Snippet 17:

```
// Create a class named TestHello in the com.example.test package
package com.example.test;
import com.example.hello.Hello;
public class TestHello {
    public static void main(String[] args) {
        // Call the sayHello method from the Hello class in
        //the com.example.hello module
        Hello.sayHello();
    }
}
```

For the code given in Code Snippet 17 to work successfully, you must first export 'com.example.hello' package from hello module and then, include `hello` module in `test` module. Code Snippets 18 and 19 depict this.

Code Snippet 18:

```
// Create a module named com.example.hello with a module-info.java file
module hello {
    // Export the com.example.hello package to other modules
    exports com.example.hello;
}
```

Code Snippet 19:

```
// Create another module named com.example.test with a
//module-info.java file
module com.example.test {
    // Require the com.example.hello module
    requires com.example.hello;
}
```

The first module exports the `com.example.hello` package, which contains a class named `Hello` with a static method `sayHello` that prints a message to the console. The second module requires the first module and contains a class named `TestApp` with a `main` method that calls the `sayHello` method from the `Hello` class in the `com.example.hello` module.

The output of Code Snippet 17 is as shown in Figure 11.6.

```
Hello from module
```

Figure 11.6: Output of Using Modules

Some best practices for using modules in Java are as follows:



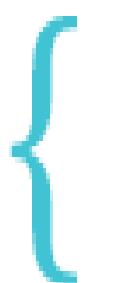
- Design your modules to be cohesive and loosely coupled. A module should have a clear and consistent purpose and should only depend on the modules that are necessary for its functionality. Avoid circular dependencies between modules, as they can cause problems in resolving and loading modules.



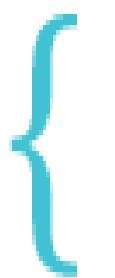
- Use the 'module-info.java' file to declare your module's name, dependencies, exports, opens, uses, and provides directives. These directives define the properties and relationships of your module, such as its public API, its required modules, its reflection access, and its service consumption and provision.



- Use the 'requires transitive' directive to propagate the dependencies of your module to its consumers. This way, you can avoid repeating the same dependencies in multiple modules and simplify the module graph.



- Use the 'exports' and 'opens' directives to control the visibility and accessibility of your module's packages. By default, all packages in a module are encapsulated and inaccessible to other modules. You can use the 'exports' directive to make a package available to other modules as part of your module's public API. You can use the 'opens' directive to make a package available for deep reflection by other modules, such as frameworks that use reflection to access private members.



- Use the 'uses' and 'provides' directives to declare and implement service interfaces. A service interface is an abstract type that defines a contract for a specific functionality. A service provider is a concrete type that implements a service interface. A service consumer is a module that uses a service interface. You can use the 'uses' directive to declare that your module consumes a service interface. You can use the 'provides' directive to declare that your module provides a service provider for a service interface.



- Use the 'java --list-modules' command to list the modules available in the JDK or in a custom runtime image. You can also use the 'java --describe-module' command to describe the properties and dependencies of a specific module.



- Use the 'jdeps' tool to analyze the dependencies of your modules or JAR files. You can use this tool to check if the modules or JAR files are modularized correctly and if they have any unused or missing dependencies.



- Use the 'jlink' tool to create custom runtime images that contain only the modules you require for your application. You can use this tool to reduce the size of your application, improve its performance, and make it standalone.

- **Records:** Records are a way of creating classes that are mainly used to hold data. They allow us to declare the state of the class as a list of fields in the class header and automatically generate constructors, getters, equals, hashCode, and toString methods for the class. Records can make the code more concise and consistent, especially when using value-based classes or data transfer objects. Records were introduced in Java 16.

An example of using records to create and use a record class named `Point` is shown in Code Snippet 20.

Code Snippet 20:

```
// Declare a record class named Point with two fields x and y
record Point(int x, int y) {}

// Create and use an instance of Point
public class RecordDemo {

    public static void main(String[] args) {
        // Create a point instance with x = 1 and y = 2
        Point point = new Point(1, 2);

        // Print the point instance using the generated
        // toString method
        System.out.println(point);

        // Access the fields using the generated getters
        System.out.println("x = " + point.x());
        System.out.println("y = " + point.y());
    }
}
```

Code Snippet 20 defines a record class `Point` with two immutable fields `x` and `y` and shows how to create, print, and access a point instance using the methods generated by the Java compiler. A record class `Point` with fields `x` and `y` is declared and used to create, print, and access a point object with the methods automatically provided by Java. The output of Code Snippet 20 is shown as shown in Figure 11.7.

```
Point[x=1, y=2]
x = 1
y = 2
```

Figure 11.7: Output of Records

- ◆ **Switch expressions:** Switch expressions are a way of writing switch statements that can return values. They allow us to use the `->` operator to assign values to different cases and use the `yield` keyword to return values from complex cases. Switch expressions can make the code more compact and readable, especially when using multiple or nested switch statements. Switch expressions were introduced in Java 14.

An example of using switch expressions to assign a letter grade based on a numeric score is shown in Code Snippet 21.

Code Snippet 21:

```
// Assign a letter grade based on a numeric score using
//switch expressions
public class SwitchDemo {
    public static void main(String[] args) {
        // A numeric score
        int score = 85;
        // A switch expression that returns a letter grade
        String grade = switch (score / 10) {
            case 10, 9 -> "A";
            case 8 -> "B";
            case 7 -> "C";
            case 6 -> "D";
            default -> {
                // A complex case that uses yield to return a value
                if (score > 0) {
                    yield "E";
                } else {
                    yield "F";
                }
            }
        };
        System.out.println("Grade: " + grade);
    }
}
```

```

    }
}

// Print the letter grade

System.out.println(grade);

}
}

```

Code Snippet 21 uses a `switch` expression to assign a letter `grade` based on a numeric score. The `switch` expression evaluates the score divided by 10 and matches it with a case label. If the case label matches, it returns the corresponding letter `grade` as a string. If none of the case labels match, it uses the default case to return either "E" or "F" using a `yield` statement. The `yield` statement allows the `switch` expression to return a value from a block of code. The code then, prints the letter `grade` using `System.out.println()`. The output of Code Snippet 21 is as shown in Figure 11.8.



B

Figure 11.8: Output of Using Switch Expressions

- ◆ **Text blocks:** Text blocks are a way of writing multi-line strings in Java. They allow us to use three double quotes (""""") to delimit a block of text that can span multiple lines, without having to use escape sequences or concatenation operators. Text blocks can make the code more readable and maintainable, especially when writing formatted text such as HTML, JavaScript Object Notation (JSON), or Structured Query Language (SQL). Text blocks were introduced in Java 15. An example of using text blocks to create and print a JSON string is shown in Code Snippet 22.

Code Snippet 22:

```

// Create a JSON string using text blocks
public class TextBlockDemo {
    public static void main(String[] args) {
        // A text block that contains a JSON string
        String json = """
            {
                "name": "John",
                "age": 25,
                "hobbies": [
                    "reading",
                    "coding",
                    "gaming"
                ]
            }
        """;
        System.out.println(json);
    }
}

```

```

        ]
    }
"""

// Print the JSON string
System.out.println(json);
}
}

```

Code Snippet 22 defines a `public class` named `TextBlockDemo`. The class has a `main` method, which is the entry point for any Java application. Inside the main method, Code Snippet 22 uses a `text block` to create a `JSON string`. A JSON string is a format for storing and exchanging data using key-value pairs.

The text block starts and ends with three double quotes (""""") and contains the JSON string as it is, without any extra indentation or escaping.

The JSON string has four keys: `name`, `age`, `hobbies`, and `address`. The values are `"John"`, `25`, an array of strings (`["reading", "coding", "gaming"]`), and an object with two keys (`"city"` and `"country"`).

Code Snippet 22 then, prints the JSON string to the standard output using the `System.out.println` method. The output of Code Snippet 22 is as shown in Figure 11.9.

```
{
  "name": "John",
  "age": 25,
  "hobbies": [
    "reading",
    "coding",
    "gaming"
  ]
}
```

Figure 11.9: Output of Text Blocks

- ◆ **Pattern matching for instanceof:** Pattern matching for `instanceof` is a way of simplifying the casting and testing of objects in Java. It allows us to combine the `instanceof` operator with a binding variable, which can be used in the following scope without explicit casting. Pattern matching for `instanceof` can make the code more concise and readable, especially when using multiple or nested `instanceof` checks. Pattern matching for `instanceof` was introduced as a preview feature in Java 14, and became a standard feature in Java 16.

An example of using pattern matching for `instanceof` to check and print the type and value of an object is shown in Code Snippet 23.

Code Snippet 23:

```
public class MyInstance {
    // An object that can be an Integer, a String, or null
    Object obj = 42;

    public void myMethod() {

        // A traditional way of checking and printing the type
        // and value of obj
        if (obj instanceof Integer) {
            int i = (Integer) obj; // Explicit casting //required
            System.out.println("Integer value: " + i);
        } else if (obj instanceof String) {
            String s = (String) obj; // Explicit casting required
            System.out.println("String value: " + s);
        } else {
            System.out.println("Unknown value");
        }
    }

    public void myMethod2() {

        // A modern way of checking and printing the type and
        // value of obj using pattern matching for instanceof
        if (obj instanceof Integer i) { // Binding variable i
            System.out.println("Integer value: " + i); // No
            //casting required
        } else if (obj instanceof String s) { // Binding
            //variables
            System.out.println("String value: " + s); // No
            //casting required
        } else {
            System.out.println("Unknown value");
        }
    }
}
```

```
}

public static void main(String[] args) {
    MyInstance myInstObj = new MyInstance();
    myInstObj.myMethod();
    myInstObj.myMethod2();
}
```

Code Snippet 23 defines a class named `MyInstance` that has an object field `obj` that can be an `Integer`, a `String`, or `null`. The class also has two methods `myMethod` and `myMethod2` that print the type and value of `obj` in different ways.

The method `myMethod` uses the traditional way of checking the type of `obj` using the `instanceof` operator and then, casting it to the appropriate type before printing it. This requires repeating the type name three times for each conditional block.

The method `myMethod2` uses the modern way of checking and printing the type and value of `obj` using pattern matching for `instanceof`. This allows us to test the type of `obj` and assign it to a binding variable of the proper type in one expression, without necessary explicit casting or repeating the type name.

The `main` method creates an instance of `MyInstance` and calls both methods to demonstrate the difference between the traditional and modern ways of using `instanceof`.

The output of Code Snippet 23 is as shown in Figure 11.10.

```
Integer value: 42
Integer value: 42
```

Figure 11.10: Output of Pattern Matching Using `instanceof`

11.4 Check Your Progress

1. What is the purpose of the @Override annotation in Java?

(A)	It indicates that a method is overriding a superclass method	(C)	It improves the readability and maintainability of code
(B)	It helps the compiler to detect errors in overriding methods	(D)	All of these

2. Which of these features were introduced in recent versions of Java?

(A)	Applets	(C)	Lambda Expression
(B)	Sealed Class	(D)	Enhanced Java Virtual Machine (JVM)

3. What is the advantage of using generics in Java?

(A)	They will make the code more type-safe and reduce the necessity for type casting	(C)	They allow the code to work with any kind of object
(B)	They increase the performance and efficiency of the code	(D)	They enable the code to use multiple inheritance

4. What is a stream in Java?

(A)	A sequence of bytes that can be read from or written to a file, network, or other source, or destination	(C)	A sequence of elements that can be processed by applying operations such as filtering, mapping, reducing, or collecting
(B)	A sequence of character that can be processed by a scanner, formatter, or other utility class	(D)	A sequence of events that can be handled by a listener, observer, or callback function

5. What is a module in Java?

(A)	A collection of related classes and interfaces that can be grouped together and reused in different applications	(C)	A mechanism for encapsulating data and behavior in a single entity that can be instantiated multiple times
(B)	A self-contained unit of code that can declare its dependencies and exports and can be compiled and run independently	(D)	A feature that allows multiple inheritance of state and behavior from more than one superclass

11.4.1 Answers

1.	D
2.	B
3.	A
4.	C
5.	B

```
    g.pac
    main()
    string pac!
    {} ma
    string pr
    catch {}
    aids string
    y catch
```

Summary

- Lambda expressions are a way of writing anonymous functions in Java.
 - Generics allows us to work with different types of objects, without having to write duplicate code or cast the objects.
 - Annotations are a way of adding metadata to Java code.
 - Streams are a way of processing collections of data in a declarative and functional way.
 - Modules are a way of organizing and encapsulating Java code.
 - Records are a way of creating classes that are mainly used to hold data.
 - Switch expressions are a way of writing switch statements that can return values.
 - Text blocks are a way of writing multi-line strings in Java.
 - Pattern matching for instanceof is a way of simplifying the casting and testing of objects in Java.

Try It Yourself

- Suppose you are developing a Java application that uses a third-party library called **FooLib**. FooLib provides some useful classes and methods for manipulating data, but it also has some bugs and limitations. You want to use the `@Deprecated` annotation to mark the classes and methods from FooLib that you should avoid using in your code. You also want to use the `@SuppressWarnings` annotation to suppress the compiler warnings that are generated by using FooLib.

Write a Java Code Snippet that demonstrates how to use these two annotations in your application. Assume that FooLib has a class called `Foo` and a method called `bar()` that are both deprecated and generate warnings.

- Suppose you are developing a Java application that uses polymorphism to handle different types of shapes. You have an abstract class called `Shape` and three subclasses called `Circle`, `Rectangle`, and `Triangle`. You also have a method called `printArea(Shape shape)` that takes a `Shape` object as a parameter and prints its area.

Write a Java Code Snippet that uses pattern matching with `instanceof` to check the type of the `shape` parameter and cast it to the appropriate subclass without using an explicit cast operator. Assume that each subclass has a method called `getArea()` that returns the area of the shape.

Session - 12

JDK 20 New Features and Functionalities

Welcome to the Session, **JDK 20 New Features and Functionalities**.

This session explains new features of JDK 20 in many functional areas. Features such as Virtual Threads, Vector API, Structured Concurrency, Scoped Values, and Record patterns are covered. The session also takes a deep dive into Foreign Function and Memory API and Pattern matching for switch statement and expressions.

In this Session, you will learn to:

- Explain Virtual Threads
- Explain Vector API
- Describe Structured Concurrency
- Outline Scoped Values
- Describe Foreign Function and Memory API
- Summarize Features of Record Patterns
- Explain Pattern Matching for switch Statements and Expressions



12.1 Additional Features in Java and JDK 20

Java 20 is the latest short-term incremental release of Java, which was released on March 21, 2023. It introduces several new features. Some of these features are:

- **Virtual Threads:** Virtual threads are a new feature in Java 20 that allow creating lightweight threads that can run on a small stack and are managed by the JVM. This can improve the performance and scalability of concurrent applications that use a large number of threads.

A virtual thread is an instance of `java.lang.Thread`, but unlike a platform thread, it is not tied to a specific Operating System (OS) thread. A virtual thread still runs code on an OS thread, but when it calls a blocking I/O operation, the JVM suspends the virtual thread until it can be resumed. The OS thread associated with the suspended virtual thread is now free to perform operations for other virtual threads.

Virtual threads are implemented in a similar way to virtual memory. To simulate a lot of memory, an OS maps a large virtual address space to a limited amount of RAM. Similarly, to simulate a lot of threads, the JVM maps a large number of virtual threads to a small number of OS threads.

Virtual threads have several advantages over platform threads, such as:

• They have a shallow call stack, which reduces the memory consumption and improves the performance.

• They support thread-local variables, but they are immutable and inheritable, which avoids the problems of mutability, memory leaks, and excessive memory footprints.

• They simplify the programming model and reduce the risk of errors and resource leaks in concurrent applications.

Virtual threads are suitable for running tasks that spend most of the time blocked, often waiting for I/O operations to complete. However, they are not intended for long-running CPU-intensive operations.

- **Vector API:** The Vector API is a new feature in Java 20 that provides a way to express vector computations which can be compiled and optimized for various CPU architectures.

Vector operations can improve the performance of data-intensive applications such as machine learning and image processing applications.

A vector is a sequence of primitive values of the same type that can be operated on in parallel. For example, a vector of four integers can perform four additions, four multiplications, or four comparisons at the same time. This can speed up the execution of loops that perform the same operation on multiple elements of an array.

The Vector API is implemented as a set of classes and methods in the `jdk.incubator.vector` package. It allows creating and manipulating vectors of different shapes (number of lanes) and species (element type and size). It also supports various operations on vectors, such as arithmetic, bitwise, logical, comparison, conversion, and reduction.

The Vector API is platform-independent, meaning that it can run on any CPU architecture that supports vector instructions. This technique can rearrange the elements of a vector to match the instruction set of the CPU.

Benefits of Using Vector API over Traditional Loops:

The benefits of using the Vector API over traditional loops are mainly related to performance and simplicity of vector computations.

- The Vector API can leverage the data-parallel capabilities of modern CPUs, which can execute the same operation of multiple elements of an array at the same time.
- This can speed up the execution of loops that perform the same operation of multiple elements of an array.

- The Vector API can also adapt to different CPU architectures and generate the optimal vector instructions for the target platform at runtime.
- This can avoid the necessity to write platform-specific code and simplify the vectorization of operations.

- **Structured Concurrency:** In Java 20, Structured Concurrency is introduced to make multithreaded programming easier by providing a new API. This API treats multiple tasks running in different threads as a single unit of work, thereby, streamlining error handling and cancellation, improving reliability, and enhancing observability.

Structured concurrency is based on the idea that concurrent tasks should form a tree-like hierarchy, where each task has a parent task that is responsible for creation, coordination, and termination. A task can create subtasks, which are also tasks, and wait for them to complete. A task can also cancel its subtasks or be cancelled by its parent task. A task can also propagate exceptions to its parent task or handle them locally.

The main class of the structured concurrency API is `StructuredTaskScope`, which represents a scope for a group of tasks. A scope can be created using a `try-with-resources` statement, which ensures that the scope is closed when the block exits. A scope can also be configured with different policies, such as `ShutdownOnFailure` and `ShutdownOnCancel`.

`ShutdownOnFailure` aborts all tasks within the scope should any of them encounter an error, whereas, `ShutdownOnCancel` terminates all tasks within the scope if the scope itself is canceled.

To create a task within a scope, the `fork` method can be used, which takes a `Callable` or a `Runnable` instance as an argument and returns a `Future`. The `fork` method does not block the current thread but returns immediately. To wait for the completion of all the tasks in the scope, the `join` method can be used, which blocks the current thread until all the tasks are done. To get the result of a task, `resultNow` method can be used, which returns value of the `Future` or throws an exception if task failed or was cancelled.

Benefits of Using Structured Concurrency:

- It minimizes the chances of thread leaks and delays in cancellation, which are prevalent risks when dealing with the termination and interruption of the concurrent tasks.
- It simplifies the management of errors and cancellations by passing on exceptions and interruptions to the parent task.
 - It also performs cancelling all subtasks within the scope in case of a failure or the scope being cancelled.
- It enhances reliability by ensuring that all tasks within the scope are either completed or canceled before closing the scope.
 - This prevents any task from being left incomplete or abandoned.
- It boosts the visibility by offering a means to monitor the status and advancement of tasks within the scope and collect diagnostic data related to those tasks.
- It simplifies the programming model, by providing a clear and intuitive way to structure and control flow of concurrent tasks, and by avoiding the necessity to use low-level synchronization primitives.

- **Scoped Values:** Scoped values are a new feature in Java 20 that allows to safely and efficiently share immutable data within and across threads for a limited period of time. Scoped values are intended to replace thread-local variables, which have various drawbacks such as mutability, memory leaks, and excessive memory footprints.

A scoped value is an instance of the class `jdk.incubator.concurrent.ScopedValue<T>`, which represents a key-value pair. The key is the scoped value itself and the value is an object of type `T`. A scoped value can be bound to a value in the current thread using the `where` method, which returns a `ScopedValue.Carrier` object that contains the mapping.

A `ScopedValue.Carrier` object can be used to run an operation (a `Runnable` or a `Callable`) with all the scoped values in the mapping bound to their values in the current thread. When the operation completes, the scoped values revert to being unbound, or revert to their previous values when previously bound.

Scoped values advantages over thread-local variables:

- They are immutable and can only be set once for a bounded period of execution, which prevents data from flowing in any direction between components.

- They are automatically removed from the thread when the operation ends, which avoids memory leaks and reduces memory consumption.

- They support inheritance by child threads, but only for the duration of operation, which avoids allocating unnecessary storage for parent thread-local variables.

- **Foreign Function and Memory API:** The Foreign Function and Memory API (FFM API) is a new feature in Java 20 that enables Java programs to interoperate with code and data outside the Java runtime. This API enables Java programs to call native libraries and process native data without the brittleness and dangers of Java Native Interface (JNI).

The FFM API consists of two main components: the foreign function API and the foreign memory API. The foreign function API allows Java programs to invoke functions written in other languages, such as C, using a method handle (PREVIEW feature, which means that it is not yet mature in current version). The foreign memory API allows Java programs to access and manipulate memory regions that are not managed by the JVM, such as off-heap memory or memory-mapped files, using a memory segment (PREVIEW feature).

The FFM API is designed to be safe, efficient, and expressive. It offers controlled and secure access to foreign code and data through mechanisms such as segment scopes (PREVIEW features) and restricted methods (PREVIEW feature). Native method handles (PREVIEW feature) are also available. It also leverages the performance and optimization capabilities of the JVM, such as universal variable shuffling (PREVIEW feature), which can generate the best vector instructions for the target platform at runtime. Additionally, it includes features such as memory layouts (PREVIEW feature) and layout paths (PREVIEW feature), enhancing the programming model's convenience and data structure manipulation. The introduction of `jextract` (PREVIEW feature) further simplifies complex data structure creation. `jextract` is a tool that generates Java bindings from a native library header.

Benefits of Using FFM API over JNI:

- The FFM API provides a higher-level API that is more user-friendly and intuitive than JNI. The FFM API allows Java programs to invoke foreign functions using method handles, which are first-class objects that can be manipulated and composed.
- The FFM API supports automatic memory management, which reduces the risk of memory leaks and errors. The FFM API uses segment scopes to control the lifecycle and access of memory segments. When a segment scope is closed, all the memory segments associated with it are invalidated and their backing memory regions are deallocated.
- The FFM API ensures type safety, which prevents data corruption and crashes. The FFM API uses memory layouts to describe the structure and alignment of foreign data. The FFM API also uses layout paths to access the elements of complex data structures, such as structs and unions.

- **Record Patterns:** Record patterns are a preview feature that improves pattern matching for record classes. Record classes are classes that act as transparent carriers for immutable data. Record patterns allow to deconstruct record instances into their components and bind them to variables in a pattern matching expression.

Record patterns can test whether a value is an instance of a record class type and if it is, to recursively perform pattern matching on its component values. Record patterns can also extract the component values from the value directly, without calling accessor methods.

A record pattern consists of a type and a record component pattern list (that may be empty). A record component pattern list can contain pattern variables, which are declared with a type or inferred with `var` and can be used to bind and extract the component values.

A record pattern can be used in the `instanceof` operator, the switch expression, and the switch statement. They can also be nested with other patterns, such as type patterns and constant patterns, to create more complex patterns.

Record patterns were first proposed as a preview feature in an earlier proposal and then, delivered in Java 19.

Java 20 proposes the second preview aiming to gather more feedback and improvement suggestions from developer community after the use. Record patterns are expected to be a permanent feature in a future Java release.

- **Pattern Matching:** Pattern Matching for switch is a new feature in Java 20 that enables Java programs to use patterns in switch expressions and statements. Patterns are a way of testing whether a value has a certain shape or property and extracting information from it. Patterns can simplify and improve the readability of switch expressions and statements, which are often used for conditional logic and data extraction.

Pattern Matching for switch introduces three kinds of patterns that can be used in switch cases: type patterns, constant patterns, and record patterns. Type patterns can test whether a value is an instance of a given type and bind it to a variable of that type. Constant patterns can test whether a value is equal to a given constant. Record patterns can test whether a value is an instance of a record class and extract its component values.

Pattern Matching for switch also introduces some improvements and changes to the syntax and semantics of switch expressions and statements, such as:

- Simplifying the grammar for switch labels, by removing the need for a colon after a case label.
- Catching the type for an exception thrown when no switch label applies at runtime, from `NullPointerException` to `IncompatibleClassChangeError`, when the selector expression is an enum class.
- Strengthening the test for determining whether a switch block is exhaustive, by requiring that all possible values of the selector expression are covered by switch cases.
- Inferring the type arguments for generic record patterns, by using the type of selector expression and the types of the pattern variables.

12.2 Check Your Progress

1. Which of the following statements is true for virtual threads?

(A)	Virtual threads are tied to a specific operating system thread and cannot be suspended or resumed by the JVM	(C)	Virtual threads do not support thread-local variables, which limits the sharing of data across threads
(B)	Virtual threads have a deep call stack, which increases the memory consumption and reduces the performance	(D)	Virtual threads are managed by the JVM and can run on a small stack and be suspended or resumed by the JVM

2. What is the main purpose of Vector API in Java 20?

(A)	To provide a way to create and manipulate vectors of different shapes and species	(C)	To provide a way to access and process vector data from native libraries and memory
(B)	To provide a way to express vector computations that can be compiled and optimized for various CPU architectures	(D)	To provide a way to support vector graphics and animations in Java applications

3. What is the main class of structured concurrency API in Java 20?

(A)	StructuredTask	(C)	StructuredTaskScope
(B)	StructuredExecutor	(D)	StructuredFuture

4. What is the main difference between scoped values and thread-local variables in Java 20?

(A)	Scoped values are automatically removed from the thread when the scope ends and thread-local variables are retained until explicitly removed	(C)	Scoped values can only be accessed by the thread that created them and thread-local variables can be accessed by any thread within the same scope
(B)	Scoped values are mutable and thread-local variables are immutable	(D)	Scoped values are declared as private instance fields and thread-local variables are declared as public static fields

5. Which of the following statements is false about record patterns in Java 20?

(A)	Record patterns can test whether a value is an instance of a record class type and extract its component values	(C)	Record patterns can be nested with other patterns, such as type patterns and constant patterns, to create more complex patterns
(B)	Record patterns can be used in the instanceof operator, the switch expression, and the switch statement	(D)	Record patterns can match any value, including the null value

12.2.1 Answers

1.	D
2.	B
3.	C
4.	A
5.	D

```
g package main
import java.util.StringJoiner;
public class Main {
    public static void main(String[] args) {
        StringJoiner sj = new StringJoiner(" ");
        sj.add("Hello")
            .add("World");
        System.out.println(sj);
    }
}
```

Summary

- Virtual Threads allow creating lightweight threads that can run on a small stack and are managed by the JVM.
- Vector API provides a way to express vector computations that can be compiled and optimized for various CPU architectures.
- Structured concurrency provides a framework for managing the lifecycle and dependencies of concurrent tasks.
- Scoped Values provide a mechanism for sharing immutable data within and across threads for a limited period of time.
- Foreign Function and Memory API provides a way to access native libraries and memory without using the Java Native Interface (JNI).
- Record Patterns allows using record types in pattern matching expressions, such as `instanceof` and `switch`.
- Pattern Matching for switch extends the `switch` statement and expression to support pattern matching, which can test the shape and properties of an object.



Try It Yourself

1. You are developing a Java application that performs image processing tasks, such as resizing, cropping, and filtering images. You want to use the Vector API to speed up the execution of these tasks by taking advantage of the data-parallel capabilities of modern CPUs.
 - Explain how the Vector API can be useful here and what are its main components? Present your research in a PowerPoint presentation.

2. You are developing a Java application that performs machine learning tasks, such as training and testing neural networks. You want to use the Vector API to improve the speed and performance of the application.
 - Perform research on how Vector API can be useful here and which components of the API can be utilized. Present research outcome using a PowerPoint presentation.

3. You are developing a Java application that processes a list of employees and calculates their salaries. The employees are represented by following record classes:

```
recordEmployee (Stringname, intage, Stringdepartment) {}  
recordManager (Employeeemployee, doublebonus) {}  
recordDeveloper (Employeeemployee, Stringlanguage) {}
```

- Write a Code Snippet that uses a switch expression to match each employee against a record pattern and return their salary. Assume that the salary is calculated as follows:
 - ◆ For managers, the salary is 100,000 plus the bonus.
 - ◆ For developers, the salary is 80,000 plus 10,000 if the language is Java, or 5,000 otherwise.
 - ◆ For other employees, the salary is 60,000.

4. You are developing a Java application that processes a list of shapes and calculates their areas. The shapes are represented by following record classes:

```
recordPoint (double x, double y) {}  
recordCircle (Point center, double radius) {}  
recordRectangle (Point topLeft, Point bottomRight) {}
```

- Explain how record patterns work and what are their main advantages.
- Write a Code Snippet that uses a switch expression to match each shape against a record pattern and return its area. Assume that the area is calculated as follows:
 - ◆ For circles, the area is pi times the square of the radius.
 - ◆ For rectangles, the area is the absolute value of the product of the differences of the x and y coordinates of the top left and bottom right points.
 - ◆ For other shapes, the area is zero.
- For rectangles, area is the product of the differences of x and y coordinates of top left and bottom right points.

Appendix



Appendix

Sr. No.	Case Studies
1.	<p>Library Management System</p> <p>Develop a Library Management System by incorporating operators, decision-making, loops, classes, objects, methods, arrays, and string. The system should incorporate following functionalities:</p> <p>Operators, Decision Making, and Loops:</p> <ul style="list-style-type: none">• Implement a console based menu-driven system that allows performing operations such as adding books, issuing books to members, returning books, and displaying available books.• Use Loops for menu navigation and decision-making statements to execute different functionalities based on user input. <p>Classes, Objects, and Methods:</p> <ul style="list-style-type: none">• Design classes such as 'Book', 'Member', and 'Library'.• Each class should have appropriate attributes and methods. For instance, 'Book' class might have attributes such as 'bookId', 'title', 'author', and methods such as 'displayBookDetails()'.• Implement methods for issuing, returning, and displaying book details within 'Library' class. <p>Arrays and Strings:</p> <ul style="list-style-type: none">• Utilize arrays to manage the collection of books and store member information.• Strings should be used to store book titles, authors, member names, and so on.

Appendix

Sr. No.	Case Studies
2.	<p>Online Shopping Application</p> <p>Develop an Online Shopping Application with a console based menu-driven system using packages, interfaces, classes, inheritance, and exceptions to accomplish following tasks:</p> <p>Packages and Structure:</p> <ul style="list-style-type: none">• Implement different packages to organize functionalities such as 'shop', 'payment', and 'user'.• Each package should contain relevant classes and interfaces. <p>Interface and Inheritance:</p> <ul style="list-style-type: none">• Create an 'OnlineShop' interface with methods such as 'browseProducts()', 'addToCart()', and 'purchase()'.• Implement classes such as 'Customer', 'Admin', and 'Seller' inheriting from an abstract class such as 'User' that implements an 'OnlineShop' interface.• Utilize inheritance to distinguish roles and functionalities among users. <p>Exception Handling:</p> <ul style="list-style-type: none">• Implement custom exception such as 'InsufficientStockException' for when the stock of a product is insufficient to fulfil an order.• Handle various exceptions that may occur during shopping such as 'PaymentFailureException' or 'InvalidCredentialsExceptions'.