

Learning Java - A Foundational Journey



Streams JavaFX
Classes
annotations volatile
public
Lambdas Modules
Collections
final

Learning Java - A Foundational Journey

© 2021 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 - 2021



Preface

The book, **Learning Java - A Foundational Journey**, 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 book intends to familiarize the reader with the latest version of Java, that is, Java SE 15. 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. The book covers Java SE features such as lambda expressions, functional programming, and Stream API. It also covers various enhancements 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

BE AHEAD OF EVERYONE ELSE READ ARTICLES



Onlinevarsity



Onlinevarsity App for **Android** devices

Download from **Google Play Store**

Onlinevarsity

**24 x 7
Access To
Learning**



Table of Contents

Sessions

- Session 01: Introduction to Java
- Session 02: Variable, Data Types, and Operators
- Session 03: Decision-Making 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: New Date and Time API
- Session 11: Annotations and Base64
- Session 12: Functional Programming in Java
- Session 13: Stream API
- Session 14: More on Functional Programming
- Session 15: Additional Features of Java
- Session 16: Swing and JavaFX

Onlinevarsity

INDUSTRY BEST PRACTICES

SYNC WITH THE INDUSTRY



Session - 1

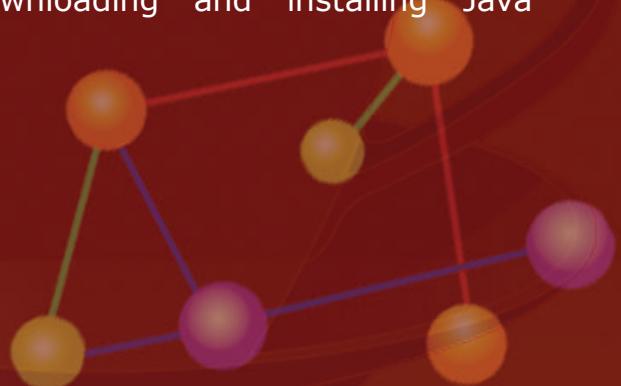
Introduction to Java

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.

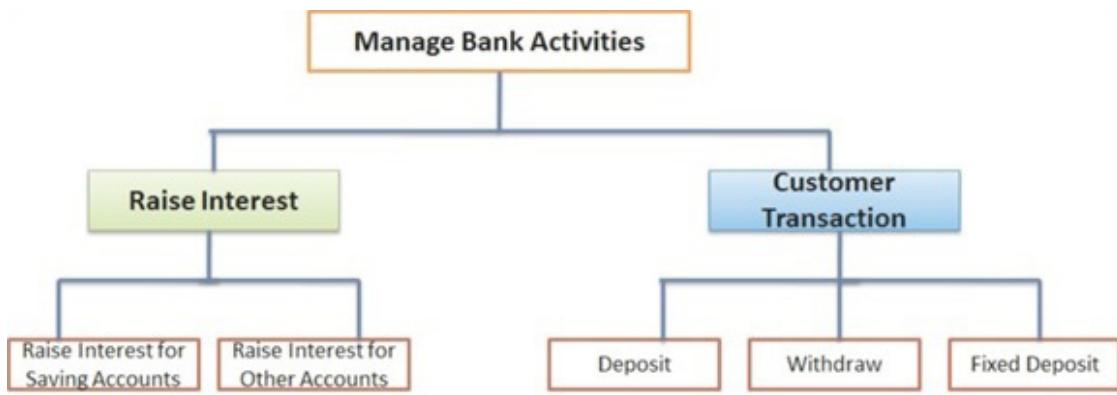


Figure 1.1: Decomposing of Problem into Subprograms

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.

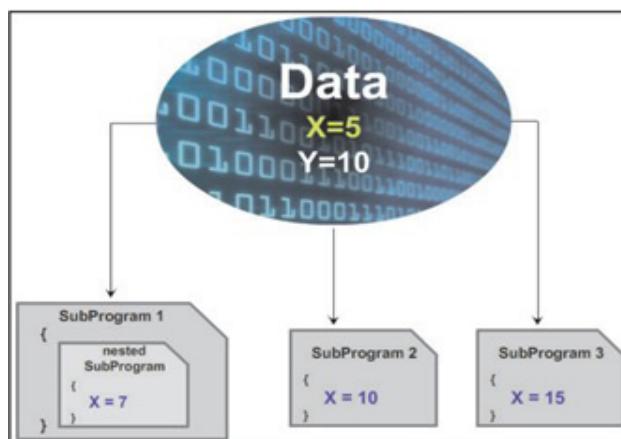


Figure 1.2: Data Shared Between Subprograms

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.

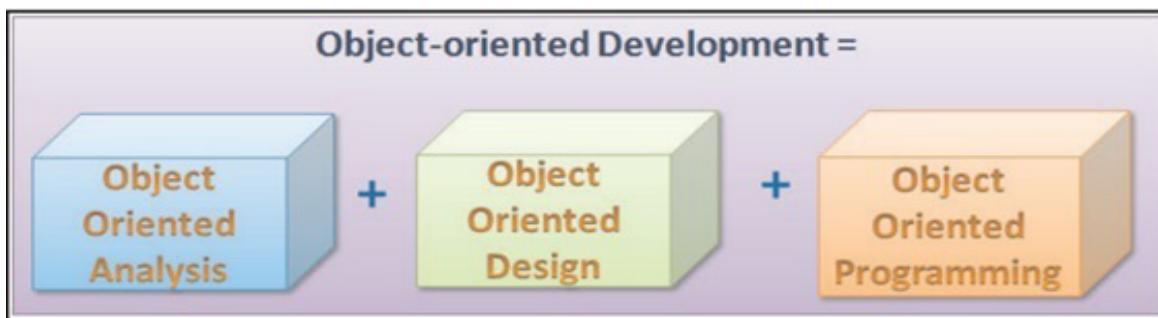


Figure 1.3: Object-oriented Software Development

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.

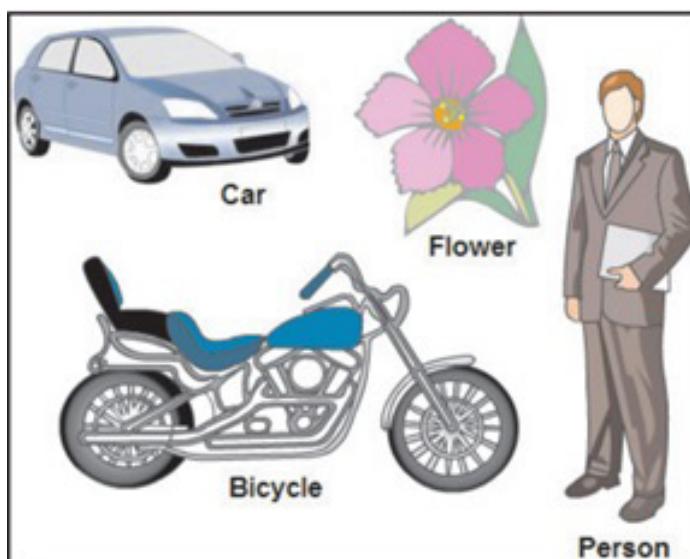


Figure 1.4: Real-world Entities

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

The concept of objects in the real-world can be extended to the programming world, where software ‘objects’ can be defined. Similar to its real-world counterpart, a software object has state and behavior. The ‘state’ of the software object refers to its characteristics or attributes, whereas the ‘behavior’ of the software object comprises its actions.

Figure 1.5 shows a software object, such as a **Car** with its state and behavior.



Figure 1.5: Software Object

As shown in figure 1.5, the object **Car** will have the state as color, make, or model. On the other hand, the behavior of a car includes: driving, changing gear, increasing speed, and applying brakes.

Advantages of using objects are as follows:

- They help to understand the real-world.

- They map attributes and actions of real-world entities with state and behavior of software objects.

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 1.6 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.

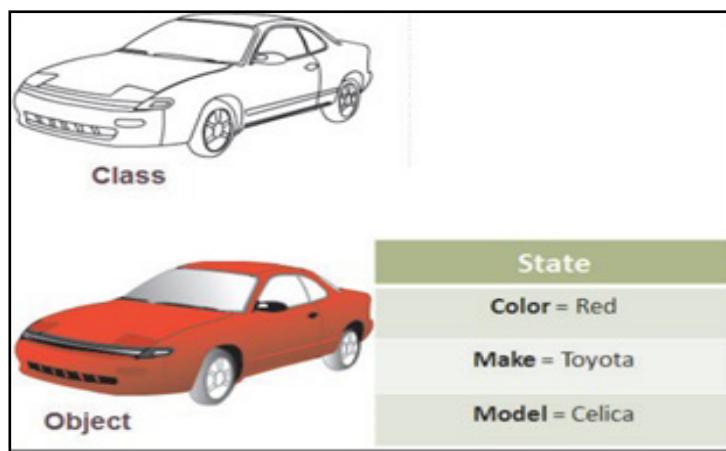


Figure 1.6: Class and an Object

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

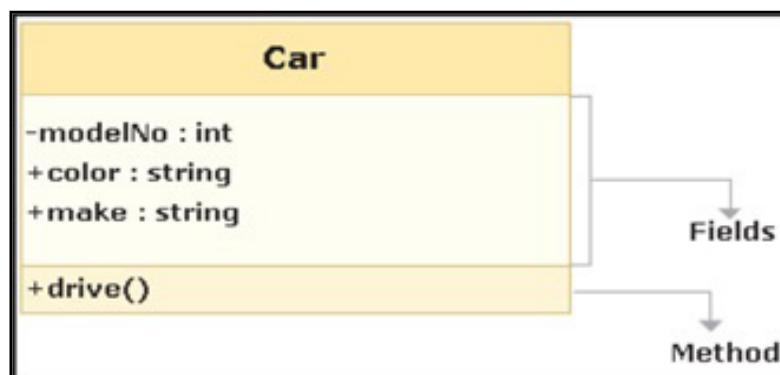


Figure 1.7: Class Design

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.

1.4 History of Java

The brief timeline of Java is depicted in figure 1.8.

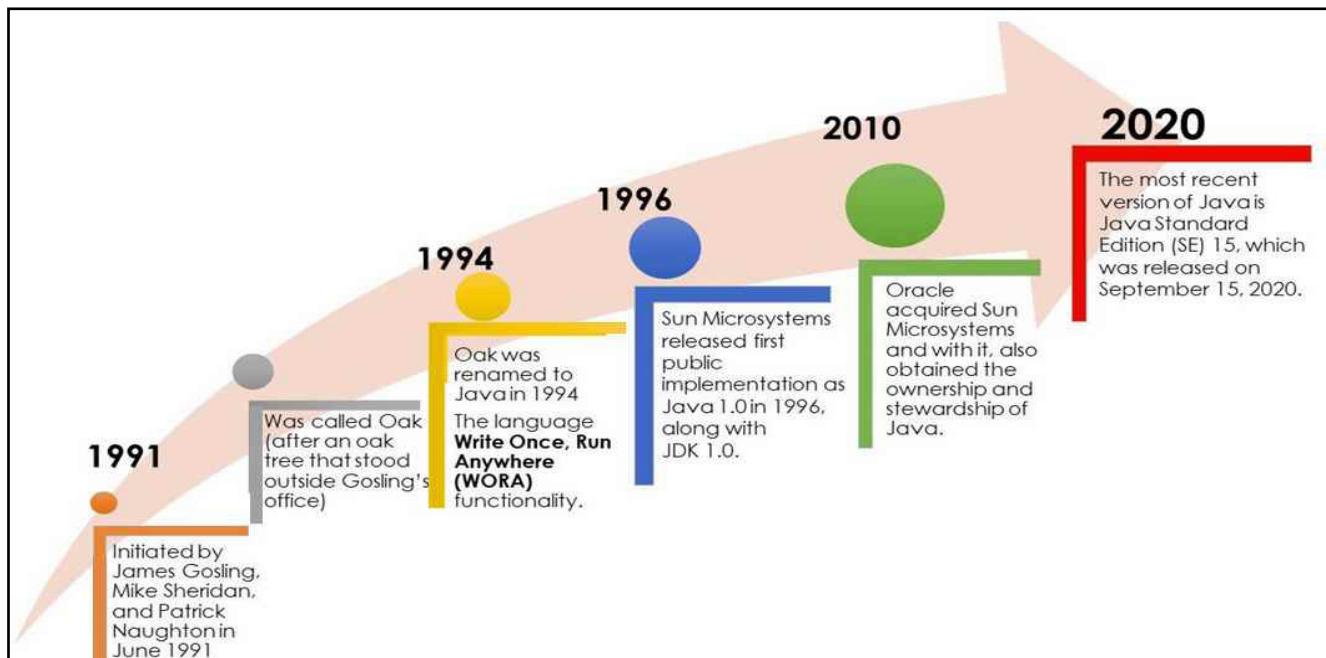


Figure 1.8: Timeline of Java

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

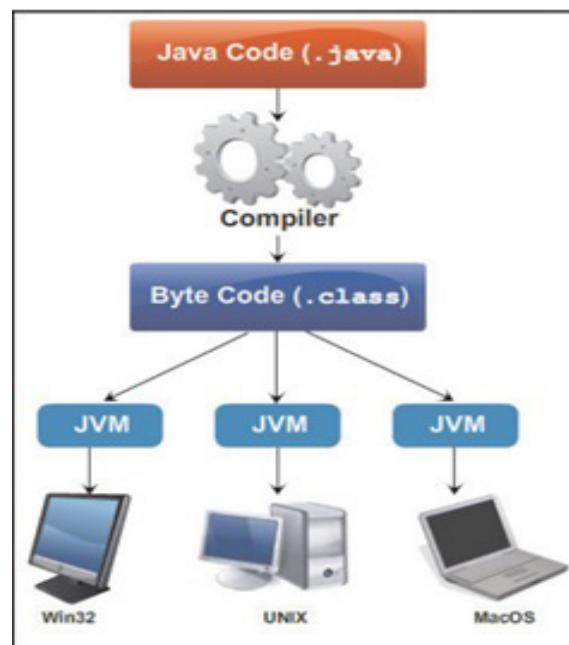


Figure 1.9: Bytecode Execution on Different JVMs

- **Java Runtime Environment (JRE)** - Java Runtime Environment is also written as Java 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 1.2 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 1.2: Differences Between JRE and Java SE

Table 1.3 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 1.3: 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.

Class data sharing is supported with the G1, serial, parallel, and parallelOldGC garbage collectors. The shared string feature (part of class data sharing) supports only the G1 garbage collector on 64-bit non-Windows platforms.

The primary motivation for including CDS in Java SE is to decrease in startup time. The smaller the application relative to the number of core classes it uses, the larger the saved fraction of startup time.

The footprint cost of new JVM instances has been reduced in two ways:

A portion of the shared archive on the same host is mapped as read-only and shared among multiple JVM processes. Otherwise, this data would require to be replicated in each JVM instance, which would increase the startup time of your application.

The shared archive contains class data in the form that the Java Hotspot VM uses it. The memory that would otherwise be required to access the original class information in the runtime modular image, is not used. These memory savings allow more applications to be run concurrently on the same system. In Windows applications, the memory footprint of a process, as measured by various tools, might appear to increase, because more pages are mapped to the process's address space. This increase is offset by the reduced amount of memory (inside Windows) that is required to hold portions on the runtime modular image. Reducing footprint remains a high priority.

1.6.1 Application Class-Data Sharing

To further reduce the startup time and the footprint, Application Class-Data Sharing (ApsCDS) is introduced that extends the CDS to include selected classes from the application class path.

This feature allows application classes to be placed in a shared drive. The common class metadata is shared across different Java processes. AppCDS allows the built-in system class loader, built-in platform class loader, and custom class loaders to load the archived classes. When multiple JVMs share the same archive file, memory is saved and overall system response time improves.

1.7 Downloading and Installing the JDK

To download the JDK, follow the link <https://www.oracle.com/java/technologies/javase-jdk15-downloads.html>.

- **Step 1:** Click the required version of JDK as shown in figure 1.10.

Product / File Description	File Size	Download
Linux ARM 64 RPM Package	141.82 MB	 jdk-15.0.2_linux-aarch64_bin.rpm
Linux ARM 64 Compressed Archive	157 MB	 jdk-15.0.2_linux-aarch64_bin.tar.gz
Linux x64 Debian Package	154.81 MB	 jdk-15.0.2_linux-x64_bin.deb
Linux x64 RPM Package	162.03 MB	 jdk-15.0.2_linux-x64_bin.rpm
Linux x64 Compressed Archive	179.35 MB	 jdk-15.0.2_linux-x64_bin.tar.gz
macOS Installer	175.93 MB	 jdk-15.0.2_osx-x64_bin.dmg
macOS Compressed Archive	176.51 MB	 jdk-15.0.2_osx-x64_bin.tar.gz

Figure 1.10: JDK Versions

- **Step 2:** Next, accept license agreement and then, click **Download jdk-15.0.2_windows-x64_bin.exe** as shown in figure 1.11.



Figure 1.11: JDK Download Window

- **Step 3:** Run the JDK Installer

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

To run the JDK installer:

- Start the JDK 15 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 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 15 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-15\bin\javac" MyClass.java
```

To set the PATH variable permanently, add the full path of the jdk-15\bin directory to the PATH variable. Typically, the full path is:

```
C:\Program Files\Java\jdk-15\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-15\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 Experimental JIT Compiler

Experimental JIT compiler is a part of Project Graal.

What is Project Graal?

Project Graal was created by Oracle with the ultimate goal of improving JVM based languages performance to match performance levels that native languages enjoy. One part of this project involves implementing a high performance dynamic Java compiler and interpreter also known as a Just-In-Time (JIT) compiler.

Just-In-Time Compiler

JIT compilation is a form of dynamic compilation that combines the use of Ahead-Of-Time (AOT) compilation and interpretation, and thus, has advantages and disadvantages from both technologies.

Ahead-Of-Time Compilation

An AOT compiler was added to the OpenJDK as part of the Java 9 release. Main goal was to improve the start-up time of Java applications with minimum impact to the application's peak performance. While it is not as fast as a warmed up JIT compilation, AOT avoids incurring performance penalties from having to repeat interpreted invocations.

Interpretation

It is a process that interprets code and performs specified actions. There are different ways an interpreter can work: it can parse the code and execute it directly, turn the code into an intermediate representation and execute that, or it can execute code that has already been pre-compiled. Each of these different methods has their own advantages and disadvantages.

JIT compilation combines the start-up speed of AOT compiled code with the flexibility of interpretation to create a faster solution. Main difference between JIT and AOT is that instead of interpreting bytecode each time a method is invoked, a JIT compiler will compile the bytecode into the machine code instructions of the running machine, and then, invoke this object code instead. Another difference is that AOT also turns the bytecode of a virtual machine into native machine code so that interpreted code can be executed.

Using experimental compiler

Unfortunately, this feature is not available for everyone as it only works for Linux/x64 platforms at the moment.

It is easy to enable experimental JIT compiler though it is a part of JDK. User just has to set following options when running Java from the command line:

```
-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler
```

After this, user can start testing it out. JIT compiler itself is in testing phase. It is worth being aware that at this time, this option is not being created to meet performance of the existing JIT compilers. Instead, it exists in order to investigate feasibility of a Java-based JIT for JDK.

1.9 New Features of Java

Table 1.4 outlines various new features introduced between Java 9 to Java 15.

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 13	Reimplement the Legacy Socket API
	Dynamic CDS Archive
	FileSystems.newFileSystem() Method
	Support for Unicode
	DOM and SAX Factories with Namespace

Java Version	Features
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

Table 1.4: New Features of Java

1.10 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 1.12 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 1.12: Structure of a Java Class

As shown in figure 1.12, 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.

1.11 Developing a Java Program on Windows Platform

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

1. JDK 15 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

1.11.1 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 1.13 shows the **Save As** dialog box to save the `HelloWorld.java` file.

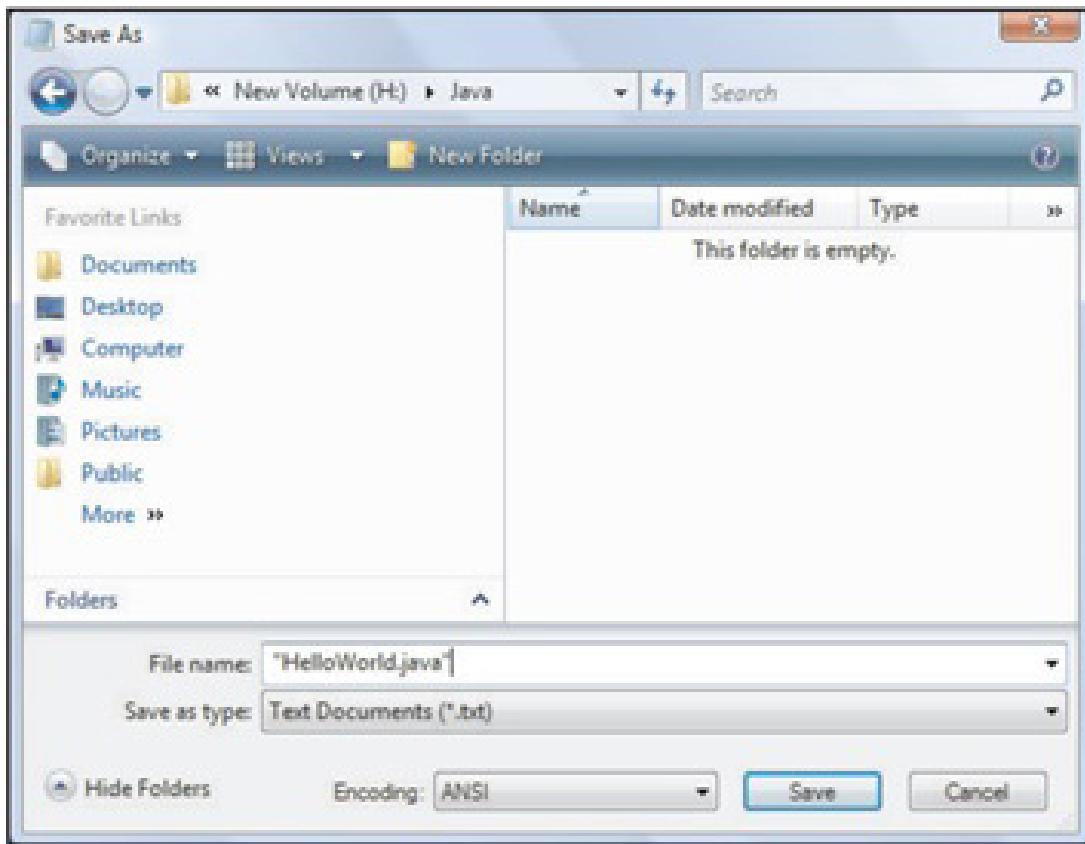


Figure 1.13: Notepad Editor - Save As Dialog Box

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

1.11.2 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 1.14 shows the compilation process of the Java program.

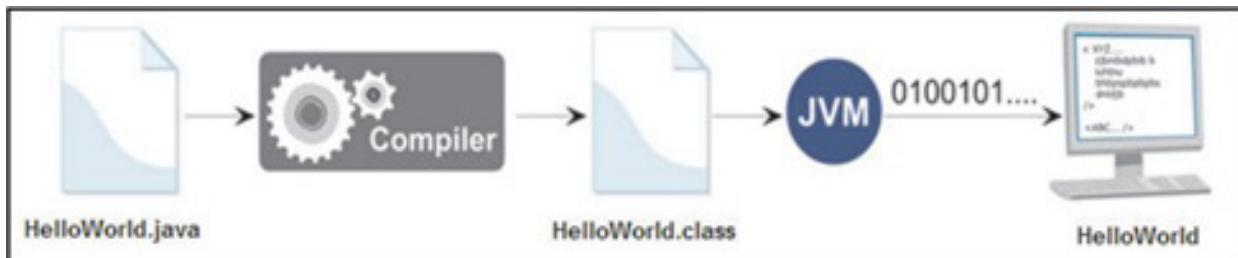


Figure 1.14: Compilation of Java Program

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



Figure 1.15: 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 H:\Java.

To change the directory path, perform following steps:

- Type the command **H:** and press **Enter**.
- Type **cd Java** 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 1.16 shows the commands on the **Command Prompt** window for changing the directory path to H:\Java.

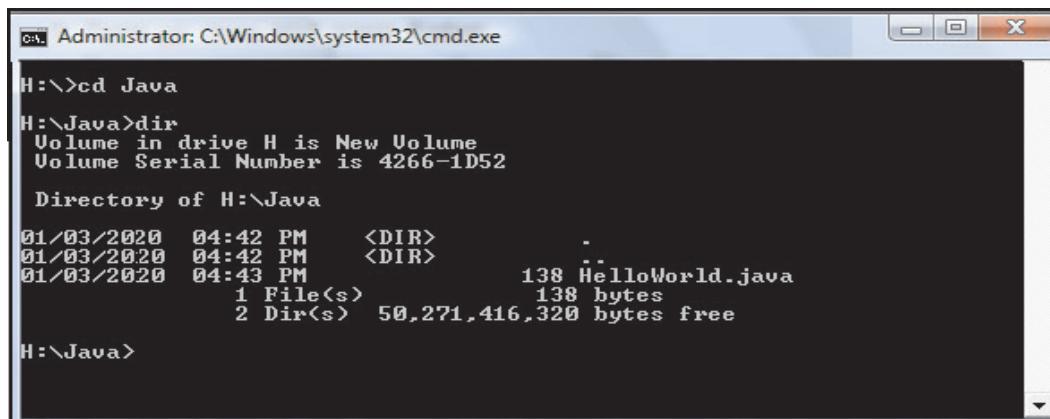


Figure 1.16: Command Prompt – Change Drive and Directory

As shown in figure 1.16, 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 1.5 lists some of the options that can be used with the `javac` command.

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

Table 1.5: 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.

1.11.3 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 1.17 shows the components of JVM involved in the execution of the compiled bytecode.

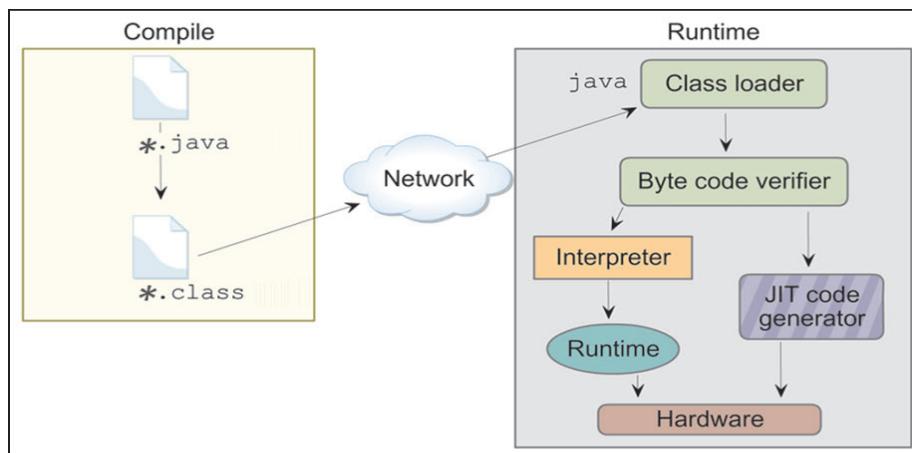


Figure 1.17: Components of JVM

As shown in figure 1.17, 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 boosts 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 1.18 shows the output of the Java program on the **Command Prompt** window.

```

Administrator: C:\Windows\system32\cmd.exe
H:\Java>javac HelloWorld.java
H:\Java>dir
Volume in drive H is New Volume
Volume Serial Number is 4266-1D52
Directory of H:\Java
01/04/2020  10:18 AM    <DIR>
01/04/2020  10:18 AM    <DIR>
01/04/2020  10:19 AM                442 HelloWorld.class
01/03/2020  04:43 PM                138 HelloWorld.java
                           2 File(s)   580 bytes
                           2 Dir(s)  50,267,156,480 bytes free
H:\Java>java HelloWorld
Welcome to the world of Java

```

Figure 1.18: Output of HelloWorld Program

Table 1.6 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 <code>CLASSPATH</code> 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 1.6: Options for `java` Command

1.12 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 graphical user interface for building, compiling, debugging, and packaging of applications.
- Provides simple and user-friendly IDE configuration.

1.12.1 Elements of NetBeans IDE

The NetBeans IDE has following elements and views:

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

Apache NetBeans 12.0 adds support for latest Java language features and integrates new code donations from Oracle for Java Web development. 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.

It also introduces new out of the box templates for JavaFX, upgrades and extends its PHP editor, provides new dark look and feel options, and incorporates a wide range of fixes.

All features provided by Apache NetBeans are free, supported out of the box, without requiring additional plugins to be installed. Apache NetBeans 12.0 is the first Long-Term Support (LTS) release of NetBeans as a top level Apache project.

Figure 1.19 shows various elements in the NetBeans IDE 12.0.

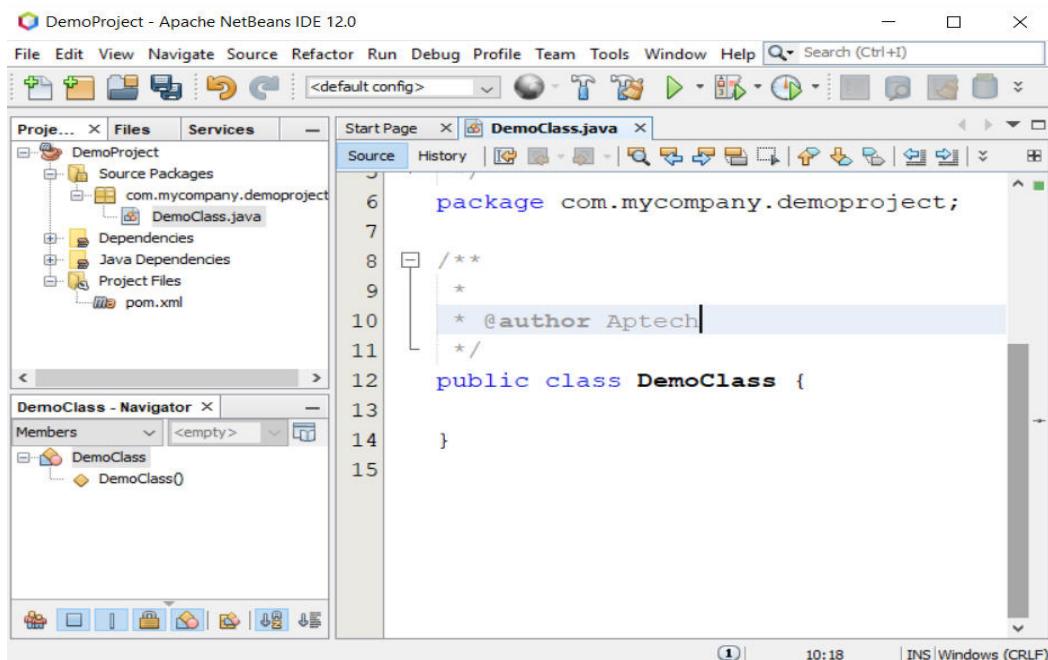


Figure 1.19: Elements in NetBeans IDE

As shown in figure 1.19, 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 1.20 shows the menu bar in the NetBeans IDE.

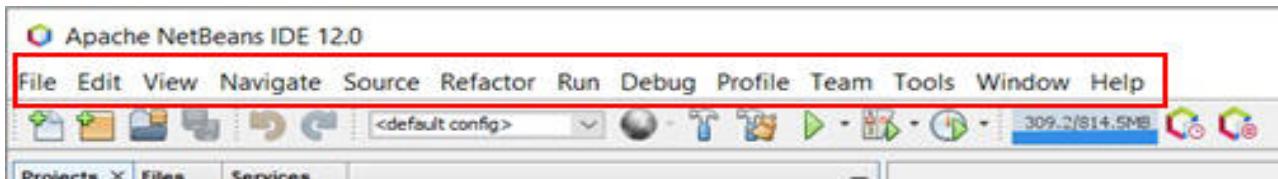


Figure 1.20: Menu Bar in NetBeans IDE

Table 1.6 lists and explains different menus in the NetBeans IDE.

Menu	Description
File	Displays commands for using the project and the NetBeans IDE

Menu	Description
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
Run	Compiles source files, executes the application, and generate packaged build output, such as JAR or 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 1.6: 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 1.21 shows the folder view in the NetBeans IDE.

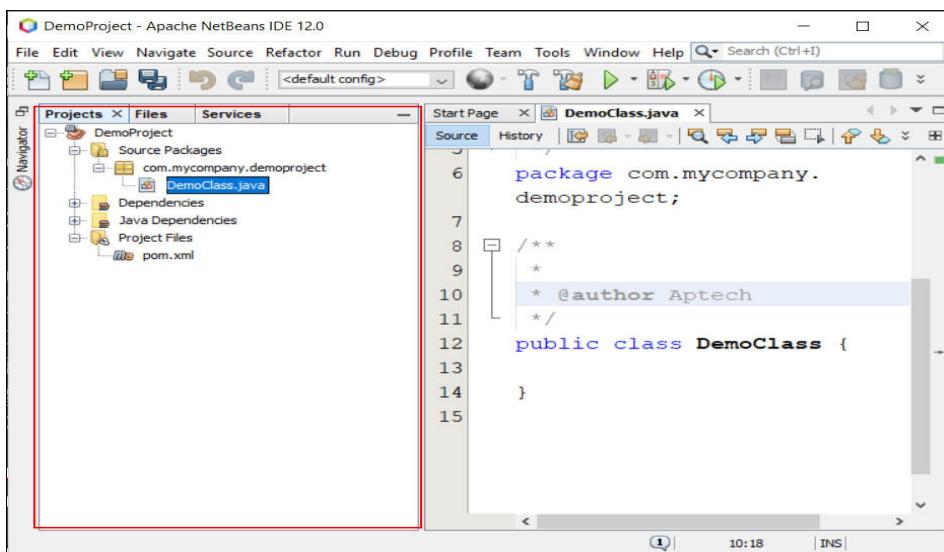


Figure 1.21: Folder View in NetBeans IDE

Table 1.7 lists and describes the 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 1.7: 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 1.22 shows the **Navigator** window in the NetBeans IDE.

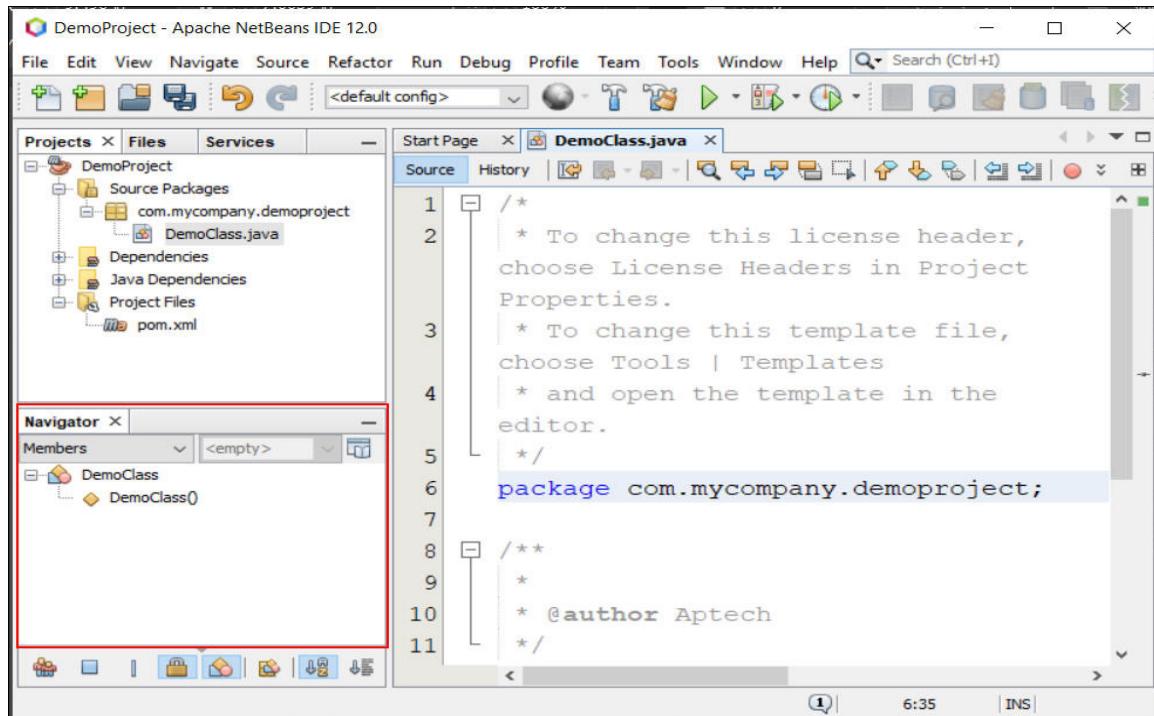


Figure 1.22: Navigator Window

→ Code View

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

Figure 1.23 shows the **Source Editor** used for viewing the code.

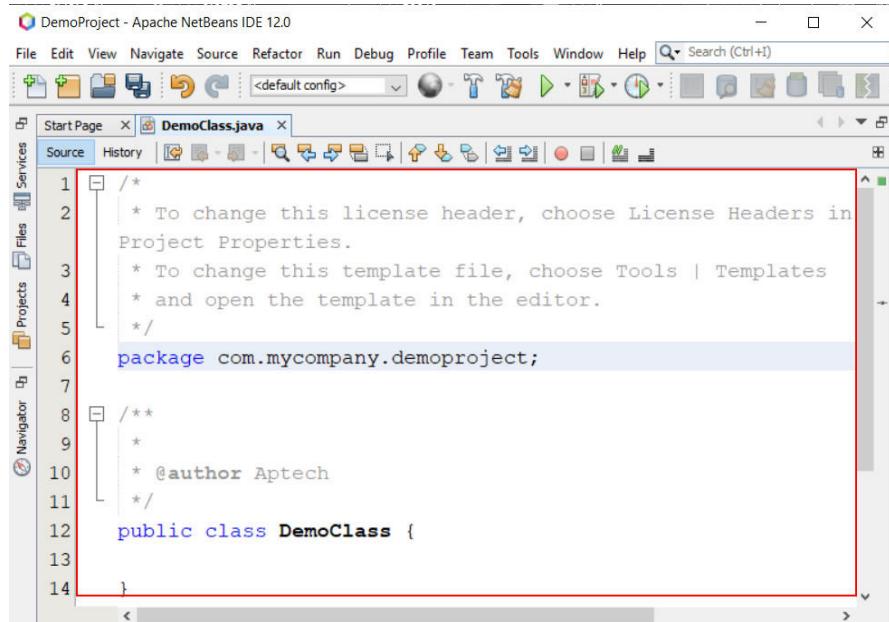


Figure 1.23: Source Editor

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 1.24 shows the **Output** window.

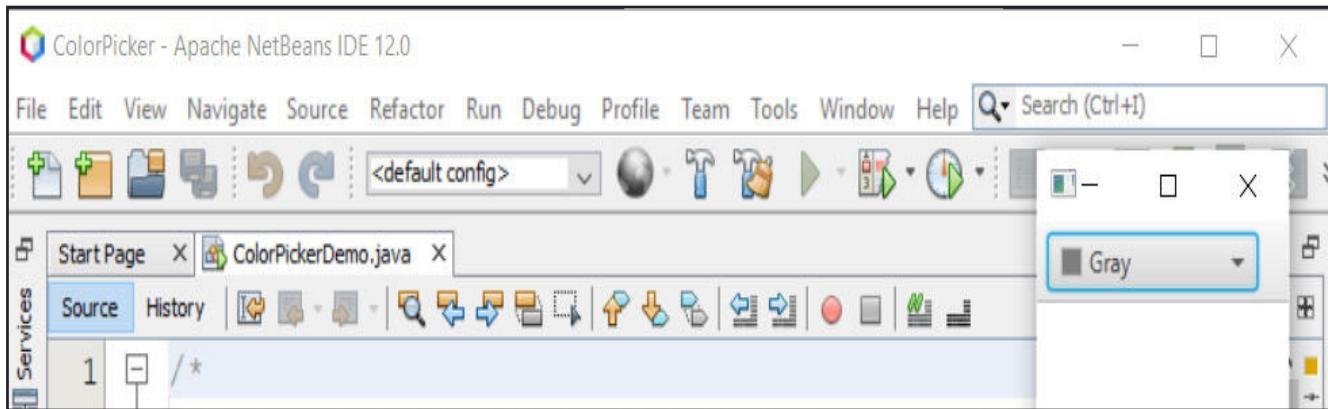


Figure 1.24: Output Window

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 1.25.



Figure 1.25: Download Option

- **Step 2:** Select appropriate binary distribution as shown in figure 1.26.

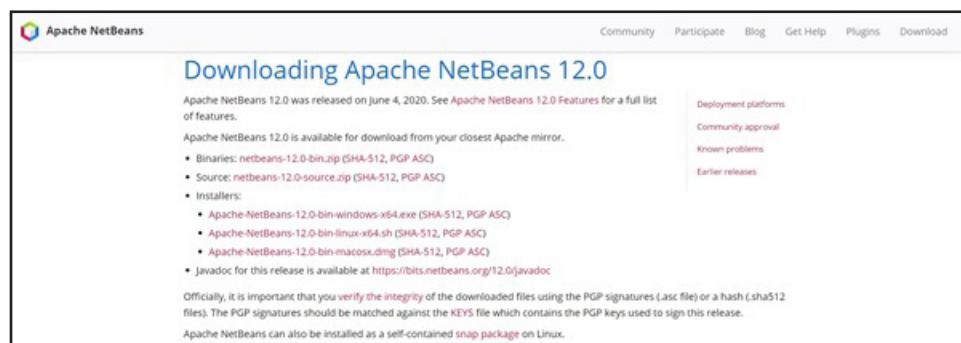


Figure 1.26: Selecting Binary Distribution

- **Step 3:** Select mirror to download installer as shown in figure 1.27.

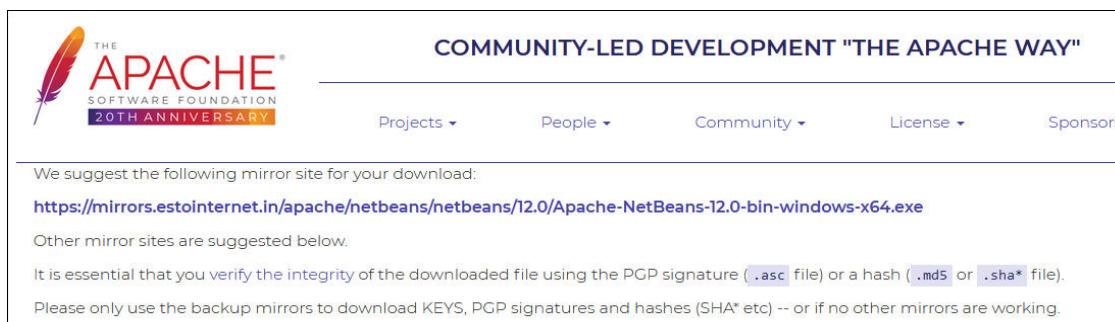


Figure 1.27: Selecting Mirror

- **Step 4:** Select and install downloaded file by following instructions provided by the installer, one by one. Refer to figure 1.28 for the downloaded file.

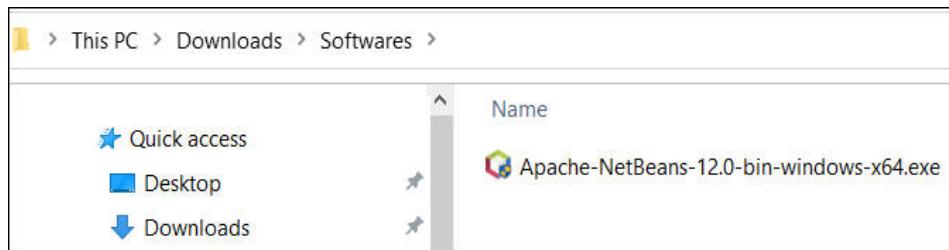


Figure 1.28: Downloaded File

1.12.2 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 1.29 shows the **Choose Project** page in the wizard.

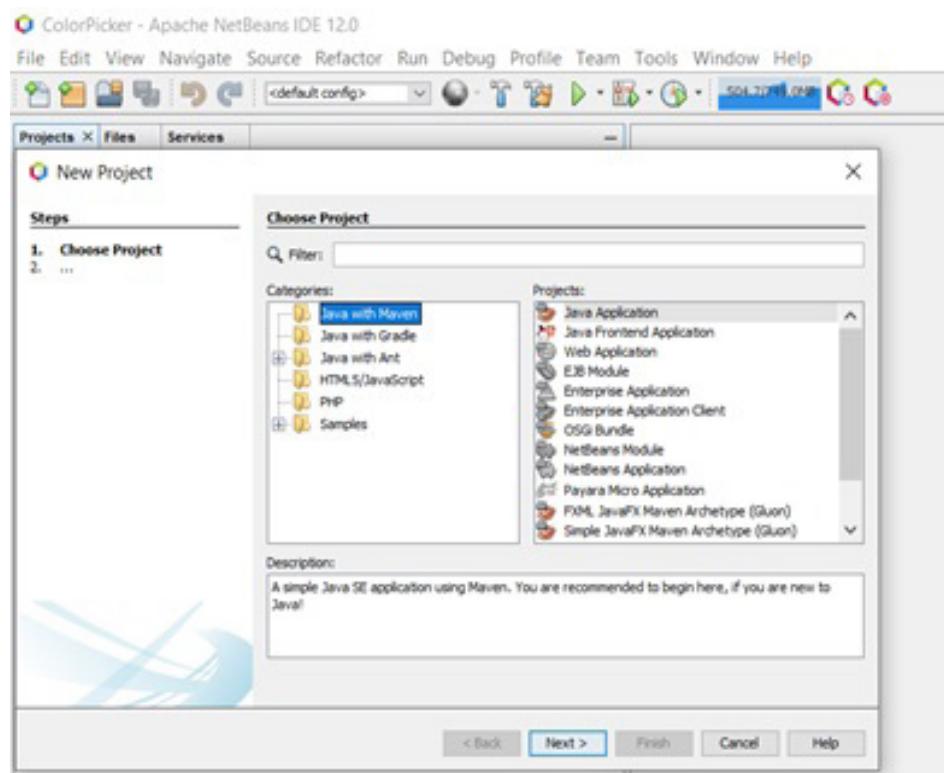


Figure 1.29: New Project Wizard – Choose Project Page

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.
6. Click **Finish**.

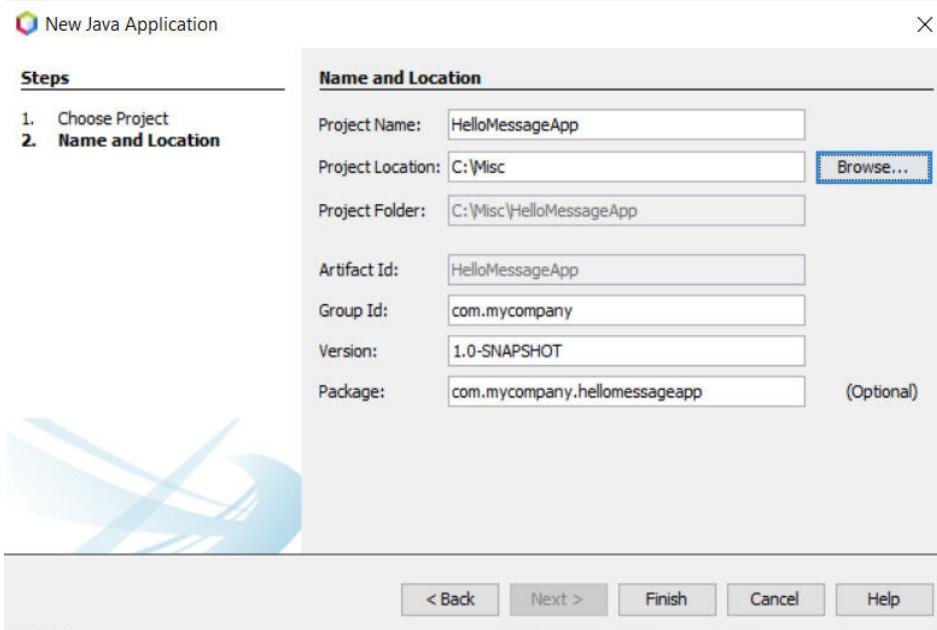


Figure 1.30: New Project Wizard – Name and Location Page

Figure 1.31 shows the project **HelloMessageApp** with class **HelloMessageApp** and auto-generated code.

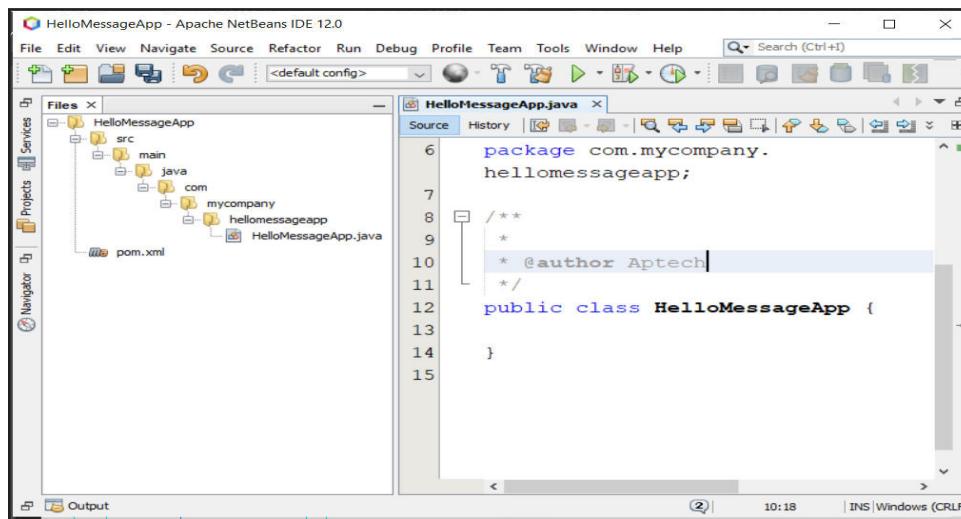


Figure 1.31: NetBeans IDE – HelloMessageApp

Next, modify code by adding following code statements inside the class and then, **Build** and **Run** the project to see the output as shown in figure 1.32.



Figure 1.32: Output Window - HelloMessageApp

1.13 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.

1.13.1 Single-line Comments

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

Figure 1.33 shows the `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 1.33: 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
...
```

1.13.2 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;
...
```

1.13.3 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;
```

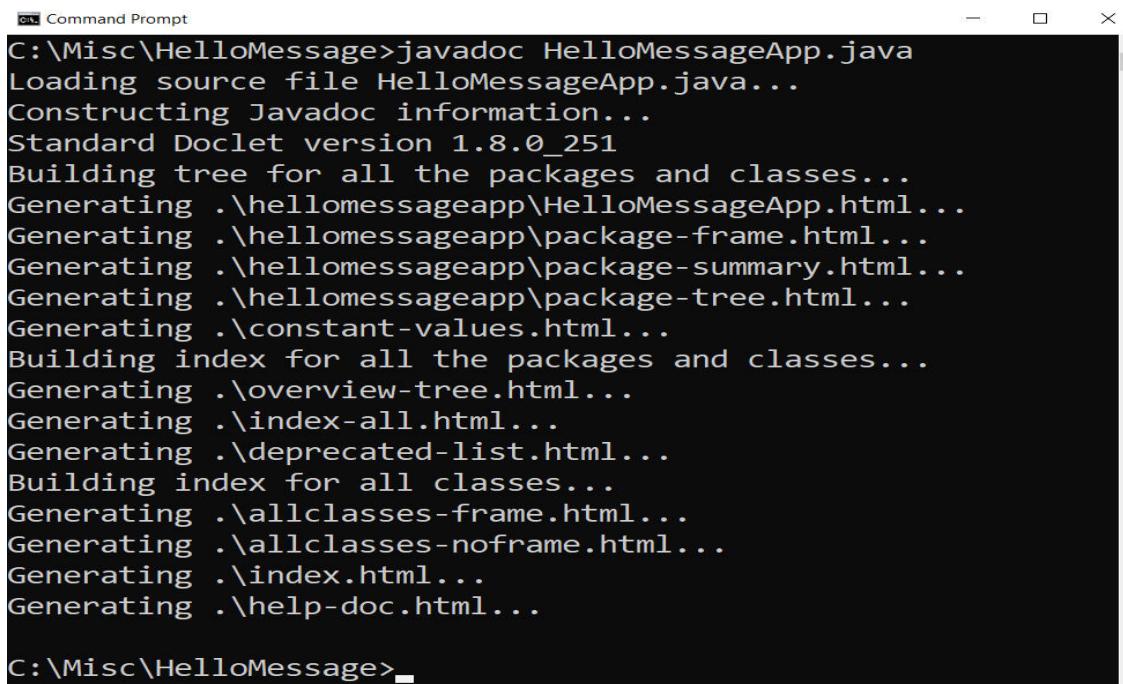
```

/**
 *
 * @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 1.34, an HTML file will be created showing documentation for the class.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The path "C:\Misc\HelloMessage>" is visible at the top. The window displays the output of the `javadoc` command being run on the file `HelloMessageApp.java`. The output shows the process of generating JavaDoc files, including the creation of a package frame, package summary, package tree, constant values, overview tree, index-all, deprecated list, all classes frame, all classes noframe, index, and help doc files. The command prompt ends with a cursor at the bottom.

```

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 1.34: javadoc Command

1.14 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

1.14.1 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 15.
- Apache NetBeans IDE version 12 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.

```
  package com.apttech;
  class Main {
    public static void main(String[] args) {
      System.out.println("Hello Java");
    }
  }
```

Try it Yourself

1. Check the directory structure of JDK 15 on your local system. Also, identify and list various tools provided by JDK 15. 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
- ➔ State the syntax of variable declaration
- ➔ Explain the rules and conventions for naming variables
- ➔ Explain data types
- ➔ Describe primitive and reference data types
- ➔ Describe escape sequence
- ➔ Describe 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.

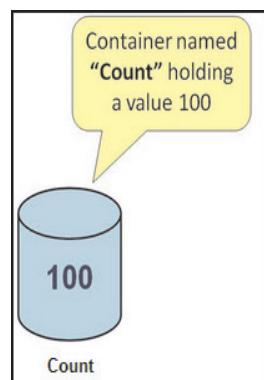


Figure 2.1: Variable

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

Variable Name	Valid/Invalid
_2010_tax	Valid
\$_\$	Valid
amount#Balance	Invalid and contains the illegal character #
double	Invalid and is a keyword
4short	Invalid and the first character is a digit

Table 2.1: Valid and Invalid 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.

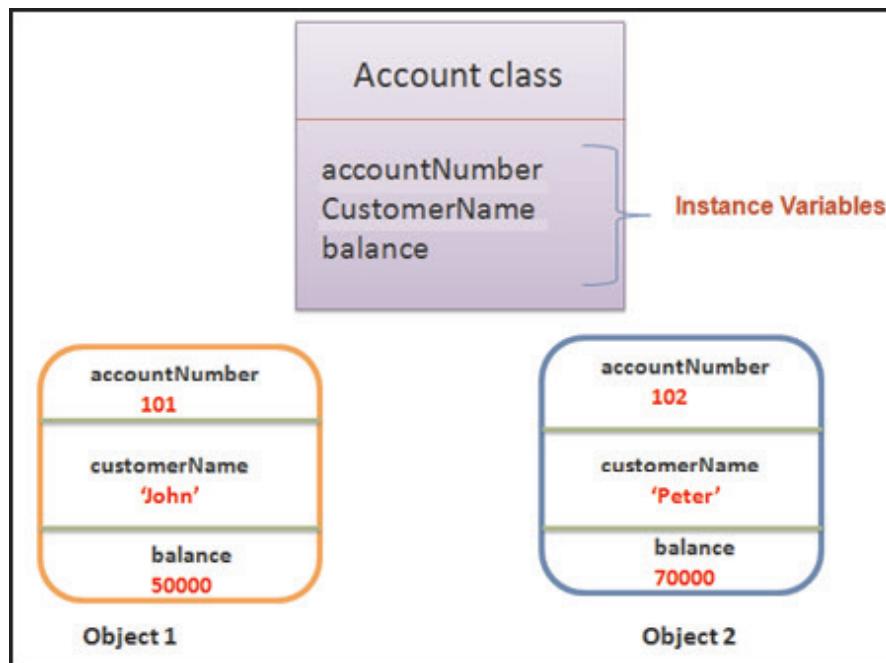


Figure 2.2: Instance Variables

- **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.

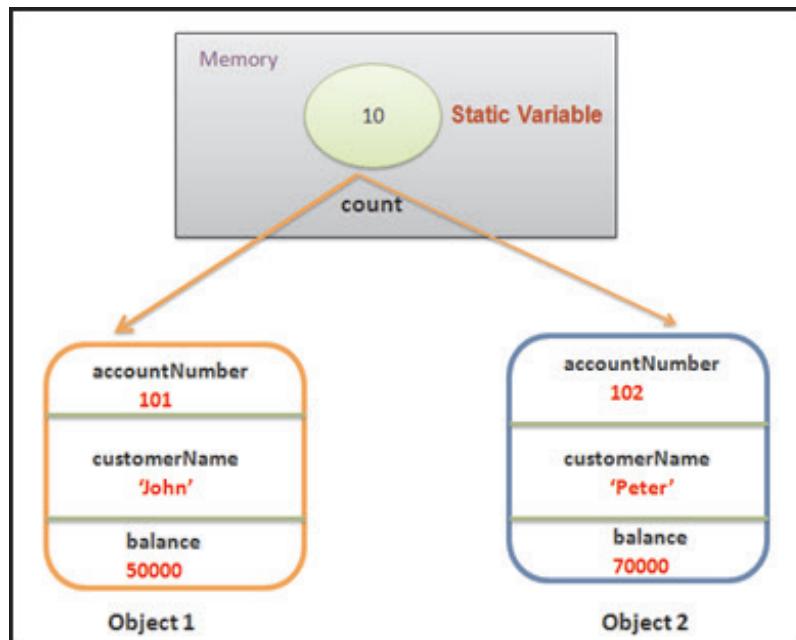


Figure 2.3: Static Variables

- **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.

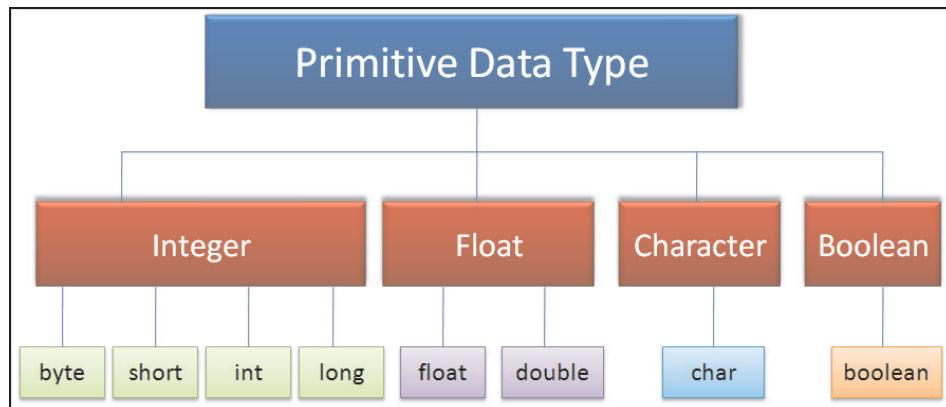


Figure 2.5: Primitive Data Types

2.3.2 Integer Types

The integer data types supported by Java are `byte`, `short`, `int`, and `long`. These data type can store signed integer values. Signed integers are those integers, which are capable of representing positive as well as negative numbers, such as -40.

Languages, such as C or C++, support signed and unsigned integers, but Java does not provide support for unsigned integers.

The brief description of the integer data types are as follows:

- **byte** - Used to store small amount of data. This is a signed 8-bit type and can store values ranging from -128 to 127. It is useful when working with a stream of data from a network or file. This data type is also useful when working with raw binary data that may not be compatible with Java's other built-in types. `byte` keyword is used to declare a byte variable. For example, `byte val;`
- **short** - This data type is least used. It is a signed 16-bit type and can store values ranging from -32,768 to 32,767. This type is mostly applicable to 16-bit computers. This data type is used to store smaller numbers, for example, employee number. `short` keyword is used to declare a short variable.
- **int** – This is the most commonly used data type for storing whole numbers. It is a signed 32-bit type, and can store values ranging from -2,147,483,648 to 2,147,483,647. It is the most versatile and efficient data type. It can be used to store the total salary being paid to all the employees of the company. `int` keyword is used to declare an integer variable.
- **long** - It is a signed 64-bit type and can store values ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. It is useful when an `int` type is not large enough to hold the desired value. The range of value a `long` data type can store is quite large for example, population of a country. `long` keyword is used to declare long variables for storing large numeric values.

2.3.3 Floating-point Types

The floating-point data types supported by Java are `float` and `double`. These are also called real numbers, as they represent numbers with fractional precision. For example, calculation of a square root or `PI` value is represented with a fractional part. The brief description of the floating-point data types is as follows:

- **`float`** – This type supports a single precision value with 32-bit storage. They are useful when a number requires a fractional component, but with less precision. For example, the representation of money in dollars and cents are required to be in floating-point values. The `float` keyword is used to store float values. For example, `float squRoot, cubeRoot;`
- **`double`** - This type supports a double precision with 64-bit storage. The double precision is faster on processors that are optimized for mathematical calculations. This data type is useful when accuracy is required to be maintained while performing calculations. The `double` keyword is used to store large-valued numbers. For example, `double bigDecimal;`

2.3.4 Character Type

`char` data type belongs to this group and represents symbols in a character set such as letters and numbers. `char` data type stores 16-bit Unicode character and its value ranges from 0 ('\u0000') to 65,535 ('\uffff').

Unicode is a 16-bit character set, which contains all the characters commonly used in information processing. It is an attempt to consolidate the alphabets of the world's various languages into a single and international character set.

2.3.5 Boolean Type

`boolean` data type belongs to this group and represents `true` or `false` values. This data type is used to track `true/false` conditions. Its size is not defined precisely.

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 an `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 commandline 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

2.3.6 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.

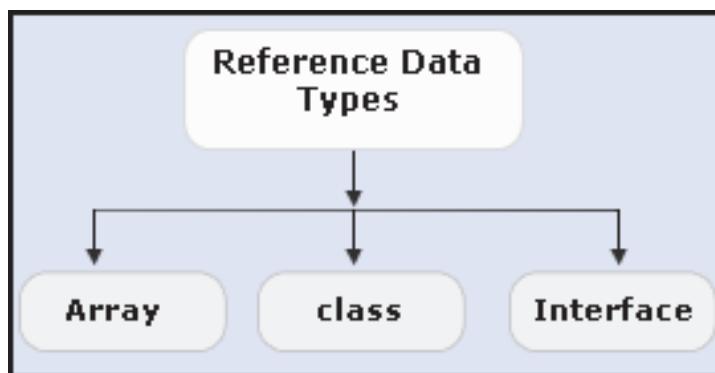


Figure 2.7: Reference Data Types

Table 2.2 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 2.2: 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 and consist of numbers 0 and 1. Java SE 7 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 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 2.3 list 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 2.3: 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 2.4 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 2.4: 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.

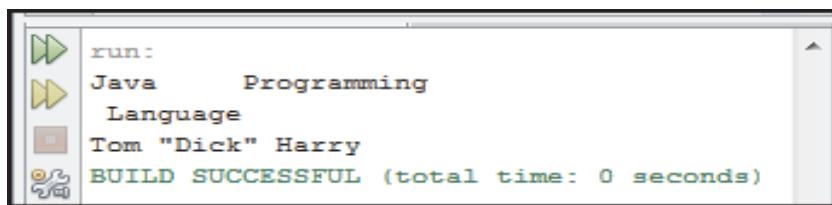


Figure 2.9: Output of Escape Sequences

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.

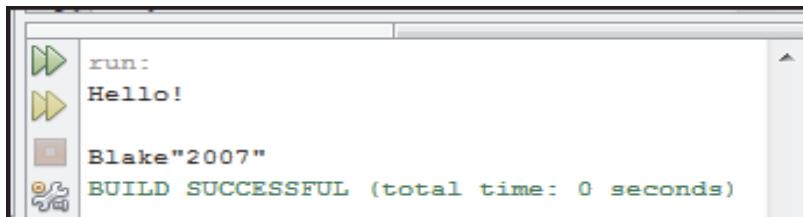


Figure 2.10: Output of Unicode Sequence

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.

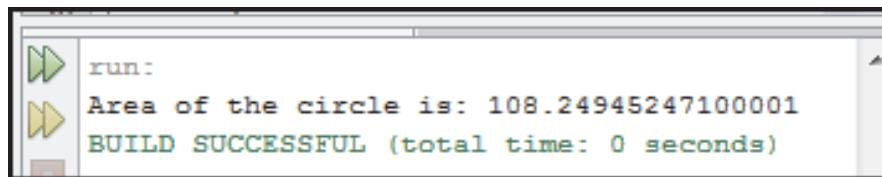


Figure 2.11: Output of Constant Variable

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.

Java SE 5.0 introduced enumerations. An enumeration is defined as a list that contains constants. In previous languages, like 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.

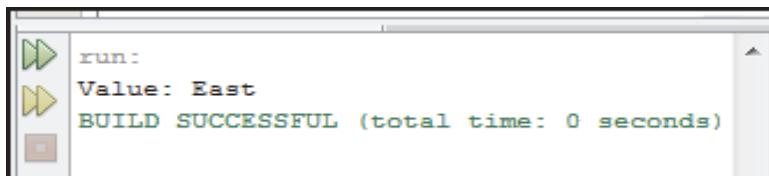


Figure 2.12: Output of Enumeration Variable

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.

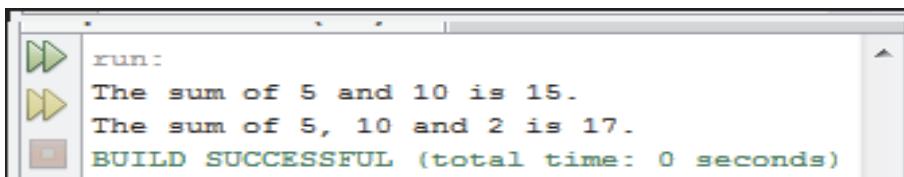


Figure 2.13: Output of `print()` and `println()` Methods

2.7.2 `printf()` Method

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

Table 2.5 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 2.5: 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.

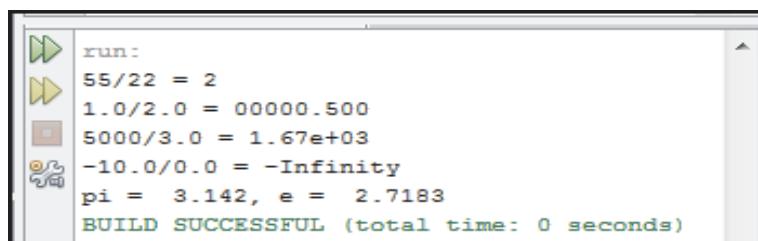


Figure 2.14: Output of Format Specifiers

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 2.6 lists some of the flags available in Java.

Flag	Description
"_"	Left justify the argument

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

Table 2.6: 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.

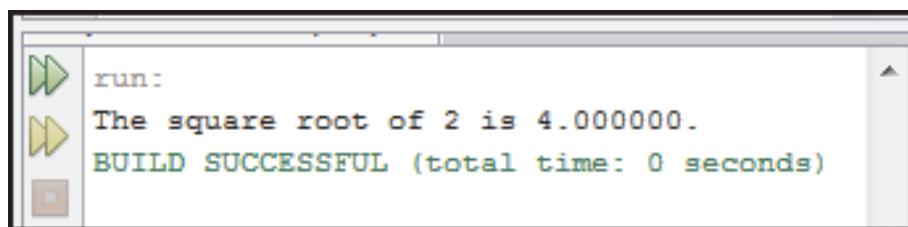
The values within '[]' are optional. The only required elements of format specifier are the % and a conversion character.

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 2.7 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 2.7: 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 and they are as follows:

- Assignment Operator
- Arithmetic Operator
- Unary Operator
- Conditional Operator

- Logical Operator
- Assignment Operator
- Bitwise Operator

2.8.1 Assignment Operators

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

Consider following statements:

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

The value 3456 and 'M' are assigned to the variables, **balance** and **gender**.

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

```
X = 3;
```

```
X += 5;
```

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

Code Snippet 15 demonstrates the use of assignment operators.

Code Snippet 15:

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

2.8.2 Arithmetic Operators

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

Table 2.8 lists the arithmetic operators.

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

Table 2.8: Arithmetic Operator

Code Snippet 16 demonstrates the use of arithmetic operators.

Code Snippet 16:

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

2.8.3 Unary Operator

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

Table 2.9 lists the unary operators.

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

Table 2.9: Unary Operator

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

Code Snippet 17 demonstrates the use of unary operators.

Code Snippet 17:

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

2.8.4 Conditional Operators

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

Table 2.10 lists various conditional operators.

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

Table 2.10: Conditional Operators

Code Snippet 18 demonstrates the use of conditional operators.

Code Snippet 18:

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

```

int value1=10;
int value2=20;
// Use of conditional operators
System.out.print("value1==value2: ");
System.out.println(value1==value2);
System.out.print("value1 !=value2: ");
System.out.println(value1 !=value2);
System.out.print("value1>value2: ");
System.out.println(value1>value2);
System.out.print("value1<value2: ");
System.out.println(value1<value2);
System.out.print("value1<=value2: ");
System.out.println(value1<=value2);

}
}

```

The output of the code is shown in Figure 2.17.

Figure 2.17: Output of Conditional Operators

2.8.5 Logical Operators

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

Table 2.11 lists the two logical operators.

Operator	Description
<code>&&</code>	Conditional AND - Returns true only if both the expressions are true
<code> </code>	Conditional OR - Returns true if either of the expression is true or both the expressions are true

Table 2.11: Logical Operators

Code Snippet 19 demonstrates the use of logical operators.

Code Snippet 19:

```
public class TestLogicalOperators {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int first=10;
        int second=20;
        // Use of logical operator
        System.out.println((first==30) && (second==20));
        System.out.println((first==30) || (second==20));
    }
}
```

The output of the code is shown in Figure 2.18.

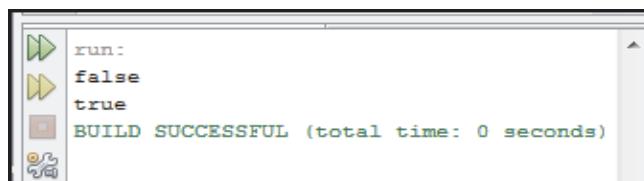


Figure 2.18: Output of Logical Operators

2.8.6 Bitwise Operators

Bitwise operators work on binary representations of data. These operators are used to change individual bits in an operand.

Table 2.12 lists various bitwise operators.

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

Operator	Description
<<	Shift Left operator - Moves all the bits in a number to the left by the specified position

Table 2.12: Bitwise Operators

Code Snippet 20 demonstrates the use of bitwise operators.

Code Snippet 20:

```
public class TestBitwiseOperators {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int x=23;
        int y=12;
        // 23=10111, 12=01100
        System.out.print("x & y: ");
        System.out.println(x&y); // Returns 4, i.e., 4=00100
        System.out.print("x | y: ");
        System.out.println(x|y); // Returns 31, i.e 31=11111
        System.out.print("x ^ y: ");
        System.out.println(x^y); // Returns 27, i.e 31=11011
        int a=43;
        int b=1;
        System.out.print("a >> b: ");
        System.out.println(a>>b); // returns 21, i.e, 21=0010101
        System.out.print("a << b: ");
        System.out.println(a<<b); // returns 86, i.e, 86=1010110
    }
}
```

The output of the code is shown in Figure 2.19.

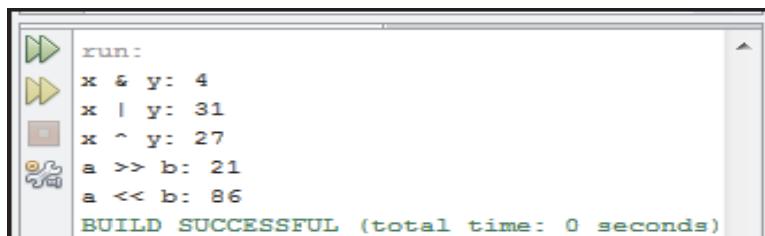


Figure 2.19: Output of Bitwise Operators

2.8.7 Ternary Operator

The ternary operator (`? :`) is a shorthand operator for an `if-else` statement. It makes your code compact and more readable.

The syntax to use the ternary operator is as follows:

Syntax:

```
expression1 ? expression2 : expression3
```

where,

`expression1`: Represents an expression that evaluates to a boolean value of `true` or `false`.

`expression2`: Is executed if `expression1` evaluates to `true`.

`expression3`: Is executed if `expression1` evaluates to `false`.

Code Snippet 21 demonstrates the use of ternary operator.

Code Snippet 21:

```
public class VariableScope {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int value1 = 10;
        int value2 = 20;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;
        System.out.println(result);
    }
}
```

As `someCondition` variable evaluates to `true`, the value of `value1` variable is assigned to the `result` variable. Thus, the program prints 10 on the console.

2.8.8 Operator Precedence

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

Table 2.13 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 2.13: 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`

2.8.9 Operator Associativity

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

Table 2.14 shows Java operators and their associativity.

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

Table 2.14: 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.

$2+10+4-5*6$

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

$2+10+4-30$

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

$12+4-30$

- 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 2.20 shows the implicit type casting.

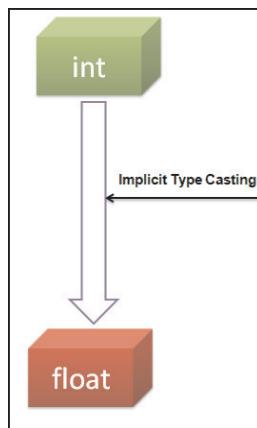


Figure 2.20: Implicit Type Casting

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 22 demonstrates implicit type conversion.

Code Snippet 22:

```
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 2.21 shows the explicit type casting of data types.

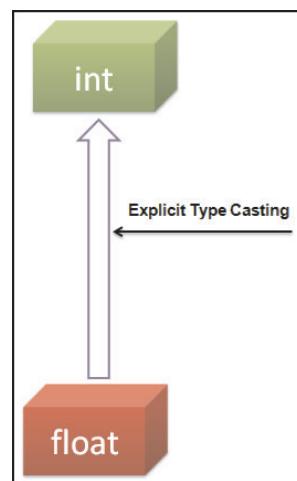


Figure 2.21: Explicit Type Casting

Code Snippet 23 adds a `float` value to an `int` and stores the result as an integer.

Code Snippet 23:

```
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 = 0.00000000000053; 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

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. Java SE 7 supports the use of the underscore characters (_) between the digits of a numeric literal.
- 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.

```
1. public class Test {  
2.     public static void main()  
3.         String pac
```

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);  
    }  
}
```

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 the different types of loops
- Explain the while statement and the associated rules
- Identify the purpose of the do-while statement
- State the need of for statement
- Describe nested loops
- Compare the different types of loops
- State 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.

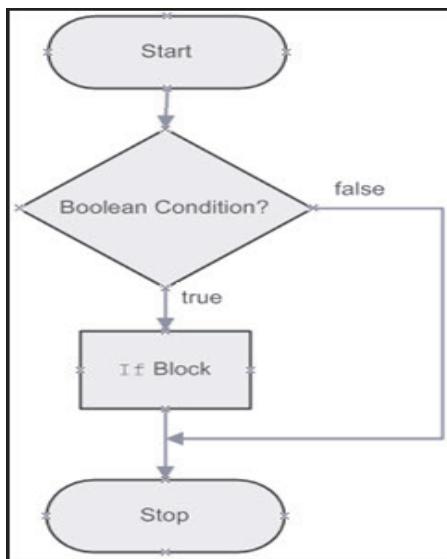


Figure 3.1: Flow of Execution – if Statement

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 commandline 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.

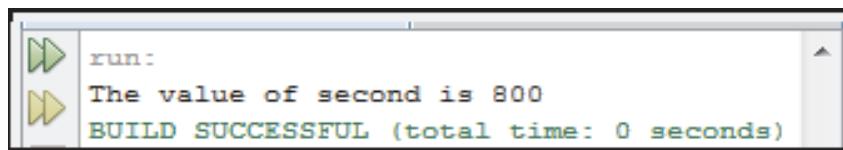


Figure 3.2: Output of Code With Simple if Statement

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.

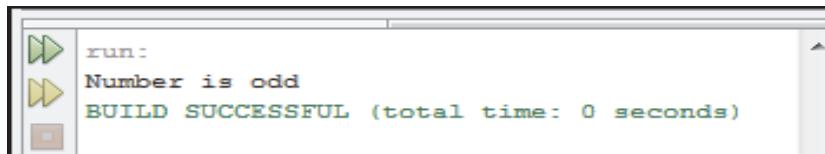


Figure 3.3: Output of Code With if-else Statement

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 commandline 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.

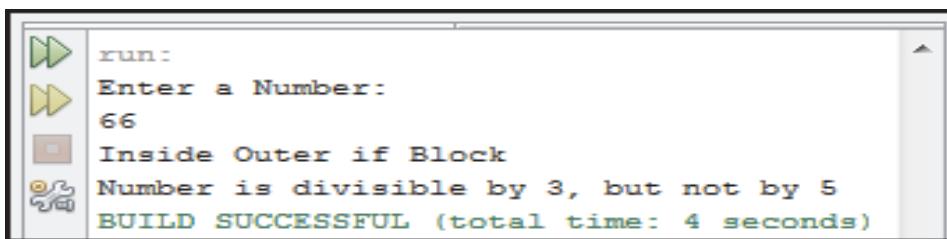


Figure 3.4: Output of Code With Nested-if Statement

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

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

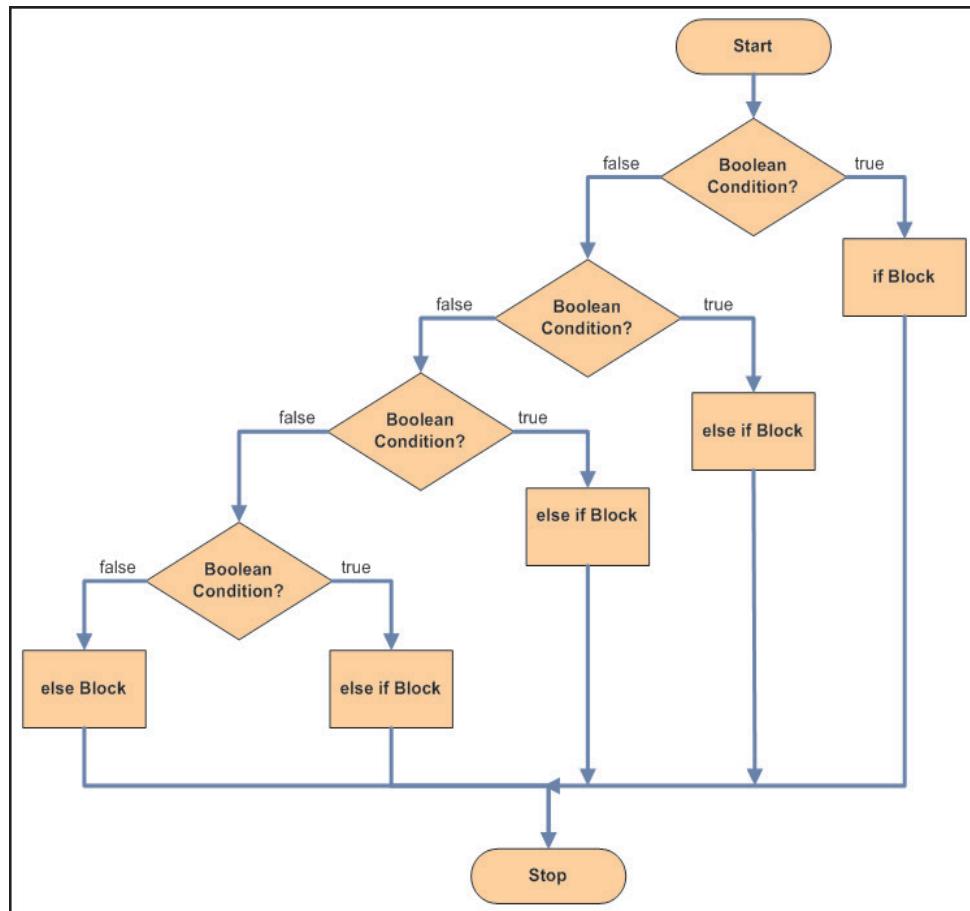


Figure 3.5: Flow of Execution - if-else-if Ladder

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.

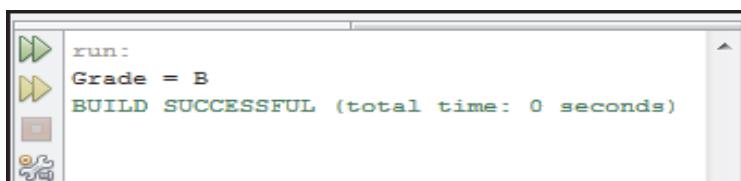


Figure 3.6: Output of Code With `if-else-if Ladder`

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`.

From Java SE 7 onwards, the `switch-case` statement also supports the use of strings and can be passed as an expression for the `switch` statement. The `String` class is used for creating string values in the Java program.

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.

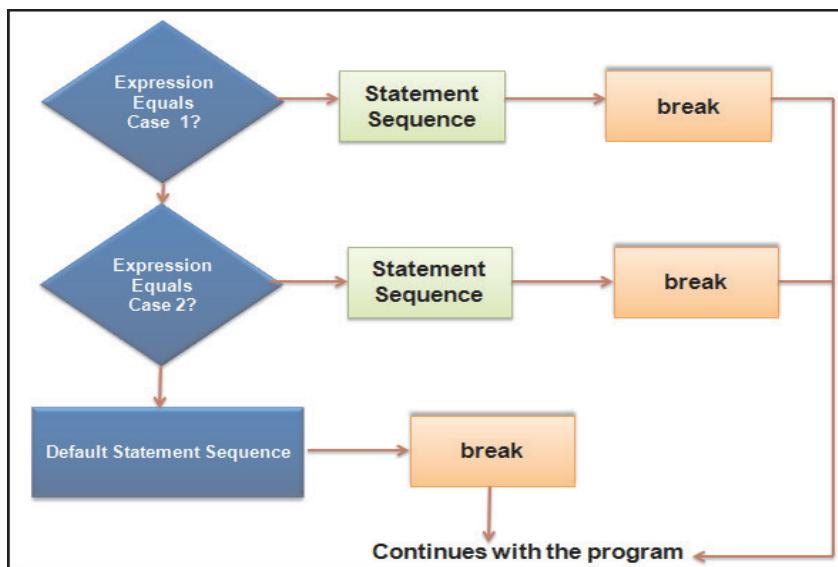


Figure 3.7: switch-case Statement

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 commandline 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.

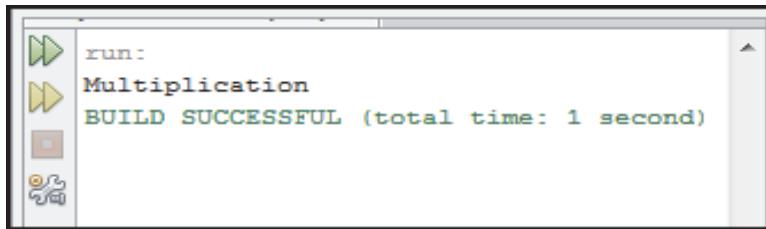


Figure 3.8: Output of Code With Simple switch-case Statement

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("Invalid Month");  
        }  
    }  
}
```

```

} // 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.

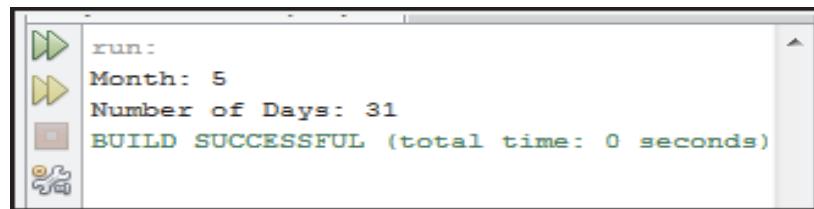


Figure 3.9: Output of Code With Multiple Cases Without `break` Statement

3.3.1 String-based `switch-case` Statement

Java SE 7 supports the use of strings in the `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.

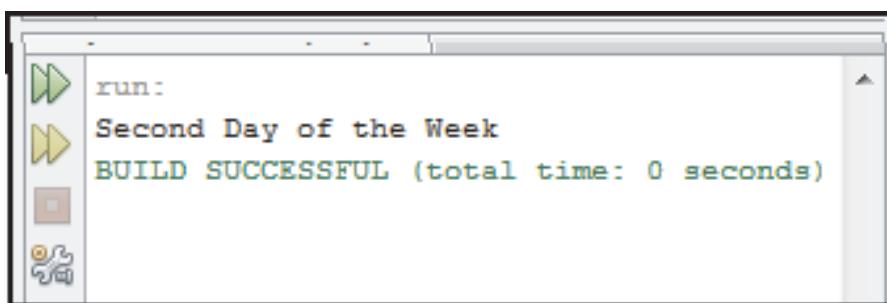


Figure 3.10: Output of Code With `switch-case` Statement Having String Expression

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;
    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.

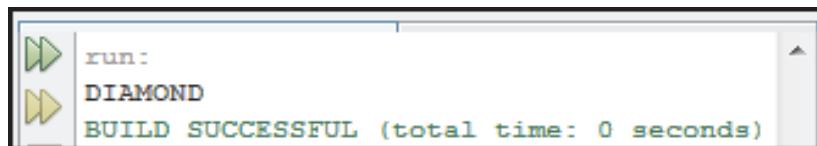


Figure 3.11: Output of Code With switch-case Statement Having Enumeration

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 commandline 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.

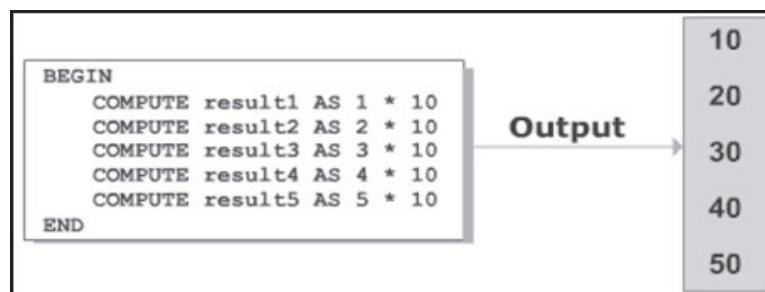


Figure 3.13: Pseudocode – Multiples of 10

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.

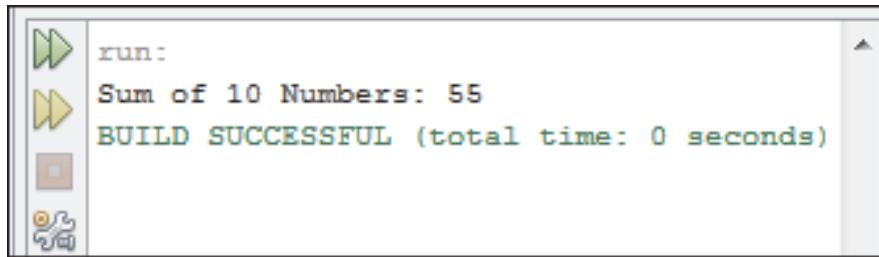


Figure 3.15: Output of Code With `do-while` Loop

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.

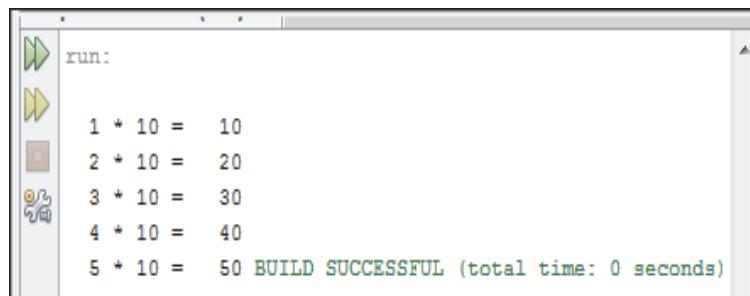


Figure 3.16: Output of Code With Simple `for` Loop

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 3.18.

```

0 + 10 = 10
1 + 9 = 10
2 + 8 = 10
3 + 7 = 10
4 + 6 = 10
5 + 5 = 10
6 + 4 = 10
7 + 3 = 10
8 + 2 = 10
9 + 1 = 10
10 + 0 = 10BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 3.18: 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 3.19.

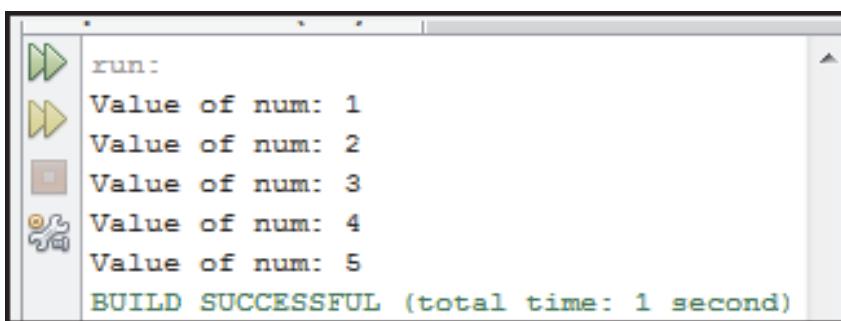


Figure 3.19: Output of Code With `for` Loop with Empty Initialization Section

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

Java SE 5 onwards supports enhanced `for` loop to increase the readability of the loop. This version of the `for` loop implements a ‘`for-each`’ style loop. It is also referred to as 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.

Code Snippet 19 demonstrates the use of a nested-for loop for displaying a pattern.

Code Snippet 19:

```
public class DisplayPattern {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int row, col;
        // The outer for loop executes 5 times
        for (row=1; row<=5; row++) {
            /*
             * For each iteration, the inner for loop will execute from col=1
             * and will continue, till the value of col is less than or equal to row
             */
            for (col=1; col<=row; col++) {
                System.out.print(" * ");
            }
            System.out.println();
        }
    }
}
```

```

    } // End of inner for loop
    System.out.println();
} // End of outer for loop
}
}

```

In Code Snippet 19, the outer `for` loop starts with the counter variable `row` whose initial value is set to 1. As the condition, `row < 5` is evaluated to `true`, the body of the outer `for` loop gets executed. The body contains an inner `for` loop which starts with the counter variable's value `col1` set to 1. The iteration of the inner loop executes till the value of `col1` is less than or equal to the value of `row` variable. Once the value of `col1` is greater than `row`, the inner `for` loop terminates.

Then, the control moves to the outer `for` loop and the value of the variable `row` is incremented by one. The outer loop continues, till the value of `row` is not greater than five. For each iteration of the outer loop, the inner `for` loop is reinitialized and continues till the condition evaluates to `false`.

The output of the code is shown in figure 3.20.

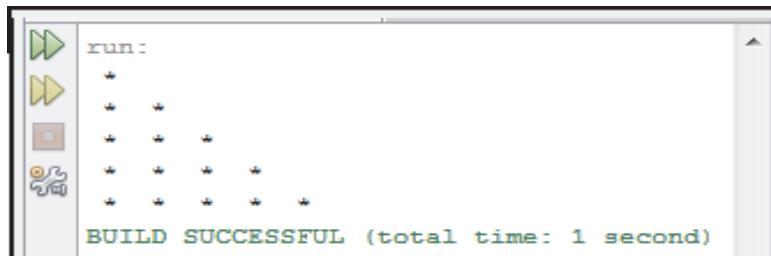


Figure 3.20: Output of Code With Nested-for Loop

3.10 Comparison of Loops

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

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

Table 3.3 lists the differences between `while/for` and `do-while` loops.

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

Table 3.3: Difference Between `while/for` and `do-while`

3.11 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.

3.11.1 `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 20 demonstrates the use of `break` statement.

Code Snippet 20:

```
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 20, 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 3.21 with different values entered by the user.

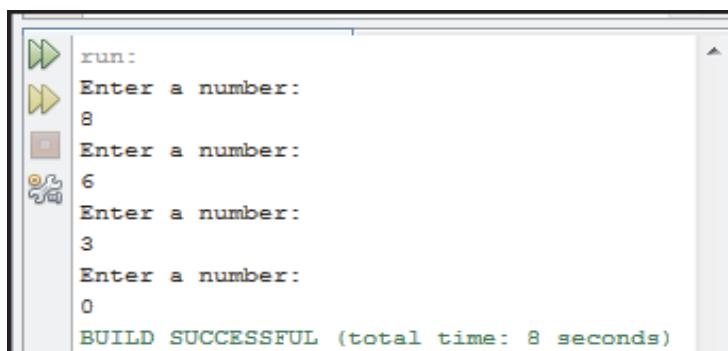


Figure 3.21: Output of Code Using `break` Statement

3.11.2 *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 21 demonstrates the code that uses `continue` statement in printing the square and cube root of a number.

Code Snippet 21:

```

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 21 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 21 is shown in figure 3.22.

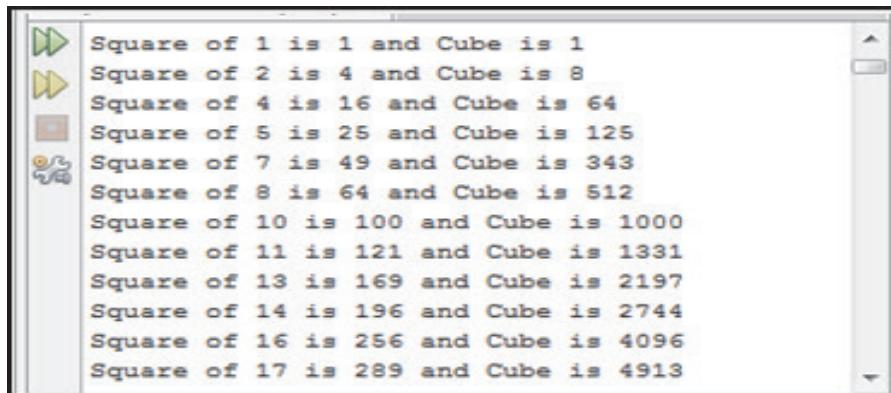


Figure 3.22: Output of Code Using `continue` Statement

3.11.3 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 22 demonstrates the use of labeled `break` statement.

Code Snippet 22:

```
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 22, 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 3.23.

```
run:
This is the outer loop.
This is the outer loop.
Hello
Good - Bye
BUILD SUCCESSFUL (total time: 1 second)
```

Figure 3.23: Output of Code Using Labeled break Statement

Similar to labeled `break` statement, you can specify a label to enclose a loop that continues with the next iteration of the loop.

Code Snippet 23 demonstrates the use of labeled `continue` statement.

Code Snippet 23:

```
public class NumberPyramid {

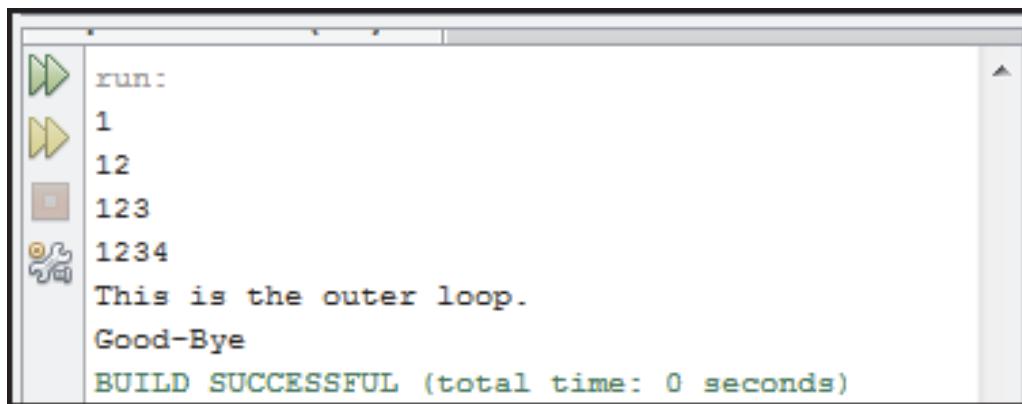
    /**
     * @param args the command line arguments
     */

    public static void main(String[] args) {
        outer:
        for (int i = 1; i < 5; i++) {
            for (int j = 1; j < 5; j++) {
                if (j > i) {
                    System.out.println();
                    /* Terminates the loop counting j and continues the
                     * next iteration of the loop counting i
                     */
                    continue outer;
                } // End of if statement
                System.out.print(j);
            } // End of inner for loop
            System.out.println("\nThis is the outer loop.");
        } // End of outer for loop
        System.out.println("Good-Bye");
    }
}
```

```
}
```

```
}
```

The output of the code is shown in figure 3.24.



A screenshot of a code editor window titled "run:". The code is as follows:

```
run:  
1  
12  
123  
1234  
This is the outer loop.  
Good-Bye  
BUILD SUCCESSFUL (total time: 0 seconds)
```

The code uses a labeled `continue` statement to skip the current iteration of the inner loop when the value of `i` is 3. The output shows the numbers 1, 2, and 4 being printed, followed by a message indicating the outer loop has completed, and finally "Good-Bye".

Figure 3.24: Output of Code Using Labeled `continue` Statement

3.12 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. Match the description against its 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. The one or more expressions used in for statement are separated by the _____ operator.

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

6. You are trying to use the `if` statement to display the message "100 Not Equal". Which of the following code will help you to achieve this?

a.	<pre>int value=100; boolean bool=true; if((bool==true) && ((value+=100)==200)) { System.out.printf("Equal "+value); } else{ System.out.println(value + "Not Equal"); }</pre>	c.	<pre>int value=100; boolean bool=false; if((bool==false) && ((value+=100)==200)) { System.out.printf("Equal "+value); { } else System.out.println(value + "Not Equal"); }</pre>
b.	<pre>int value=100; boolean bool=false; if((bool==true) && ((value==100)==200)) { System.out.printf("Equal "+value); } else{ System.out.println(value + "Not Equal"); }</pre>	d.	<pre>int value=100; boolean bool=false; if((bool==true) && ((value+=100)==200)) { System.out.printf("Equal "+value); } else { System.out.println(value + "Not Equal"); }</pre>

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

7. Which of the following statements are true regarding nested-if statements?

a.	A nested-if statement should always be used inside <code>else</code> block	c.	The <code>if</code> statement should be within the same block as the <code>else</code> statement
b.	An <code>else</code> statement should always refer to the nearest <code>if</code> statement	d.	The inner <code>if</code> condition is evaluated only when the outer <code>if</code> condition is false

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

8. The _____ statement is used inside the switch-case statement to terminate the execution of the statement sequence.

(A)	switch	(C)	case
(B)	default	(D)	break

3.12.1 Answers

1.	B
2.	A
3.	B
4.	D
5.	C
6.	C
7.	B
8.	D

Summary

- A Java program is a set of statements, which are executed sequentially in the order in which they appear.
- The 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.

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
- Explain constructors and methods
- Explain process of creation and invocation of methods
- Explain passing and returning values from methods
- Explain variable argument methods
- Describe use of Javadoc to lookup methods
- Explain memory management in Java
- Explain object initializers
- Describe access specifiers and the types of access specifiers
- Explain concept of method overloading
- Explain 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> {
    // classbody
}
```

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 explanatory text. The first two lines (`String studName;` and `int studAge;`) are grouped together with an arrow pointing to the text "Fields or Instance Variables". The `void display()` method is highlighted with an arrow pointing to the text "Functions or Instance Methods". The entire `public static void main(String[] args)` block is highlighted with an arrow pointing to the same text "Functions or Instance Methods".

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`.

4.3.2 Creation of an Object: Two Stage Process

Alternatively, an object can be created using two steps that are declaration of a reference variable and dynamic memory allocation of an object.

Using this approach, an object reference is declared first, without using the `new` operator.

The syntax for declaring the object reference is as follows:

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

where,

`object_name`: Is just a variable that will not point to any memory location.

Figure 4.2 shows the effect of the statement, `Customer objCustomer`; which just declares a reference variable.

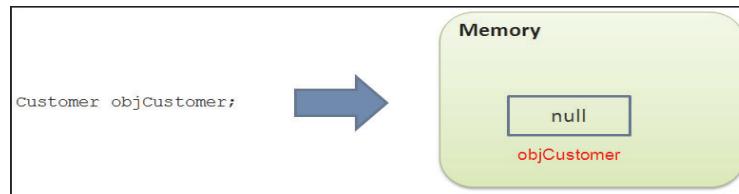


Figure 4.2: Declaration of a Reference Variable

By default, the value `null` is stored in the object's reference variable. In other words, it means that it does not point to an actual object. The object, `objCustomer` acts as a reference to an object of type `Customer`. If `objCustomer` is used at this point of time, without being instantiated, then the program will result in a compile time error.

Hence, before using such an object, the object should be initialized using the `new` operator. The `new` operator will dynamically allocate memory for an object. For example, `objCustomer = new Customer();`. This statement will allocate memory for the object and memory address of the allocated object is stored in the variable `objCustomer`.

Figure 4.3 shows the creation of object in the memory and storing of its reference in the variable, `objCustomer`.

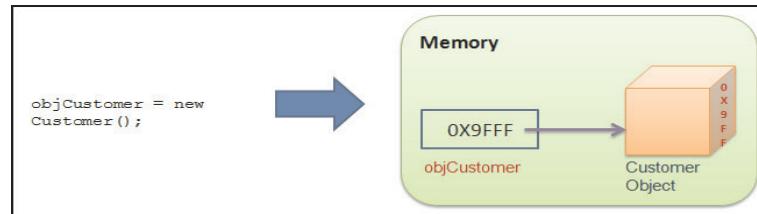


Figure 4.3: Creation of Object

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 the different data that are required to identify a customer in a banking domain and represent it as a single object?’.

Figure 4.4 shows a **Customer** object with its data requirement.

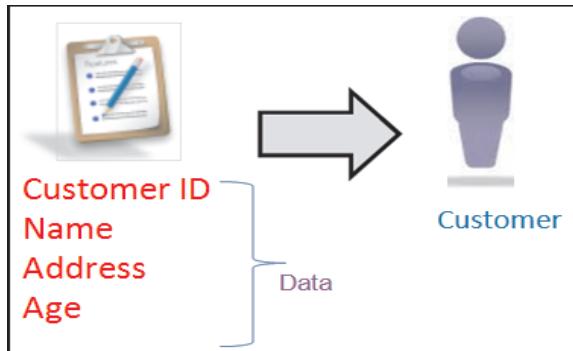


Figure 4.4: Data Requirements for Customer Object

As shown in figure 4.4, 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 4.5 shows various instances of the class with their own copy of instance variables.

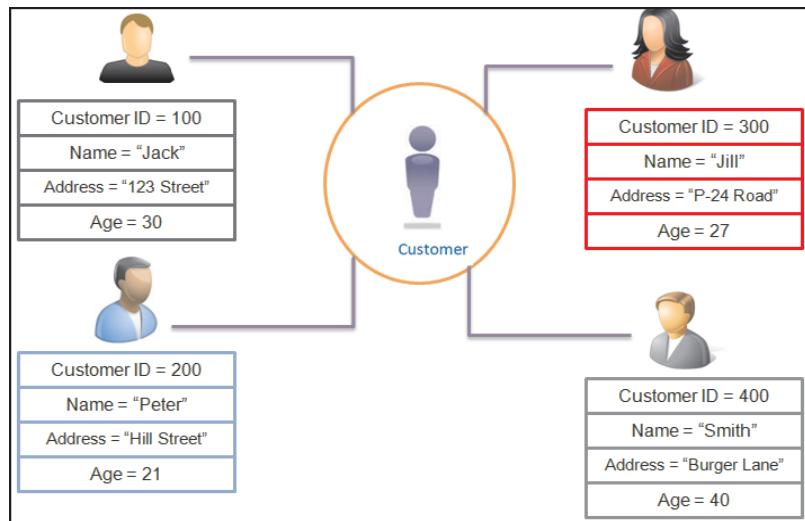


Figure 4.5: Concept of Instance Variables

As shown in figure 4.5, 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.

Instance variables are declared in the same way as local variables. Instance variables are declared inside a class, but outside any method definitions. They can be accessed only through the objects using the dot operator (.). Code Snippet 3 demonstrates the declaration of instance variables within a class in the Java program.

Code Snippet 3:

```

1: public class Customer {
2: // Declare instance variables
3: int customerID;
4: String customerName;
5: String customerAddress;
6: int customerAge;
7: /* As main() method is a member of class, so it can access other
8: * members of the class
*/
9: public static void main(String[] args) {
10: // Declares and instantiates an object of type Customer
11: Customer objCustomer1 = new Customer();
12: // Accesses the instance variables to store values
13: objCustomer1.customerID = 100;
14: objCustomer1.customerName = "John";
15: objCustomer1.customerAddress = "123 Street";
16: objCustomer1.customerAge = 30;
17: // Displays the objCustomer1 object details
18: System.out.println("Customer Identification Number: " + objCustomer1.
customerID);
19: System.out.println("Customer Name: " + objCustomer1.customerName);
20: System.out.println("Customer Address: " + objCustomer1.customerAddress);
21: System.out.println("Customer Age: " + objCustomer1.customerAge);
}
}

```

In the code, lines 3 to 6 declares instance variables. Line 11 creates an object of type **Customer** and stores its reference in the variable, **objCustomer1**. Lines 13 to 16 accesses the instance variables and

assigns them the values. Note, that to assign value to the instance variable, you qualify the variable name with an instance name followed by a dot operator.

Finally, lines 18 to 21 display the values assigned to the instance variables for the object, `objCustomer1`.

Figure 4.6 shows the allocation of `Customer` object in the memory. The reference of the object is stored in the variable, `objCustomer1` which is of type `Customer`.

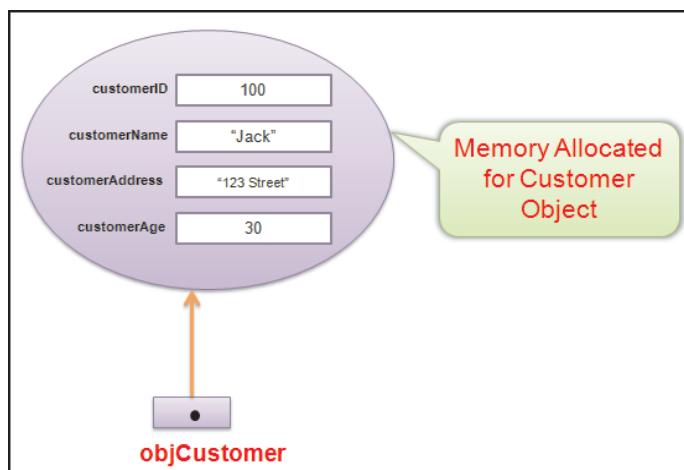


Figure 4.6: Creation of Customer Object

The output of the code is shown in figure 4.7.

A screenshot of a Java IDE showing the output of a program. The output window displays the following text:

```

run:
Customer Identification Number: 100
Customer Name: John
Customer Address: 123 Street
Customer Age: 30
BUILD SUCCESSFUL (total time: 1 second)

```

Figure 4.7: Output - Declaration of Instance Variables

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

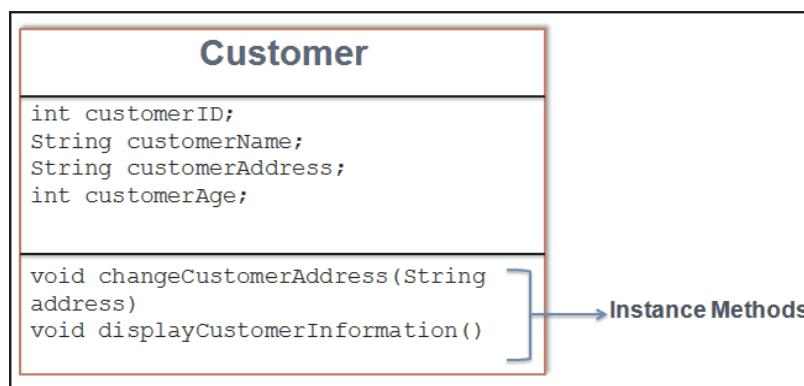


Figure 4.8: Instance Methods

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.

Figure 4.9 shows the declaration of an instance method within the class.

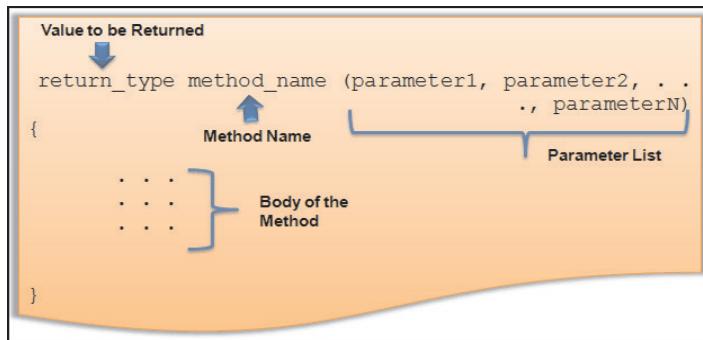


Figure 4.9: Declaration of an Instance Method

Each instance of the class has its own instance variables, but the instance methods are shared by all the instances of the class during execution.

Code Snippet 4 demonstrates the code that declares instance methods within the class, **Customer**.

Code Snippet 4:

```

public class Customer {
    // Declare instance variables
    int customerID;
    String customerName;
    String customerAddress;
    int customerAge;
    /**
     * Declares an instance method changeCustomerAddress to change the address of
     * the Customer object
     */
    void changeCustomerAddress (String address) {
        customerAddress = address;
    }
    /**
     * Declares an instance method displayCustomerInformation to display the
     * details of the Customer object
     */
    void displayCustomerInformation () {
        System.out.println ("Customer Identification Number: " + customerID);
        System.out.println ("Customer Name: " + customerName);
        System.out.println ("Customer Address: " + customerAddress);
        System.out.println ("Customer Age: " + customerAge);
    }
}

```

The class `Customer` is modified with the instance methods namely, `changeCustomerAddress()` and `displayCustomerInformation()`. The `changeCustomerAddress()` method will accept a string value through parameter address. It then assigns the value of `address` variable to the `customerAddress` field. As methods are part of class declaration, they can access the other class members, such as instance variables and methods of the class.

Similarly, the method `displayCustomerInformation()` displays details of the `Customer` object. Note, that when the `Customer` class is compiled, the compiler puts it in a `Customer.class` file. This class cannot be executed as there is no `main()` method present in it.

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

Code Snippet 5:

```
public class TestCustomer {
    /**
     * @param args commandline 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 4.10.

```

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 4.10: 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. Likewise, there may be other startup operations that are required to be performed in an application like 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 accept parameters.

Figure 4.11 shows the constructor declaration.

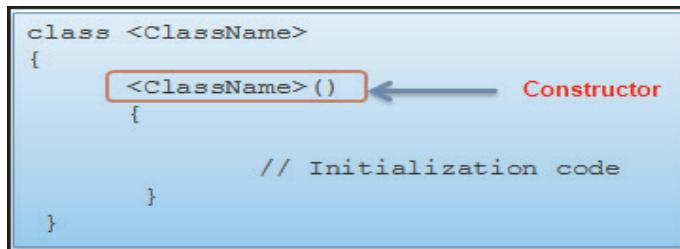


Figure 4.11: Constructor Declaration

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 6 demonstrates a class **Rectangle** with a constructor.

Code Snippet 6:

```

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 4.12 shows the use of `new` operator to understand the constructor invocation.

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

Figure 4.12: Invocation of Constructor

As shown in figure 4.12, the parenthesis after the class name indicates the invocation of the constructor.

Code Snippet 7 demonstrates the code to invoke the constructor for the class `Rectangle`.

Code Snippet 7:

```
public class TestConstructor {
    /**
     * @param args the commandline 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 4.13.

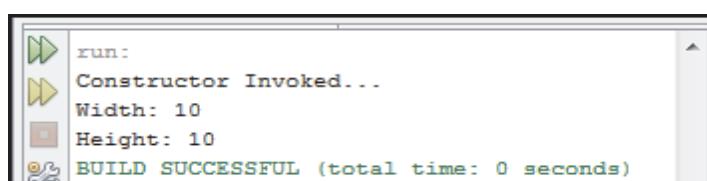


Figure 4.13: Output – Invocation of Constructor

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 8 demonstrates a class **Employee** for which no constructor has been defined.

Code Snippet 8:

```
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 9 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 9:

```
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 4.14.

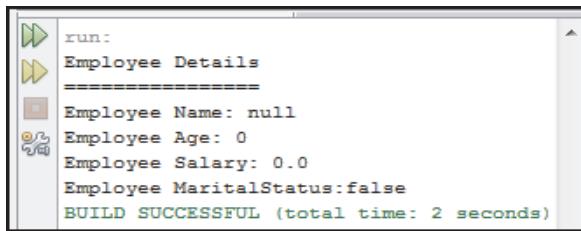


Figure 4.14: Output – Default Constructor

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 10 demonstrates a code that declares parameterized constructor for the `Rectangle` class.

Code Snippet 10:

```

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

Code Snippet 11:

```

public class RectangleInstances {
    /**
     * @param args the command line 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 4.15.

```

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 4.15: Output – Parameterized Constructor

Figure 4.16 displays both the instance of the class `Rectangle`. Each object contains its own copy of instance variables that are initialized through constructor.

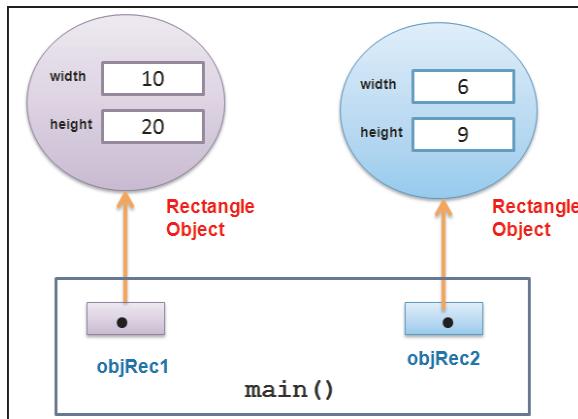


Figure 4.16: Instances of Rectangle Object

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

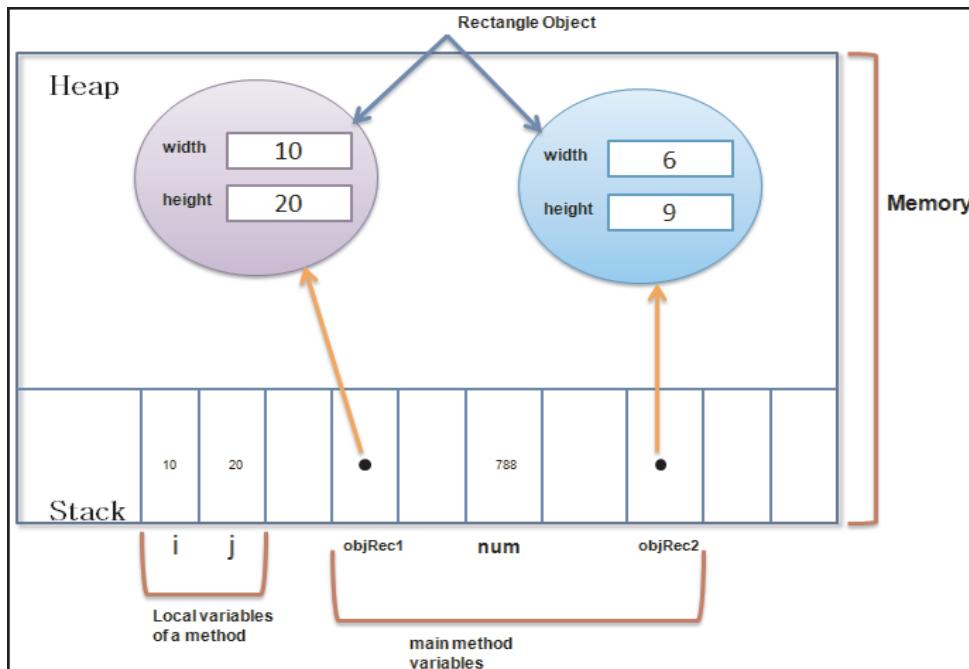


Figure 4.17: Memory Allocation for Method Variables and Objects

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 4.18 shows assigning of a value from one variable to another.

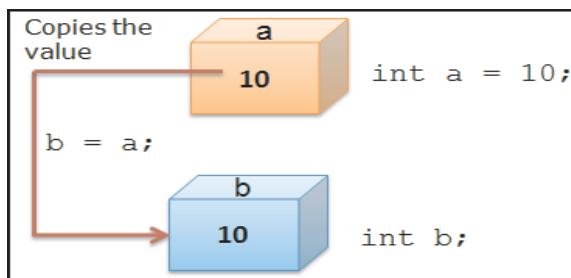


Figure 4.18: Copying of Value in Primitive Data Types

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 12 demonstrates assigning the reference of one object into another object reference variable.

Code Snippet 12:

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

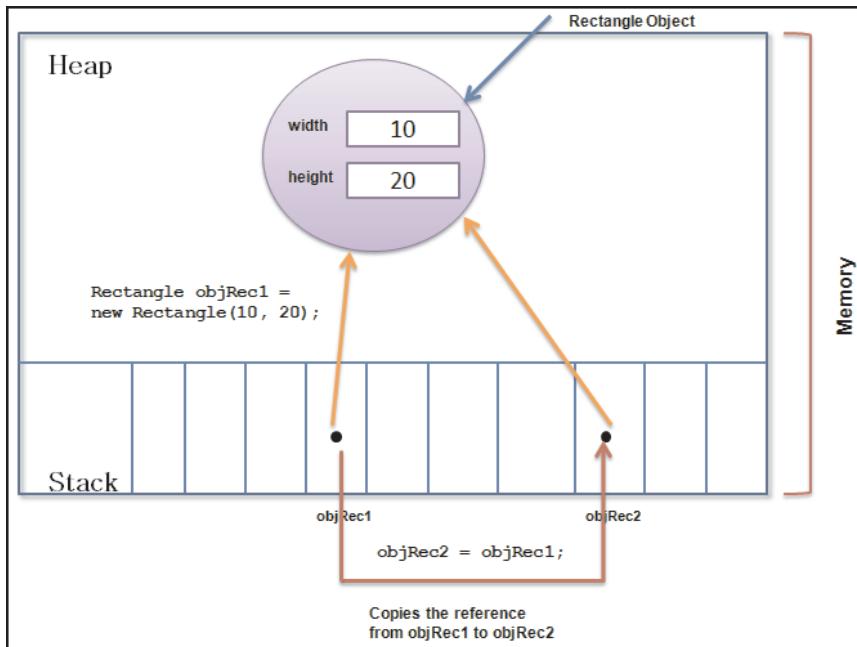


Figure 4.19: Copying of Reference on Stack

4.7 Encapsulation

Consider a scenario in which you want to learn how to drive a car. As a learner, it is not necessary for you to understand the details on how the engine works, how the petrol tank is built, how wheel implements right-turn or left-turn functionality, and so on. As a driver, you are interested and concerned with how to start and switch off the engine and how to provide fuel to the car. Also, you may be interested in knowing how to use gears and apply brakes to accomplish the complete understanding of the car. Thus, to drive a car, you are interested in the interface of the car, rather than how each part functions.

Similarly, in object-oriented programming, the concept of hiding implementation details of an object is achieved by applying the concept of 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 4.20 shows various access modifiers.

```

    /**
     * Displays the sum of 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)
    {
        int num3;
        num3=num1+num2;
        System.out.println("Result after addition is " + num3);
    }

```

session4.Calculator

public void add(int num1, int num2)

Displays the sum of two integers

Parameters:

- num1 - an integer variable storing the value of first number
- num2 - an integer variable storing the value of second number

Figure 4.20: Using Access Modifiers

Figure 4.20 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 4.20, 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
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier (default)	Y	Y	N	N
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.

Code Snippet 13 demonstrates the use of the concept of encapsulation in the class `NewRectangle`.

Code Snippet 13:

```

public class NewRectangle {
    // Declares instance variables
    private int width;
    private int height;
    /**
     * Declares a no-argument constructor
     */
    public NewRectangle() {
        System.out.println("Constructor Invoked...");
        width=10;
        height=10;
    }
    /**
     * Declares a parameterized constructor with two parameters
     * @param wid
     * @param heig
     */
    public NewRectangle (int wid, int heig) {
        System.out.println("Parameterized Constructor Invoked...");
        width=wid;
        height=heig;
    }
    /**
     * Displays the dimensions of the NewRectangle object
     */
    public void displayDimensions () {
        System.out.println("Width: " + width);
        System.out.println("Height: " + height);
    }
}

```

The code changes the access modifier of the instance variables or fields, `width` and `height` of the `Rectangle` class from default to `private`. This means that the class fields are not directly accessible from outside the class. This is done to restrict the access to data members of the class. Similarly, the access modifiers for the methods are changed to `public`. Thus, users can access the class members through its methods without impacting the internal implementation of the class.

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 14 demonstrates a Java program that declares a class, **Person** and initializes its fields.

Code Snippet 14:

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

Code Snippet 15:

```
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 4.21.

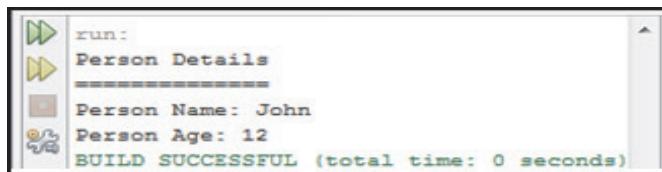


Figure 4.21: Output – Instance Variable Initializers

→ 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 16 demonstrates the class **Account** with an initialization block.

Code Snippet 16:

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

Code Snippet 17:

```
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 4.22.

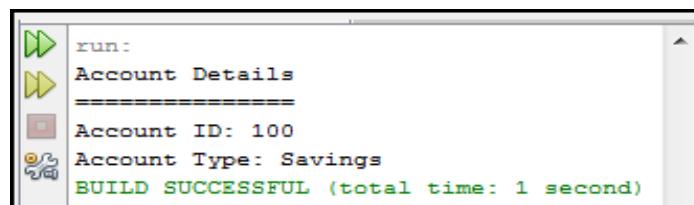


Figure 4.22: Output - Initialization Block

4.9 More on Methods

Consider a situation where in a user wants to perform some mathematical operations on two numbers. The numbers may be integers or floating-point numbers. One way to do this would be to declare several different variables in the class and write as many expressions as the number and type of mathematical operations required. In this case, all the operations will be performed at a time on the numbers during program execution.

However, what if the user wants only a single operation to be executed at a time? In such a case, it is necessary to have a feature that allows execution of only the required expressions and not the entire program. Methods in Java are such a feature that allows grouping of statements and execution of a specific set of statements instead of executing the entire program. Again, what if the user wants to restrict access to certain methods? Java provides a set of access modifiers that can help the user to accomplish this task.

A Java method can be defined as a set of statements grouped together for performing a specific task. For example, a call to the `main()` method which is the point of entry of any Java program, will execute all the statements written within the scope of the `main()` method.

The syntax for declaring a method is as follows:

Syntax:

```
modifier return_type method_name([list_of_parameters]) {
    // Body of the method
}
```

where,

modifier: Specifies the visibility of the method. Visibility indicates which object can access the method. The values can be `public`, `private`, or `protected`.

return_type: Specifies the data type of the value returned by the method.

method_name: Specifies the name of the method.

list_of_parameters: Specifies the comma-delimited list of values passed to the method.

Generally, a method declaration has following six components, in order:

1. Modifiers such as `public`, `private`, and `protected`.
2. A return type that indicates the data type of the value returned by the method. The return type is set to `void` if the method does not return a value.
3. The method name that is specified based on certain rules. A method name:
 - cannot be a Java keyword
 - cannot have spaces
 - cannot begin with a digit
 - cannot begin with any symbol other than a \$ or _
 - can be a verb in lowercase
 - can be a multi-word name that begins with a verb in lowercase, followed by adjectives or nouns
 - can be a multi-word name with the first letter of the second word and each of following words capitalized
 - should be descriptive and meaningful

Some valid method names are `add`, `_view`, `$calc`, `add_num`, `setFirstName`, `compareTo`, `isValid`, and so on.

4. The parameter list in parenthesis is separated with a comma delimiter. Each parameter is preceded by its data type. If there are no parameters, an empty parenthesis is used.
5. An exception list that specifies the names of exceptions that can be thrown by the method. An exception is an event encountered during the execution of the program, disrupting the flow of program execution.

6. The method body which consists of a set of statements enclosed between curly braces '{}'. The method body can have variables, method calls, and even classes.

Two components of a method declaration namely, method name and parameter types comprise the method signature.

4.9.1 Creating and Invoking Methods

As discussed earlier, methods help to segregate different tasks to provide modularity to the program. A program is modular when different tasks in a program are grouped together into modules or sections. For example, to perform different types of mathematical operations such as addition, subtraction, multiplication, and so on, a user can create individual methods as shown in figure 4.23.

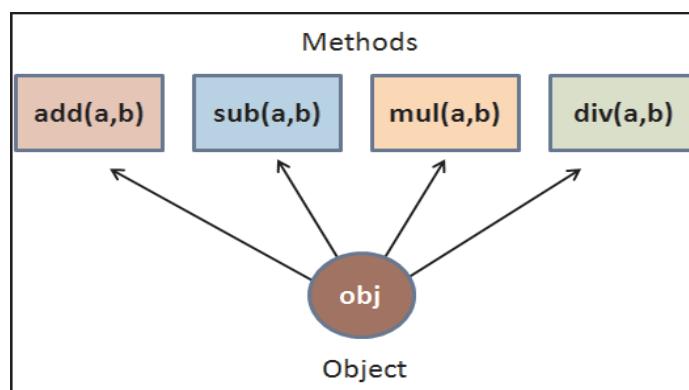


Figure 4.23: Methods and Object

Figure 4.23 shows an object named `obj` accessing four different methods namely, `add(a,b)`, `sub(a,b)`, `mul(a,b)`, and `div(a,b)` for performing the respective operations on two numbers. Thus, creating methods gives flexibility to the program so that an object can perform operations as required without having to execute the entire program.

To create a method that adds two numbers, the user can write a method as depicted in Code Snippet 18.

Code Snippet 18:

```

public void add(int num1, int num2) {
    int num3; // Declare a variable
    num3 = num1 + num2; // Perform the addition of numbers
    System.out.println("Addition is " + num3); // Print the result
}
    
```

Code Snippet 18 defines a method named `add()` that accepts two parameters `num1` and `num2`, each of type integer. Also, the method has been declared with the `public` access modifier which means that it can be accessed by all objects. The return type set to `void` indicates that the method does not return anything.

The method body consists of three statements. The first statement '`int num3;`' is a declaration of an integer variable named `num3`. The second statement '`num3 = num1 + num2;`' is an addition operation performed on parameters `num1` and `num2` using the arithmetic operator '+'. The result is stored in a third variable `num3` by using the assignment operator '='.

The last statement '`System.out.println("Addition is "+ num3);`' is used to print the value of variable `num3`. The method signature is '`add(int, int)`'.

While creating a method, a user defines the structure and function of the method. In other words, a developer specifies what the method will perform.

However, to use the method, it must be called or invoked. When a program calls a method, the control is transferred to the called method. The called method executes and returns control to the caller. The call is returned back after the return statement of a method is executed or when the closing brace is reached.

A method can be invoked in one of following ways:

- If the method returns a value, then, a call to the method results in return of some value from the method to the caller. For example,

```
int result = obj.add(20, 30);
```

Assuming that the `add()` method returns an integer value, the method call is placed to the right of the assignment operator. The returned value will be stored in a variable named `result` placed on the left of the assignment operator. The return type of a method specifies the type of value that the method returns. Thus, the data type of the variable `result` must be same as the return type of the method `add()`.

- If the method's return type is set to `void`, then, a call to the method results in execution of the statements within the method without returning any value to the caller. For example, the `add()` method in Code Snippet 1 returns `void` and prints the output using the statement `System.out.println("Addition is "+ num3);`

Hence, a call to the method would be `obj.add(23, 30)` without anything returned to the caller. In such cases, a call to the method will be a statement.

Consider the project `Session4` created in the NetBeans IDE as shown in figure 4.24.

```

1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package session4;
6
7  /**
8   *
9   * @author vincent
10  */
11 public class Calculator {
12
13     /**
14      * @param args the command line arguments
15     */
16     public static void main(String[] args) {
17         // TODO code application logic here
18     }
19 }

```

Figure 4.24: Project - Session 4

The project consists of a package named **session4** with the **Calculator** class that has the **main()** method. Several methods for mathematical operations can be added to the class.

Code Snippet 19 demonstrates an example of creation and invocation of methods.

Code Snippet 19:

```
package session4;
public class Calculator {
    // Method to add two integers
    public void add(int num1, int num2) {
        int num3;
        num3 = num1 + num2;
        System.out.println("Result after addition is " + num3);
    }
    // Method to subtract two integers
    public void sub(int num1, int num2) {
        int num3;
        num3 = num1 - num2;
        System.out.println("Result after subtraction is " + num3);
    }
    // Method to multiply two integers
    public void mul(int num1, int num2) {
        int num3;
        num3 = num1 * num2;
        System.out.println("Result after multiplication is " + num3);
    }
    // Method to divide two integers
    public void div(int num1, int num2) {
        int num3;
        num3 = num1 / num2;
        System.out.println("Result after division is " + num3);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the Calculator class
        Calculator objCalc = new Calculator();
```

```
// Invoke the methods with appropriate arguments
objCalc.add(3, 4);
objCalc.mul(3, 4);
}
}
```

The class **Calculator** consists of methods such as **add()**, **sub()**, **mul()**, and **div()** that are used to perform respective operations. Each method accepts two integers as parameters. The **main()** method creates an object, **objCalc** of class **Calculator**. The object **objCalc** uses the dot ‘.’ operator to call or invoke the **add()** and **mul()** methods. When the program is executed in NetBeans IDE, first the **main()** method is executed by the JVM. The runtime creates the object **objCalc** of class **Calculator**. Next, the **add()** method is invoked with two values 3 and 4. The **add()** method is processed and the print statement displays the result. Lastly, the **mul()** method is invoked with two parameters and prints the result.

Note - The statement **package session4;** indicates that the class belongs to a package named **session4**. A package is a folder used to group related classes. Also, the text within **/**... */** symbols are the default javadoc comments generated by the runtime to describe the code.

Figure 4.25 shows the output of the program.

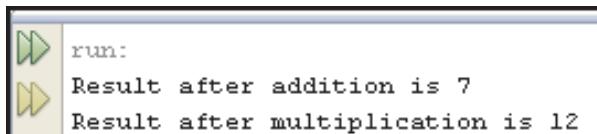


Figure 4.25: Output of Calculator Program

Figure 4.25 shows the output of methods **add()** and **mul()** that were invoked by the object **objCalc** of **Calculator** class.

Note - All the methods defined in class **Calculator** are instance methods. Instance methods are those methods that are invoked using an object of the class. Also, all the methods are public which means that they are accessible to all objects across packages.

4.9.2 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 20 demonstrates an example of passing arguments by value.

Code Snippet 20:

```
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 4.26 shows the output of the code.

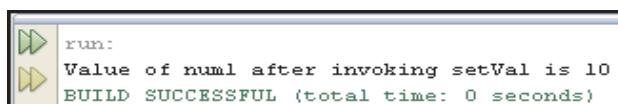


Figure 4.26: Output After Passing Arguments by Value

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 21 demonstrates an example of passing arguments by reference.

Code Snippet 21:

```
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 4.27 shows the output of the code.

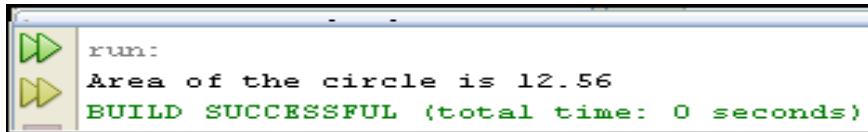


Figure 4.27: Output After Passing Arguments by Reference

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 4 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 22.

Code Snippet 22:

```
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`.

4.9.3 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 23 demonstrates an example of a variable argument method.

Code Snippet 23:

```
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 4.28 shows the output of the code.

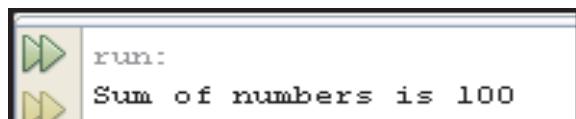


Figure 4.28: Output After Using Variable Argument Method

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.

4.9.4 Using Javadoc to Lookup Methods of a Class

Java provides a JDK tool named **Javadoc** that is used to generate API documentation as an HTML page from declaration and documentation comments. These comments are descriptions of the code written in a program. Different terminologies used while generating javadoc are as follows:

- **API documentation or API docs:** These are the online or hard copy descriptions of the API that are primarily intended for the programmers. The API specification consists of all assertions for a proper implementation of the Java Platform to ensure that the ‘write once, run anywhere’ feature of Java is retained.
- **Documentation comments or doc comments:** These are special comments in the Java source code. They are written within the `/** ... */` delimiters. These comments are processed by the **Javadoc** tool for generating the API docs.

Four types of source files from which **Javadoc** tool can generate output are as follows:

- Java source code files (`.java`) which consist of the field, class, constructor, method, and interface comments.
- Package comment files that consist of package comments.
- Overview comment files that contain the comments about set of packages.
- Miscellaneous files that are unprocessed such as images, class files, sample source codes, HTML files, applets, and any other file that is referenced from the previous files.

A doc comment is written in HTML and it must precede a field, class, method, or constructor declaration. The doc comment consists of two parts namely, a description and block tags. For example, consider the class given in Code Snippet 24.

Code Snippet 24:

```
public class Circle {
    private double PI=3.14;
    public double calcArea(double rad) {
        return (3.14 * rad * rad);
    }
}
```

Code Snippet 24 consists of a class **Circle**, a variable **PI**, and a method **calcArea()** that accepts radius as a parameter.

Now, a doc comment for **calcArea()** method can be written as depicted in Code Snippet 25.

Code Snippet 25:

```
/**
 * Returns the area of a circle
 *
 * @param rad a variable indicating radius of a circle
 * @return the area of the circle
 * @see PI
 */
```

Here, the first statement is the method description whereas `@param`, `@return`, and `@see` are block tags that refer to the parameters and return value of the method. A blank comment line must be provided between the description line and block tags. The HTML generated from running the **Javadoc** tool is as follows:

calcArea

```
public double calcArea(double rad)
```

Returns the area of a circle

Parameters:

`rad` – a variable storing the radius of the circle

Returns:

the area of a circle

See Also:

PI

To use **Javadoc** tool to lookup methods of a class, perform following steps:

1. Open the **Calculator** class created earlier.
2. Open Javadoc widow by clicking **Window** → **Other** → **Javadoc** as shown in figure 4.29.

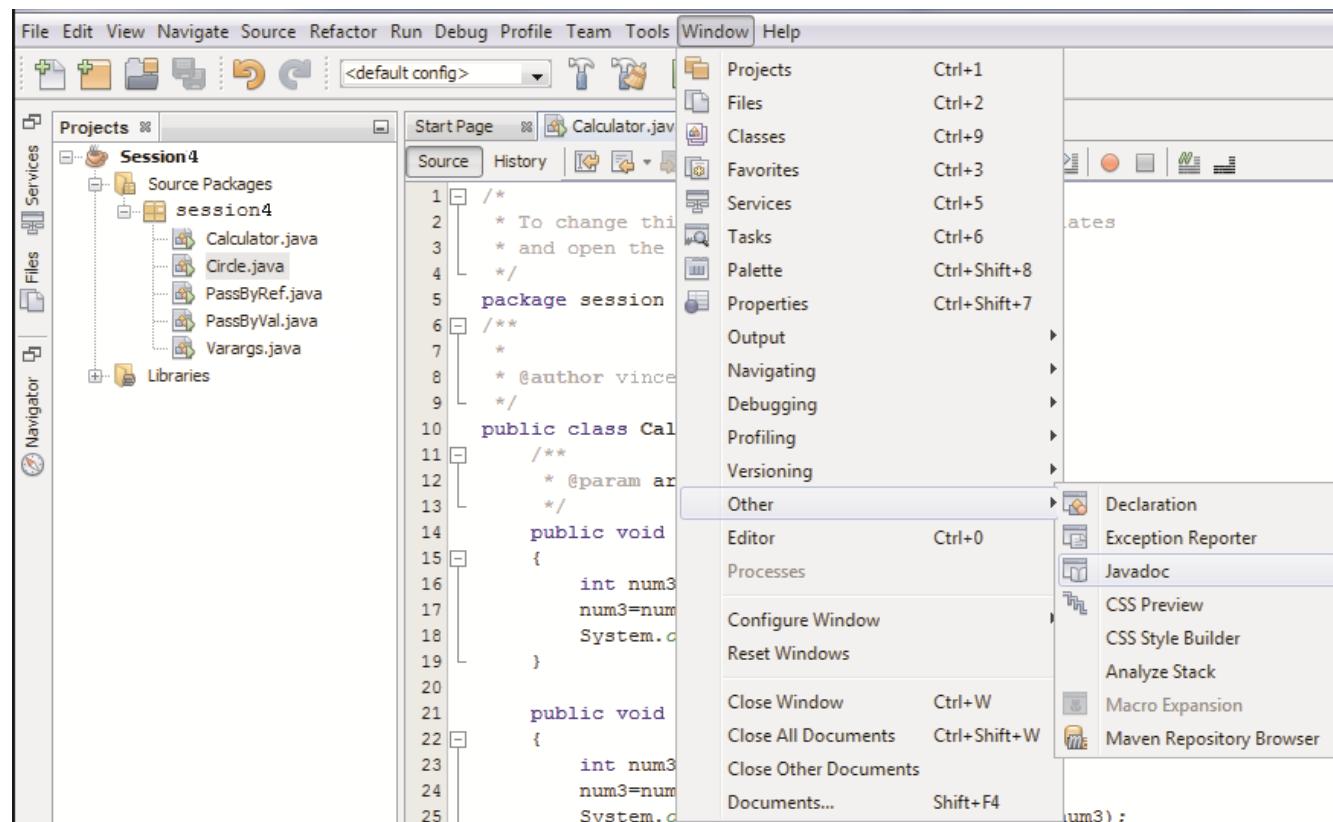


Figure 4.29: Opening Javadoc Window

Figure 4.30 shows the **Javadoc** window opened at the bottom.



Figure 4.30: Javadoc Window

The **Javadoc** window shows the description of the `Calculator` class. However, since javadoc for the class is not generated still, it gives a message that '**Javadoc not found**'.

3. Select the `println()` method of `System.out.println()` statement of the `add()` method and then, open the **Javadoc** window. The window will show the built-in Java documentation of the `println()` method as shown in figure 4.31.

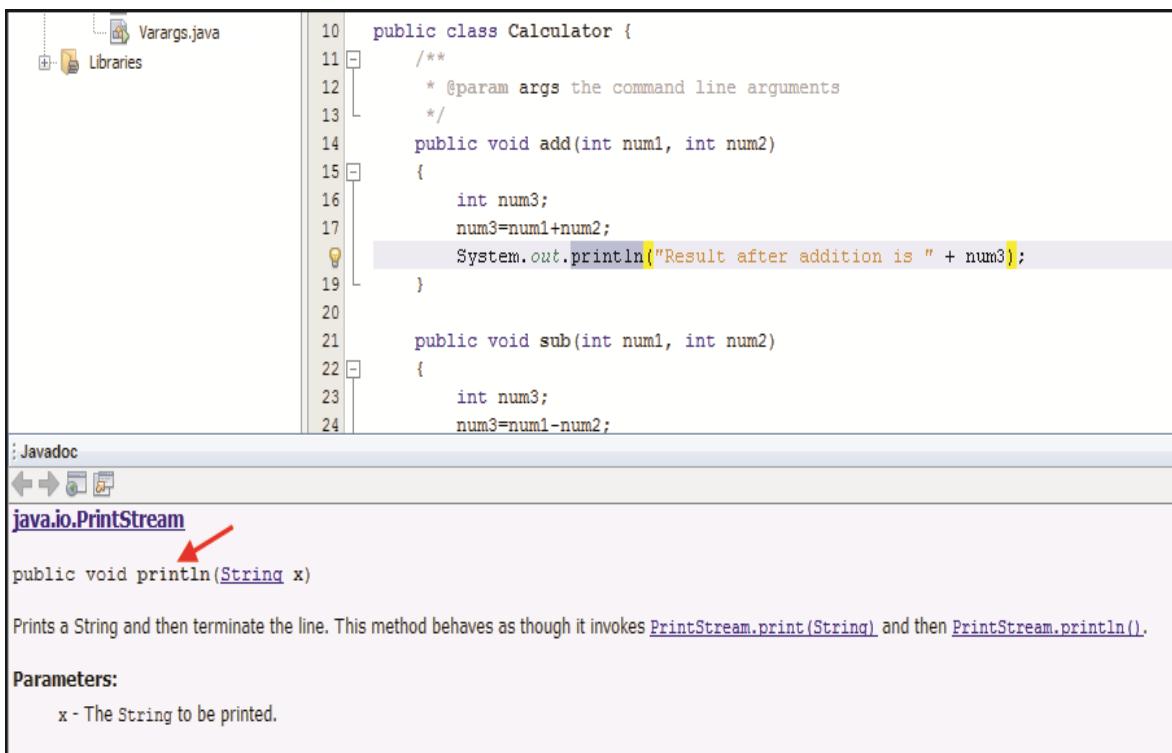


Figure 4.31: Javadoc of Println Method

Figure 4.31 shows the built-in javadoc of the `println()` method along with its description and parameters as defined in the javadoc.

4. Select the `add()` method. Again, no details about `add` method will be displayed since javadoc does not exist for it as shown in figure 4.32.

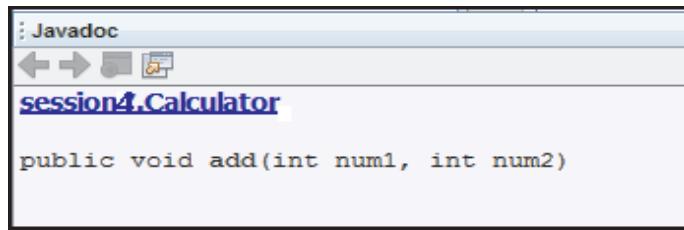


Figure 4.32: Javadoc of Add Method

5. Type following javadoc comments above the `add()` method:

```
/*
 * Displays the sum of 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
 */
```

6. Click **Run → Generate Javadoc**. The NetBeans IDE generates the javadoc for the project **Session4** and displays it in the browser as shown in figure 4.33.

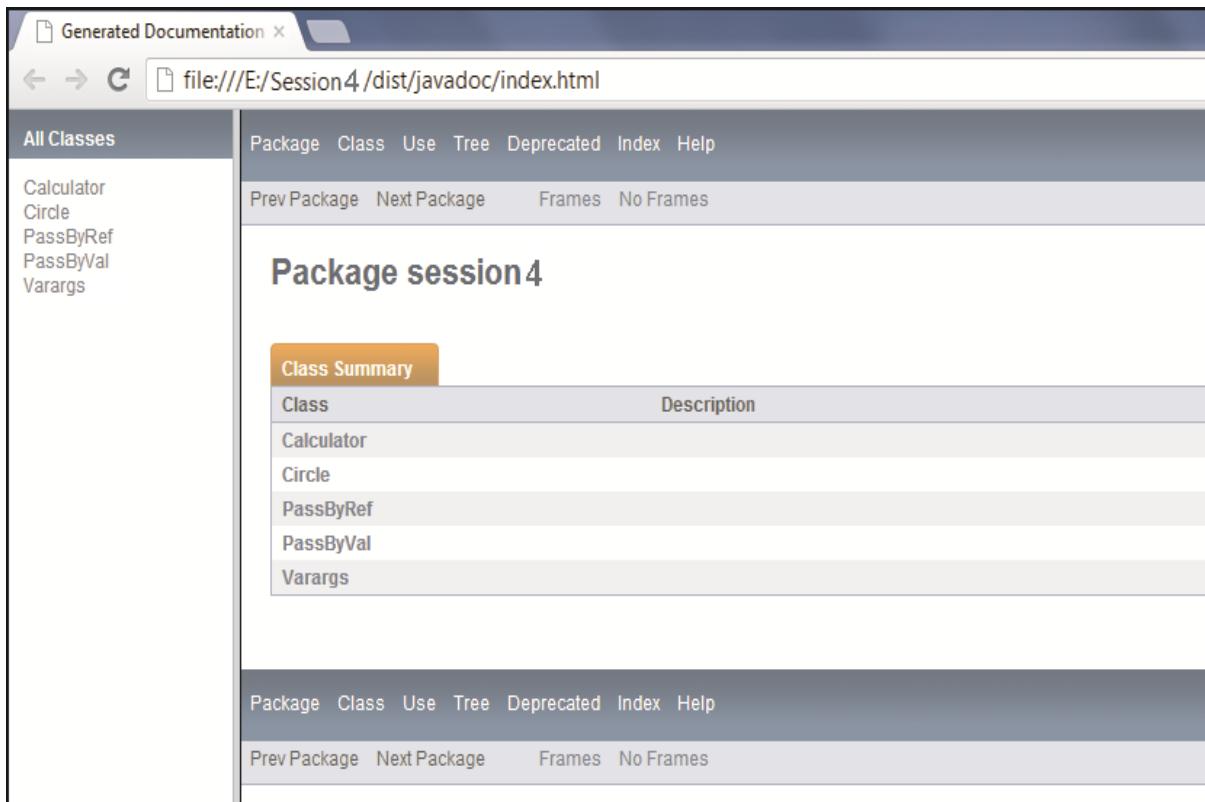


Figure 4.33: Javadoc Generated for Project Session4

The javadoc lists various classes available in the selected package. To move to the next or previous package, one can click the **PrevPackage** and **NextPackage** links on the page.

- Click the **Calculator** class under the **Class Summary** tab.

The javadoc of the **Calculator** class is shown in figure 4.34.

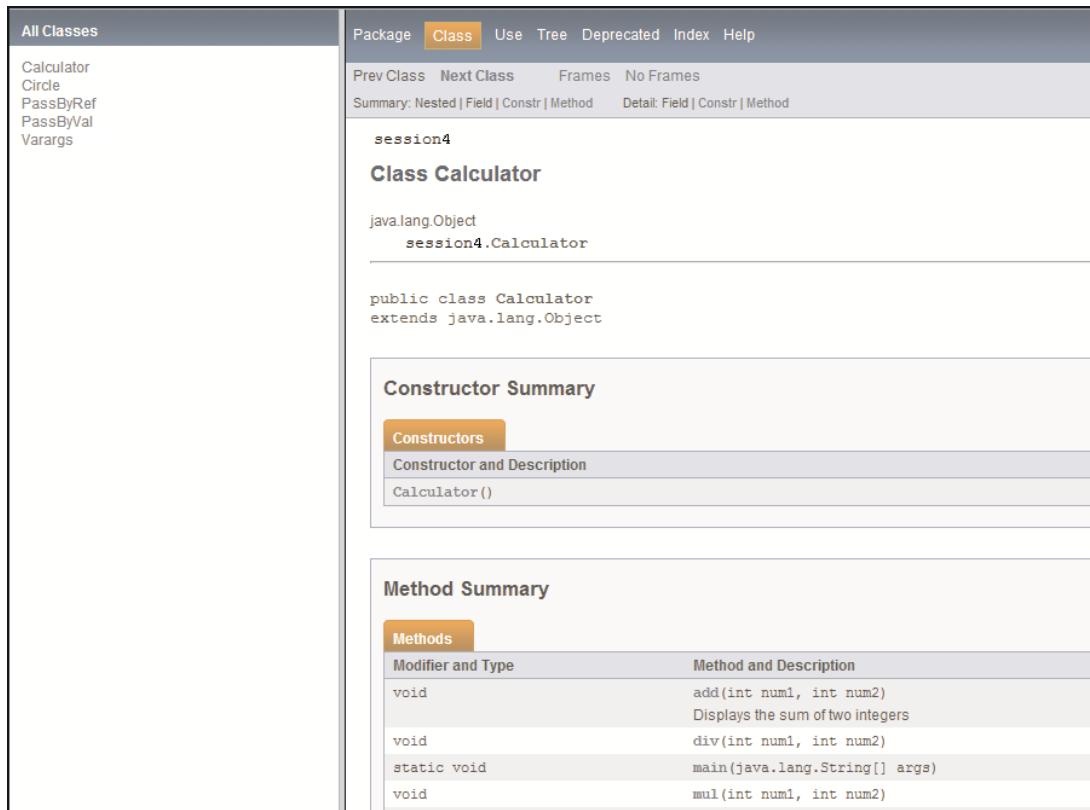
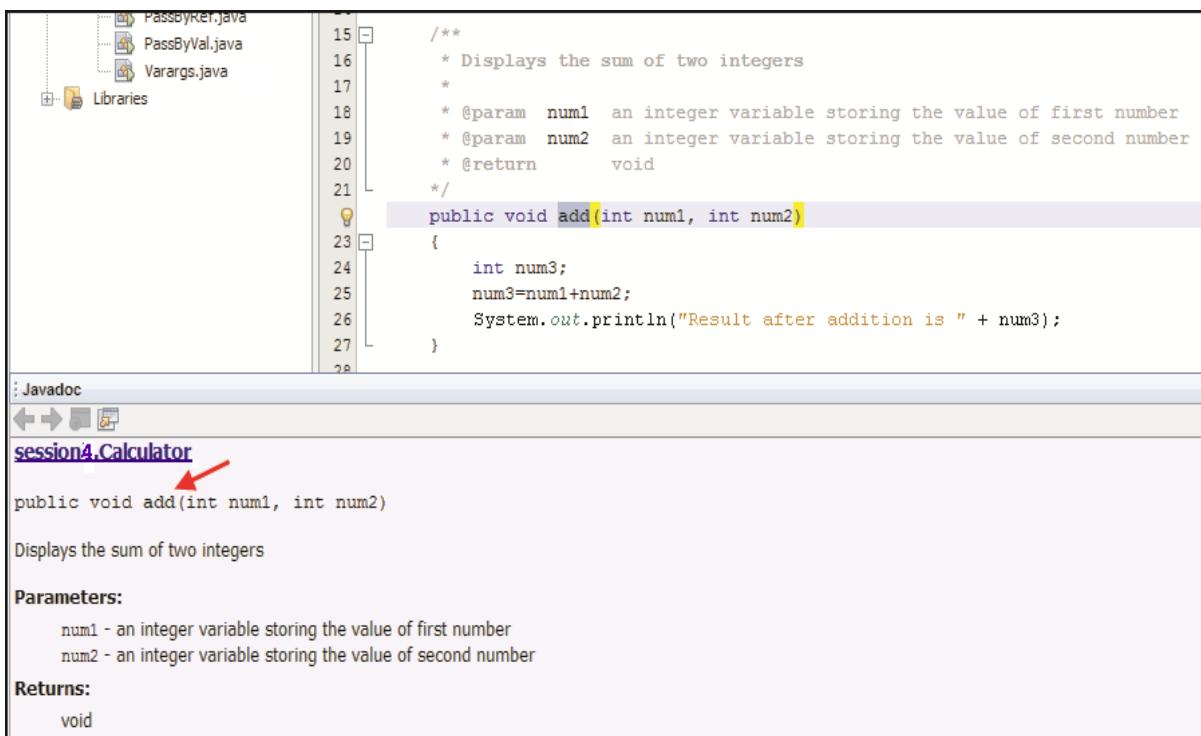


Figure 4.34: Javadoc of Class Calculator

Similarly, javadoc for other classes can also be viewed. This shows the structure of the class, its constructor, and its various methods.

- Now, select the **add()** method in the NetBeans IDE and open the **Javadoc** window.

The window will show the javadoc created for the `add()` method as shown in figure 4.35.



The screenshot shows an IDE interface with several files listed in the left sidebar: PassByRef.java, PassByVal.java, Varargs.java, and Libraries. The main pane displays the following Java code:

```

15  /**
16   * Displays the sum of two integers
17   *
18   * @param num1 an integer variable storing the value of first number
19   * @param num2 an integer variable storing the value of second number
20   * @return void
21  */
22  public void add(int num1, int num2)
23  {
24      int num3;
25      num3=num1+num2;
26      System.out.println("Result after addition is " + num3);
27  }

```

Below the code, the Javadoc tool is open, showing the generated documentation for the `add()` method. A red arrow points to the method signature in the code. The Javadoc content includes:

- session4.Calculator**
- public void add(int num1, int num2)**
- Displays the sum of two integers**
- Parameters:**
 - num1 - an integer variable storing the value of first number
 - num2 - an integer variable storing the value of second number
- Returns:**
 - void

Figure 4.35: Javadoc for `add()` Method

Notice that the position of the description, the parameters, and the return type statements was automatically adjusted by the **Javadoc** tool. Thus, it can be concluded that javadoc generates an HTML file containing information about each class. It will output an index and a hierarchy tree. Similarly, **Javadoc** tool can be used to lookup for other built-in methods as well as to generate javadoc for user-defined methods.

4.10 Using Access Modifiers with Variables and Methods

The access modifiers discussed earlier can be used with variables and methods of a class to restrict access from other classes. Code Snippet 26 demonstrates an example of using access modifiers with variables and methods.

Code Snippet 26:

```

package session4;
public class Employee {

    // Variables with default access
    int empID; // Variable to store employee ID
    String empName; // Variable to store employee name

    // Variables with private and protected access
    private String SSN; // Variable to store social security number
    protected String empDesig; // Variable to store designation
}

```

```

/**
 * Parameterized constructor
 *
 * @param ID an integer variable storing the employee ID
 * @param name a String variable storing the employee name
 * @return void
 */
public Employee(int ID, String name) {
    empID = ID;
    empName = name;
}
// Define public methods
/**
 * Returns the value of SSN
 *
 * @return String
 */
public String getSSN(){ // Accessor for SSN
    return SSN;
}
/**
 * Sets the value of SSN
 *
 * @param ssn a String variable storing the social security number
 * @return void
 */
public void setSSN(String ssn) { // Mutator for SSN
    SSN = ssn;
}
/**
 * Sets the value of Designation
 *
 * @param desig a String variable storing the employee designation
 * @return void
 */
public void setDesignation(String desig) { // Public method
    empDesig = desig;
}
/**
 * Displays employee details
 *
 * @return void
 */
public void display(){ // Public method
    System.out.println("Employee ID is "+ empID);
    System.out.println("Employee name is "+ empName);
    System.out.println("Designation is "+ empDesig);
}

```

```

        System.out.println("SSN is "+ SSN);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the Employee class
        Employee objEmp1 = new Employee(1200, "Roger Stevens");
        // Assign values to public variables
        objEmp1.empDesig = "Manager";
        objEmp1.SSN = "281-72-3873";
        // Invoke the public method
        objEmp1.display();
    }
}

```

Figure 4.36 shows the output of the code.

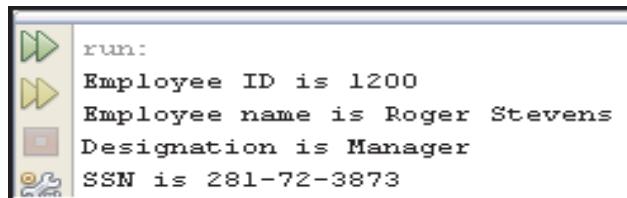


Figure 4.36: Using Access Modifiers with Variables and Methods

In Code Snippet 26, the class `Employee` has been declared `public`. The class has two members with default access modifier namely, `empID` and `empName`, a protected class member named `empDesig`, and a private class member named `SSN`. The class has one parameterized constructor that is used to set the values of member variables `empID` and `empName`. The class also has an accessor and a mutator method for variable `SSN` and two other `public` methods.

The `main()` method creates an object of `Employee` class with values of `empID` and `empName`. Next, the values of `empDesig` and `SSN` are specified by directly accessing the variables with the object `objEmp1` even though `empDesig` is protected and `SSN` is private. This is because, `objEmp1` is an object present in the same class. So, `objEmp1` has access to data members with any access modifier. The `display()` method is used to display all the values as shown in the output.

Code Snippet 27 demonstrates the use of access modifiers in another class named `EmployeeDetails` but in the same package as `Employee` class.

Code Snippet 27:

```

package session4;
public class EmployeeDetails {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Instantiate the Employee class within EmployeeDetails class
        Employee objEmp = new Employee(1300, "Clara Smith");
        // Assign value to protected variable
        objEmp.empDesig="Receptionist";
        // Use the mutator method to set the value of private variable
        objEmp.setSSN("282-72-3873");
        // Invoke the public method
        objEmp.display();
    }
}

```

Figure 4.37 shows the output of the code.

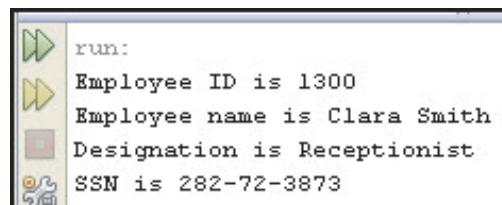


Figure 4.37: Using Members of One Class in Another Class

Within the `main()` method of `EmployeeDetails` class, an object of `Employee` is created to set the values of `empID` and `empName`. Next, value of `empDesig` is specified by directly accessing the `protected` variable `empDesig`. This is because, a `protected` variable can be accessed by another class of the same package even if it is not a child class of `Employee`.

However, to set the value of `SSN`, the `setSSN()` method is used. This is because `SSN` is a `private` member variable in `Employee` class and hence, cannot be directly accessed by other classes. Finally, the `display()` method is used to print all the details as shown in the output.

Classes of other packages that are not child class of `Employee` can set the value of `empID` and `empName` by using the constructor, of `empDesig` by using `setDesignation()` method, and of `SSN` by using `setSSN()` method.

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 4.38 shows an example of overloaded `add()` methods.

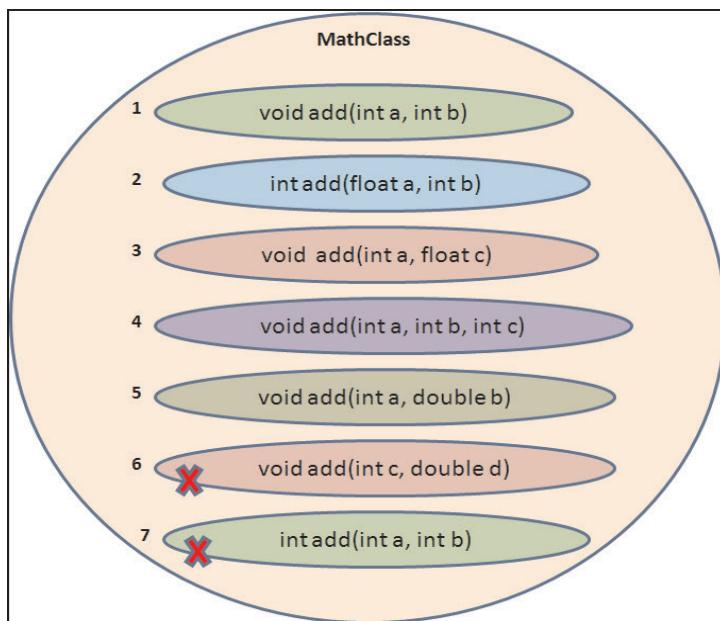


Figure 4.38: Overloading Methods

Figure 4.38 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 4.38 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 4.38 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 28 demonstrates an example of method overloading.

Code Snippet 28:

```
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 4.39 shows the output of the code.

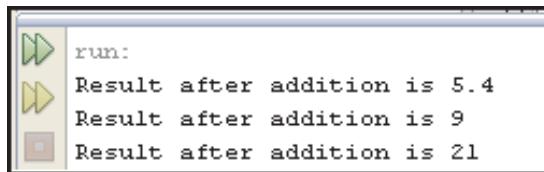


Figure 4.39: Output of Overloaded Methods

Figure 4.39 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 29 demonstrates an example of constructor overloading.

Code Snippet 29:

```
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 4.40 shows the output of the program.

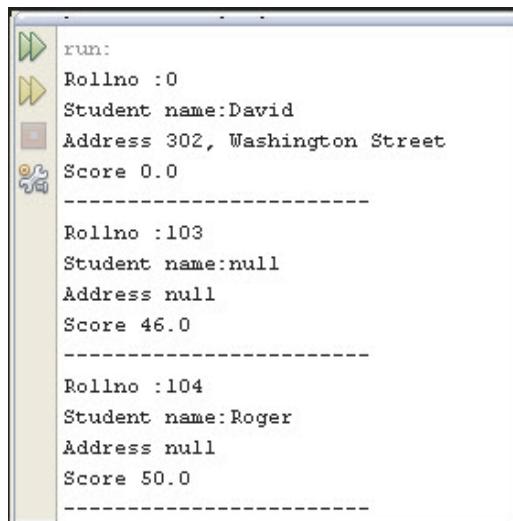


Figure 4.40: Output of Constructor Overloading

Figure 4.40 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 30.

Code Snippet 30:

```

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 31.

Code Snippet 31:

```

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 = 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 32.

Code Snippet 32:

```
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 33.

Code Snippet 33:

```
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 33 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

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

a.	<code>objJohn.rollNo=1151;</code> <code>Student objMartin=new Student();</code>
b.	<code>public static void main(String[] args) {</code> <code>Student objJohn=new Student();</code>
c.	<code>System.out.println(objMartin.rollNo);</code> }
d.	<code>objMartin.rollNo=1152;</code> <code>System.out.println(objJohn.rollNo);</code>
e.	<code>class Student {</code> <code>int rollNo;</code>

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

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

6. _____ are the list of variables specified in a method declaration.

(A)	Arguments	(C)	Values
(B)	Variables	(D)	Parameters

7. Which of the following statements about a method name are true?

a.	Can be a Java keyword	c.	Can be a multi-word name with the first letter of each of the second and remaining words capitalized
b.	Cannot begin with any symbol other than a \$ or _	d.	Can have spaces

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

8. Match the following access modifiers with the corresponding descriptions with respect to variables and methods of a class.

	Access Modifier		Description
a.	Private	1.	Can be accessed by all classes in the same package and only by subclasses in other packages.
b.	Public	2.	Gets applied to any class, field, or method that has not declared any access modifier.
c.	Default	3.	Visible to any class in a Java application in the same package or in another package.
d.	Protected	4.	Cannot be accessed from outside the enclosing class.

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

9. Which of the following options are parts of method signature?

a.	Return type	c.	Access modifier
b.	Method name	d.	Parameter types

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

10. Consider the following code:

```
public class Vehicle {
    int Vno;
    String Vtype;
    public Vehicle(int Vno, String Vtype) {
        Vno=Vno; // line1
        Vtype=Vtype; // line2
    }
}
```

The code is giving the warning '**Assignment to Itself**'. What change must be made in line1 and line2 to resolve the issue?

(A)	Vno=3764; Vtype="Car";	(C)	this.Vno=Vno; this.Vtype=Vtype;
(B)	Vno=this.Vno; Vtype=this.Vtype;	(D)	Vno=0; Vtype="";

11. Consider the following code:

```
public class Employee{
    int EmpID;
    String EmpName, Designation;
    public Employee() {
        EmpID=0;
        EmpName="";
        Designation="";
    }
    public Employee(int ID, String name) {
        EmpID=ID;
        EmpName=name;
    }
    public Employee(int ID, String name, String desig) {
        EmpID=ID;
        EmpName=name;
        Designation=desig;
    }
}
```

Which of the following programming concept has been used in the code?

(A)	Method Overriding	(C)	Method Overloading
(B)	Constructor Overloading	(D)	Operator Overloading

4.12.1 Answers

1.	C
2.	B
3.	A
4.	D
5.	C
6.	D
7.	C
8.	D
9.	B
10.	C
11.	B

```

    g pack();
    String pac()
    > String pr
    > void string
    > catch {
    >     catch

```

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

1. 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:
 - a. As a point is formed with two co-ordinates on the plane, include **x** and **y** as the fields in the **Point** class.
 - b. Implement the methods, such as **setX()**, **setY()**, and **displayPoints()** in the **Point** class to represent the behavior of the **Point** class.
 - c. Implement necessary constructors to initialize the objects of the **Point** class during their creation.
 - d. 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’.
 2. 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);
    }
}
```
3. **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

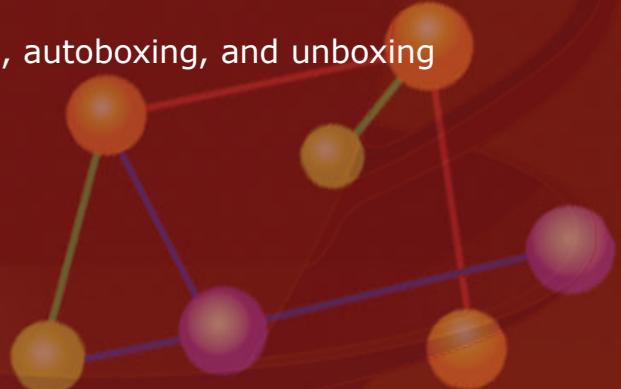
Arrays and Strings

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 command line arguments and 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
- Explain command line arguments
- 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.

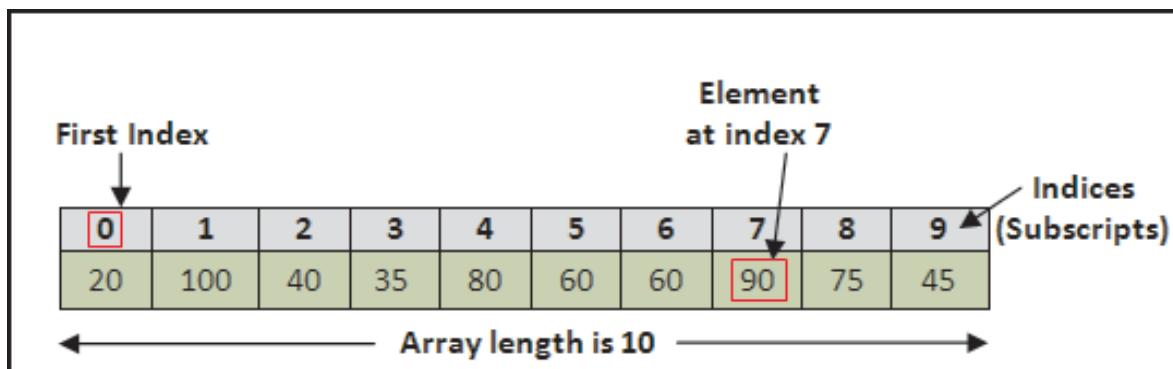


Figure 5.1: Array

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 the array named **marks** and its elements with their values and indices.

marks[4]	
Element	Value
marks[0]	65
marks[1]	47
marks[2]	75
marks[3]	50

Figure 5.2: Array **marks**

Figure 5.2 shows a single-dimensional array named **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.

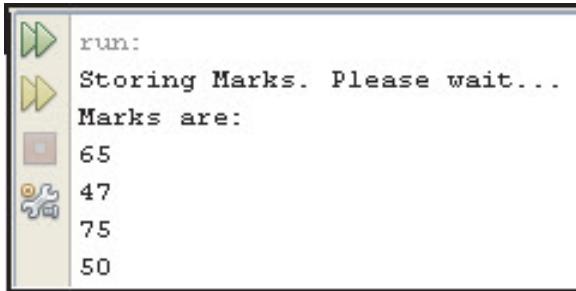


Figure 5.3: Output of Single-dimensional Array

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, the enhanced `for` loop is used to iterate through the columns of the row selected in the outer loop using `marks[row]`. The code `sum = sum + value` will add up the values of all columns of the currently selected row. The selected row is indicated by the subscript variable `row`.

Figure 5.7 shows the sum of the values of the two-dimensional array named `marks[][]` using `for` loop and enhanced `for` loop together.

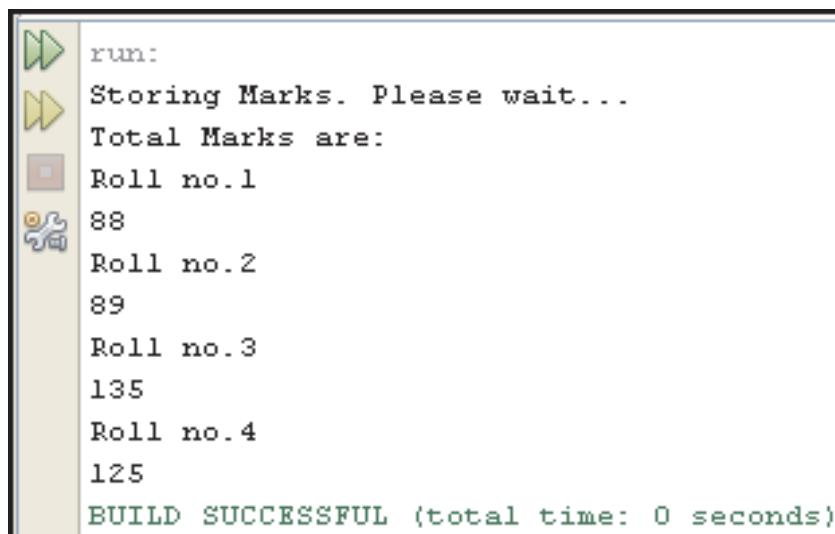


Figure 5.7: Sum of Values of a Two-dimensional Array

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 5.5.

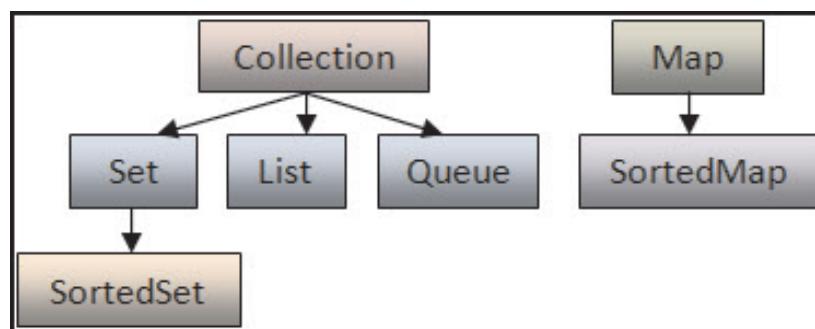


Figure 5.8: The Core Collection Interfaces

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 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 null, 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 null, 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 -1.
<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 -1.
<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 null, 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 one of following 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:

```
ArrayListmarks = 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 commandline 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:l
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)); // output: JAVA 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 **StringBuilders** 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 **StringBuilders** is instantiated in line 1. The object `objStrBuild` is used to invoke the `displayStrings()` method.

Figure 5.11 shows the output of the `StringBuilders.java` class.

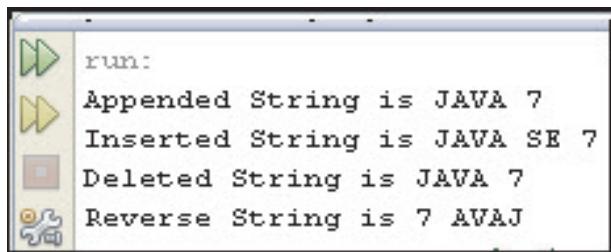


Figure 5.11: Output of `StringBuilders.java` Class

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.

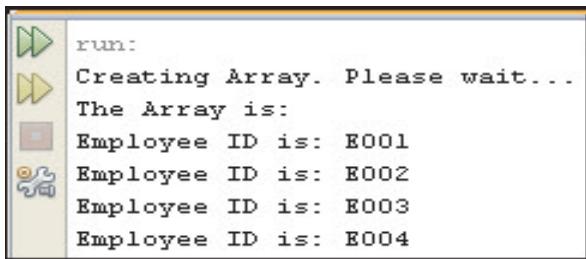


Figure 5.12: Output of `StringArray.java` Class

5.3.6 Command Line Arguments

A user can pass any number of arguments to a Java application at runtime from the OS command line. The `main()` method declares a parameter named `args[]` which is a `String` array that accepts arguments from the command line. These arguments are placed on the command line and follow the class name when it is executed. For example,

```
java EmployeeDetail Roger Smith Manager
```

Here, `EmployeeDetail` is the name of a class and `Roger`, `Smith`, and `Manager` are command line arguments which are stored in the array in the order that they are specified. When the application is launched, the runtime system passes the command line arguments to the application's `main()` method using a `String` array, `args[]`. Note, that the array of strings can be given any other name. The `args[]` array accepts the arguments and stores them at appropriate locations in the array. Also, the length of the array is determined from the number of arguments passed at runtime. The arguments are separated by a space.

The basic purpose of command line arguments is to specify the configuration information for the application.

As already learnt, that the `main()` method is the entry point of a Java program, where objects are created and methods are invoked.

The `static main()` method accepts a `String` array as an argument as depicted in Code Snippet 17.

Code Snippet 17:

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

The parameter of the `main()` method is a `String` array that represents the command line arguments. The size of the array is set to the number of arguments specified at runtime. All command line arguments are passed as strings.

Code Snippet 18 demonstrates an example of command line arguments.

Code Snippet 18:

```

package session5;

public class CommandLine {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Check the number of command line arguments
        if(args.length==3) {
            // Display the values of individual arguments
            System.out.println("First Name is "+args[0]);
            System.out.println("Last Name is "+args[1]);
            System.out.println("Designation is "+args[2]);
        }
        else {
            System.out.println("Specify the First Name, Last Name, and
Designation");
        }
    }
}

```

The class **CommandLine** consists of the `main()` method. Within `main()` method, the `if` statement checks whether the number of command line arguments specified is equal to 3. If so, it prints the values present in the respective elements of the array such as `args[0]`, `args[1]`, and `args[2]`; otherwise it prints the message specified in the `else` part.

To run the program with command line arguments at command prompt, perform following steps:

1. Open the command prompt.
2. Compile the Java program by writing following statement:

```
javac CommandLine.java
```

3. Execute the program by writing following statement:

```
java CommandLine Roger Smith Manager
```

To run the program with command line arguments using NetBeans IDE, perform following steps:

1. Right-click the project name in the **Projects** tab and click **Properties**. The **Project Properties** dialog box is displayed.
2. Click **Run** in the **Categories** pane to the left. The runtime properties are displayed in the right pane as shown in figure 5.13.

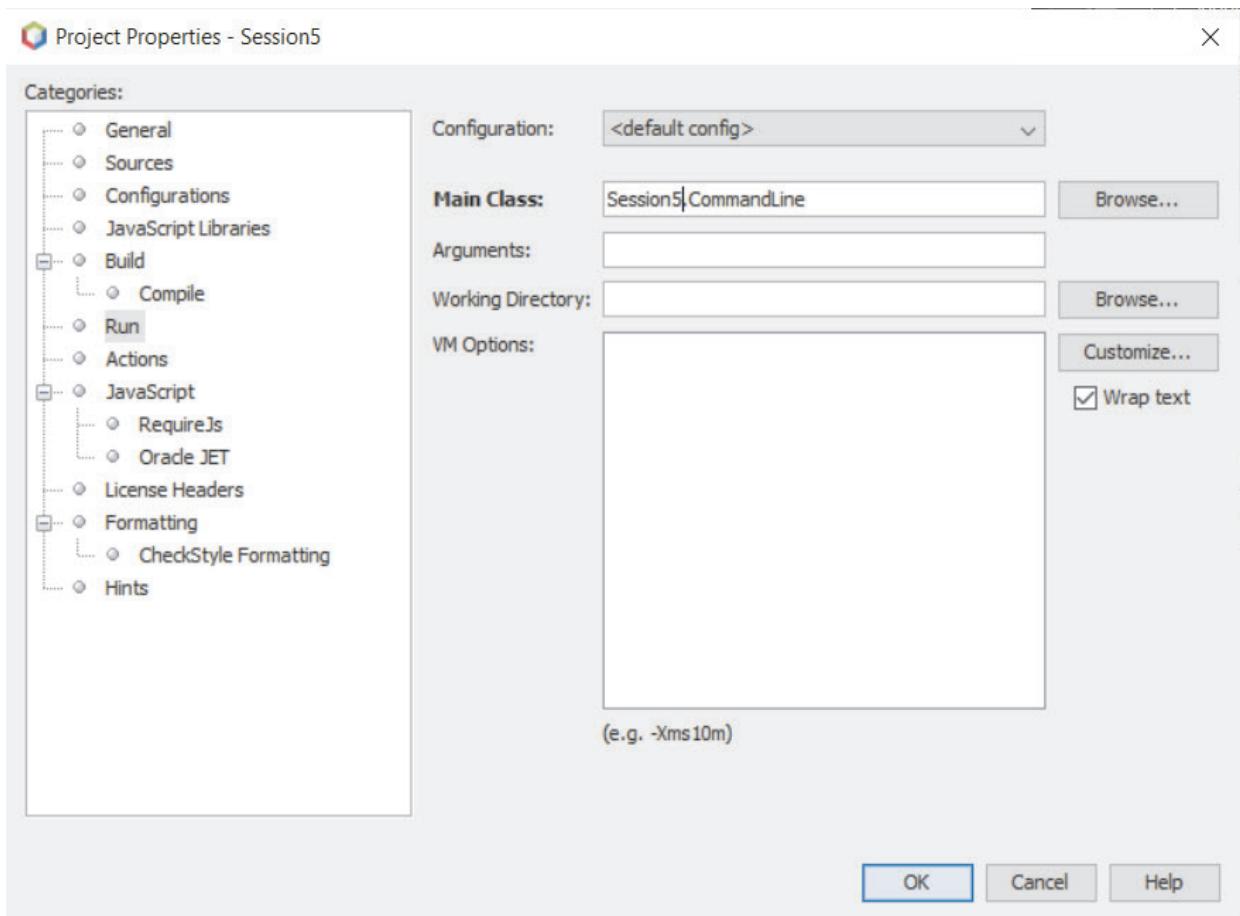


Figure 5.13: Runtime Settings of a Java Program

3. Type the arguments **Roger**, **Smith**, and **Manager** in the **Arguments** box as shown in figure 5.14.

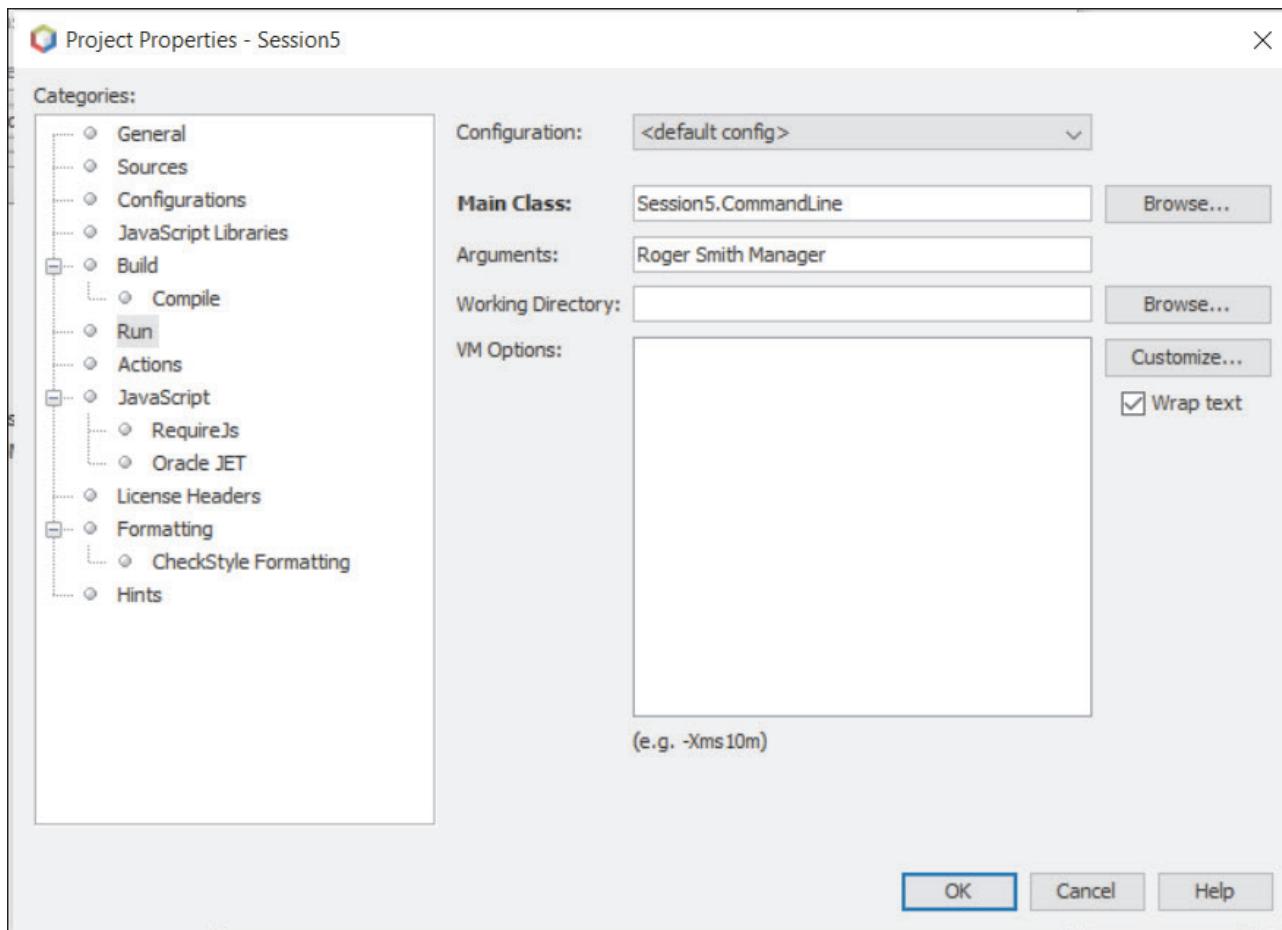


Figure 5.14: Specifying Command Line Arguments

4. Click **OK** to close the **Project Properties** dialog box.
5. Click **Run** on the toolbar or press **F6**. The command line arguments are supplied to the `main()` method and printed as shown in figure 5.15.

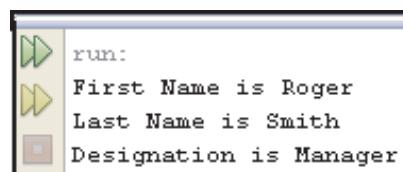


Figure 5.15: Output of `CommandLine.java` Class

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
byte	Byte
char	Character
float	Float
double	Double
int	Integer
long	Long
short	Short
boolean	Boolean

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	<code>byteValue()</code> – returns a byte value of the invoking object. <code>parseByte()</code> – returns the byte value from a string storing a byte value.	<pre>byte byteVal = Byte.byteValue(); byte byteVal = Byte.parseByte("45");</pre>
Character	<code>isDigit()</code> – checks if a character is a digit. <code>isLowerCase()</code> – checks if a character is a lower case alphabet. <code>isLetter()</code> – checks if a character is an alphabet.	<pre>if(Character.isDigit('4')) System.out.println("Digit"); if(Character.isLetter('L')) System.out.println("Letter");</pre>

Wrapper Class	Methods	Example
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 5.16.

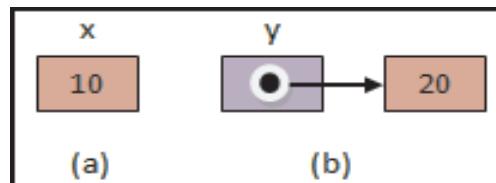


Figure 5.16: Primitive Type and Wrapper Type

It is clear from the figure 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 19 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 19:

```

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 5.17.

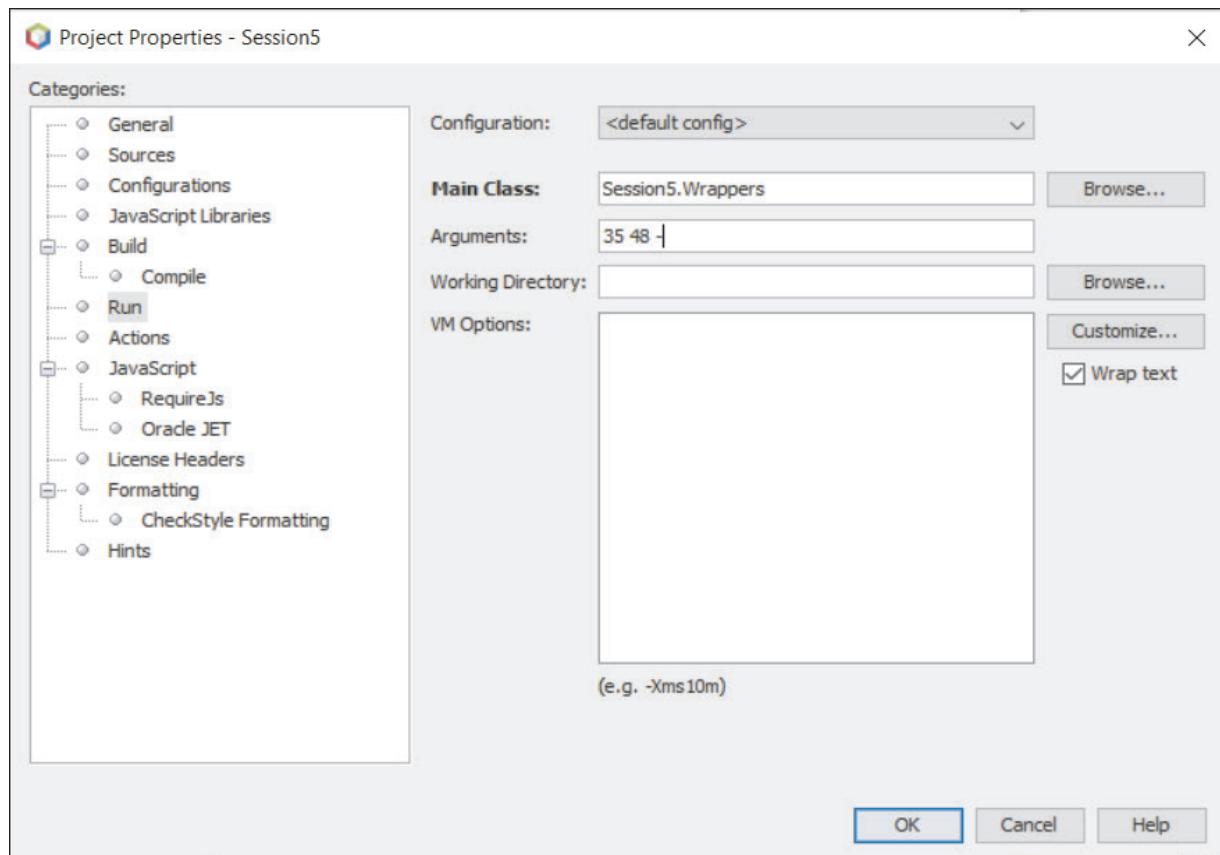


Figure 5.17: Specifying Command Line Arguments

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 5.18: 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 20 demonstrates an example of autoboxing and unboxing.

Code Snippet 20:

```
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 5.19 shows the output of the code.

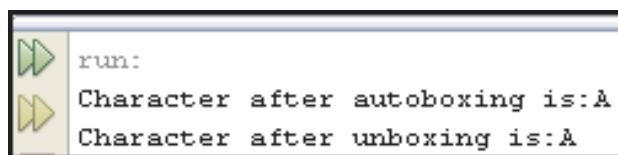


Figure 5.19: Output of `AutoUnbox.java` Class

Figure 5.19 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 Compact Strings

Compact String is one of the performance enhancements that was introduced in the JVM as part of JDK 9.

Until JDK 8, Java represented String objects as `char[]` because every character in Java is of two bytes since Java internally uses UTF-16. Many characters require two bytes to represent them, however, some characters require only one byte, (also referred to as LATIN-1 representation). Thus, there was potential to improve memory consumption and performance by changing the internal implementation of String. This led to the concept of **compact strings**.

In Java 9 and higher versions, the implementation of the String class was revised to be supported by a byte array instead of a char array. Whenever we create a String, if all its characters can be represented using a single byte (LATIN-1), a byte array will be used internally to save half of the space required. This saves memory and applications will perform better.

A final field named `coder` is used in the internal representation of String with a byte array as follows:

```

private final byte[] value;
/*can be LATIN1 = 0 or UTF16 = 1 */
private final byte coder;

```

Most of the String operations must check the `coder` value and dispatch to the specific implementation.

Consider some examples to understand the concept of compact strings. Code Snippet 21 shows code to illustrate the memory consumption by String before Java 9.

Code Snippet 21:

```
public class RegularStringDemo {
    public static void main(String[] args) {
        String str = new String("Demo Examples");
    }
}
```

Here, we create a String object with 13 characters and characters inside the object can be represented using one byte, which is nothing but LATIN-1 representation. If we run this code with JDK version 8 or earlier then, internally String will be represented as `char[]`. In this case though, we do not require `char[]` as each character could have been represented using one byte only. However, in JDK 8, instead of creating `byte[]`, a `char[]` will be created internally and for each character, two bytes will be assigned in the heap memory. This is nothing but wastage of heap memory.

The Compact String VM option is enabled by default in all versions upward of JDK 9. Since you are now using JDK 15, but want to test the code with JDK 8, you can disable it, we can use following as command line argument:

`+XX:-CompactStrings`

Code Snippet 22 shows another example.

Code Snippet 22:

```
public class CompactStringDemo {
    public static void main(String[] args) {
        String str1 = new String("Demo Examples");
        String str2 = new String("Demo Examples €");
    }
}
```

Consider that this code is compiled with JDK 9 or higher. Hence, `char[]` or `byte[]` will be created as per requirements for String objects. We created String object `str1` with 13 characters and object `str2` with 14 characters. Each character present inside object `str1` can be represented using one byte only. Hence, for object `str1`, one `byte[]` will be created. Now, for `str2`, we have one additional character apart from the characters present in object `str1`, namely, €. This € character cannot be represented using LATIN-1 character set. Here, we require two bytes to represent €. Therefore, Java will use UTF-16 to represent the characters inside `str2`. For object `str2`, internally an `char[]` will be created. Thus, instead of using two `char` arrays, only one will be used and the other will be a `byte` array, which in turn will save memory and improve performance.

5.6 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)	int marks = {65, 47, 75, 50};	(C)	int[] marks = {"65", "47", "75", "50"};
(B)	int[] marks = {65, 47, 75, 50};	(D)	int[] marks = {65}, {47}, {75}, {50};

5. Consider the following code snippet to add two numbers specified by user at command line:

```
public class Addition{
    public static void main(String[] args) {
        if(args.length==2) {
            int sum=args[0]+args[1]; // line1
            System.out.println("Sum is:"+sum);
        }
        else{
            System.out.println("Specify two integers.");
        }
    }
}
```

The code is giving compilation error ‘Incompatible types’. What change must be made in line 1 to resolve the error?

(A)	int sum = (int)(args[0]) + (int)(args[1]);	(C)	String sum = args[0] + args[1];
(B)	int sum = args[0].toInteger() + args[1].toInteger();	(D)	int sum = Integer.parseInt(args[0]) + Integer.parseInt(args[1]);

6. 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

7. Which of the following are benefits of compact string feature in Java 9 and higher versions?

(A)	Reducing memory consumption	(C)	Cheaper applications
(B)	Improving performance	(D)	Less lines of code

5.6.1 Answers

1.	C
2.	A
3.	D
4.	B
5.	D
6.	B
7.	A, B

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.
- Compact strings are a new feature in Strings in Java version 9 and higher that improve performance and reduce memory consumption.

Try it Yourself

1. 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 DLL, so that it can be loaded on the machine on which the Java program is executing. Each native method is dependent on the 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;
    /**
     * staticblock
     *
     */
}
```

```
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 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.

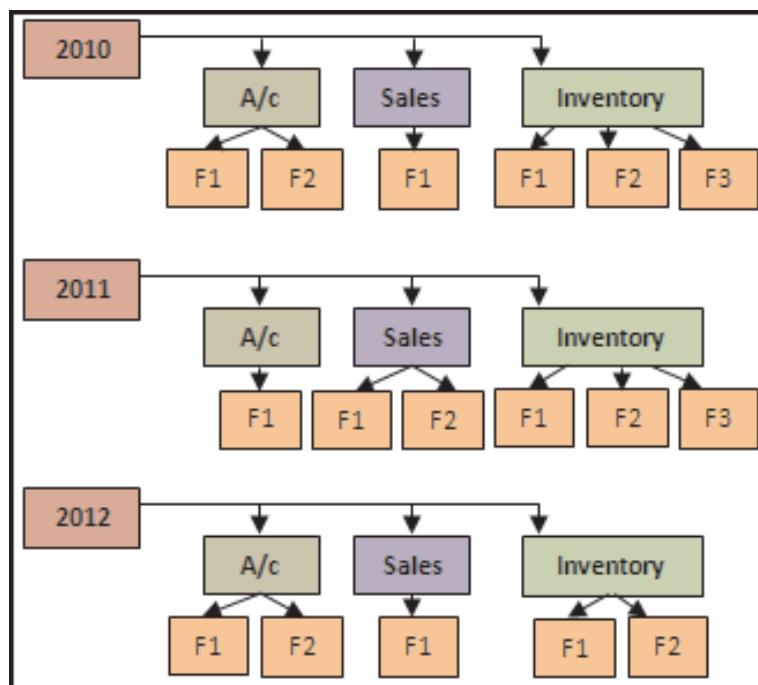


Figure 6.3: Files Organized in Specific Folders

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 example, if the domain name contains special characters such as hyphen, if the package name consists of a reserved Java keyword such as `char`, or if the package name begins with a digit or some other character that is illegal to use as the beginning of a Java package name. In such a case, it is advisable to use an underscore as shown in table 6.1.

Domain Name	Suggested Package Name
<code>sample-name.sample.org</code>	<code>org.sample.sample_name</code>
<code>sample.int</code>	<code>int_.sample</code>
<code>007name.sample.com</code>	<code>com.sample._007name</code>

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.

- 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.

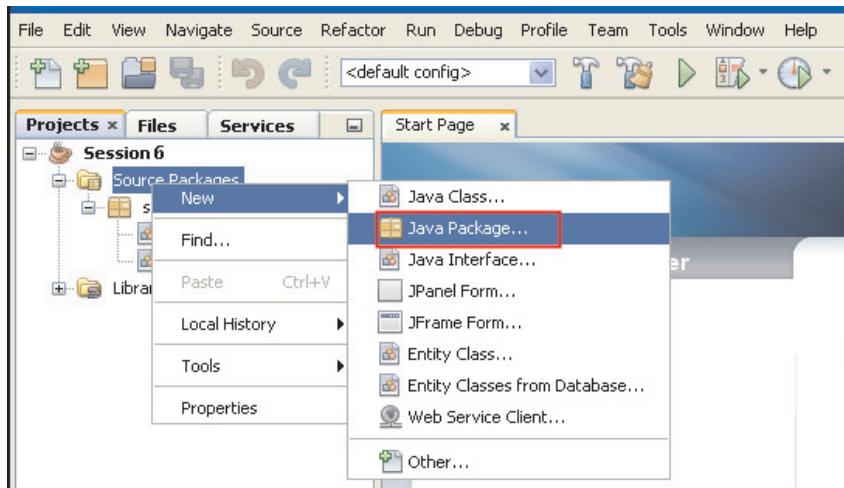


Figure 6.4: Creating a New Java Package

- Type **userpkg** in the **Package Name** box of the **New Java Package** dialog box that is displayed.
- Click **Finish**. The **userpkg** package is created as shown in figure 6.5.

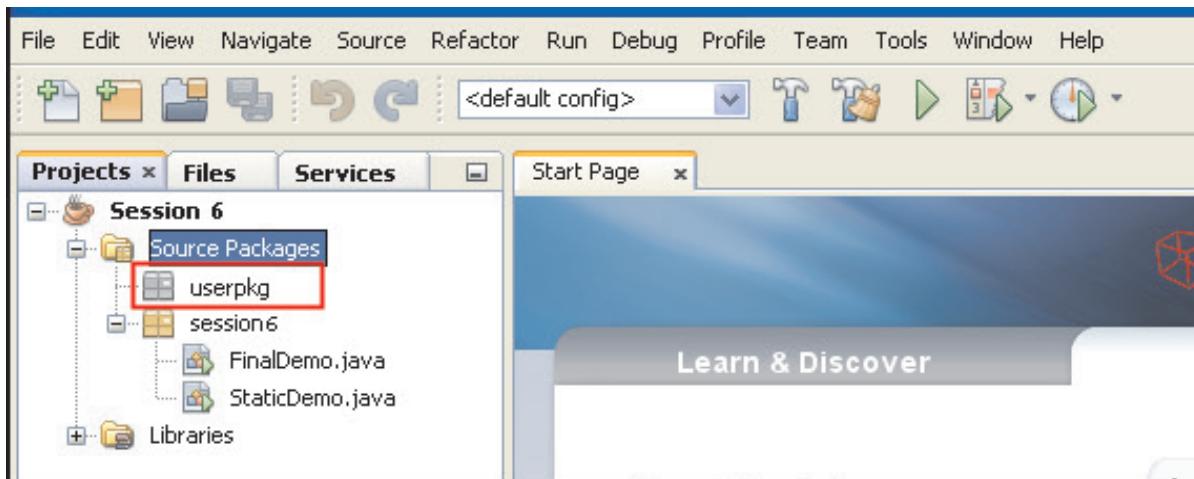


Figure 6.5: userpkg Package Created

- Right-click **userpkg** and select **New** → **Java Class** to add a new class to the package.
- Type **UserClass** as the **Class Name** box of the **New Java Class** dialog box and click **Finish**.
- 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.

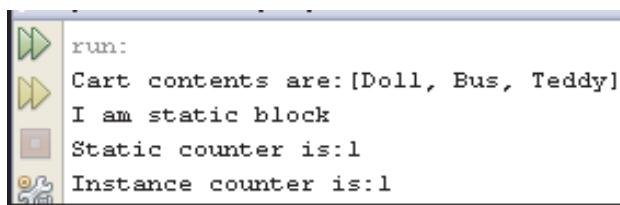


Figure 6.6: Output of `UserClass.java` Class

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. Java 3D and Java Mail are examples of extensions developed by Sun Microsystems. 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.

The 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 of the other options, apart from `cf`, available with the `jar` command are listed in table 6.2.

Option	Description
<code>v</code>	Produces VERBOSE output on stdout while the .jar is being built. The output displays the name of each of the files that are included in the .jar file.
<code>0 (zero)</code>	Indicates that the .jar file must not be compressed.
<code>M</code>	Indicates that the default manifest file must not be created.
<code>m</code>	Allows inclusion of manifest information from an existing manifest file. <code>jar cmf existing-manifest-name jar-file-name input-file-name(s)</code>
<code>-c</code>	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.

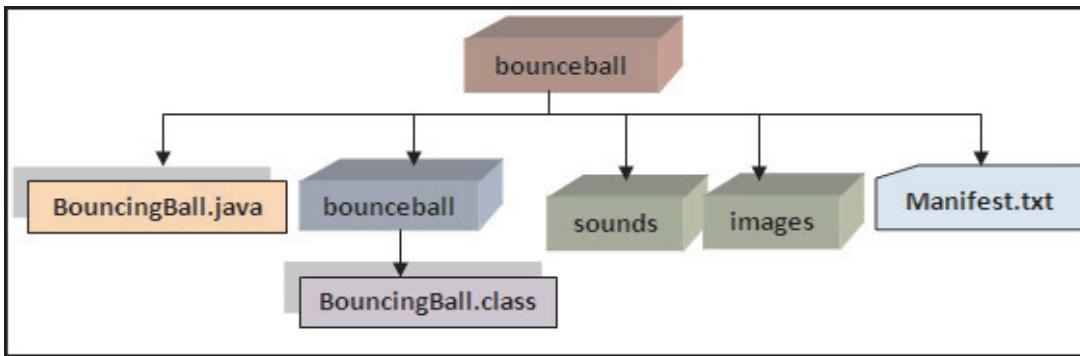


Figure 6.7: BouncingBall Application

Figure 6.7 shows the **BouncingBall** application with the source file `BouncingBall.java`, class file `BouncingBall.class`, `sounds` directory, `images` directory, and `Manifest.txt` file. `sounds` and `images` are subdirectories that contain the sound files and animated .gif images used in the application.

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\jdk1.7.0\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.

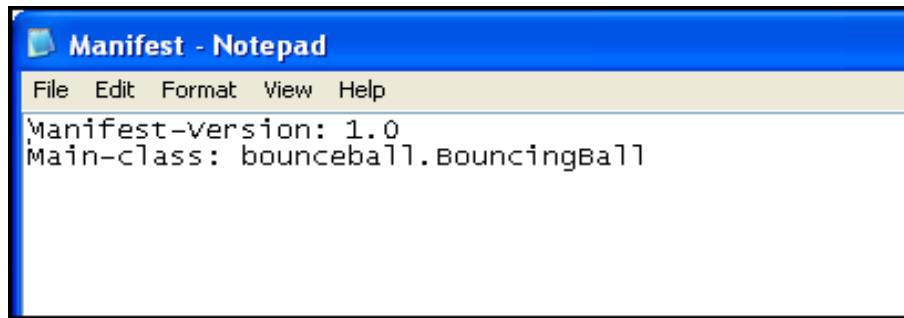


Figure 6.8: Manifest.txt File

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.

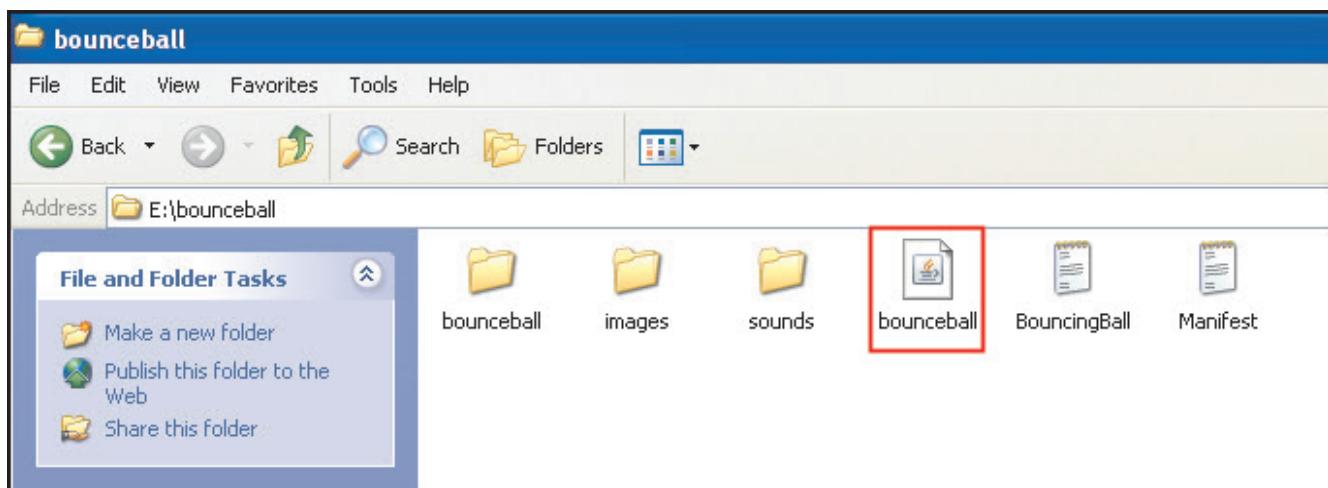


Figure 6.9: bounceball.jar File Created in Source Folder

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**. Further, the name of the class file is provided with its location as **bounceball/BouncingBall.class** followed by the directory names **sounds** and **images** so that the respective files under these directories are also included in the **.jar** file.

- 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 the entire series of steps for creation of **.jar** file with the verbose output and the final output after **.jar** file execution.

```
C:\WINDOWS\system32\cmd.exe
E:\bounceball>set path="C:\Program Files\Java\jdk1.7.0\bin"
E:\bounceball>javac -d . BouncingBall.java
E:\bounceball>jar cvmf Manifest.txt bounceball.jar bounceball/BouncingBall.class
adding: bounceball/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, the use of the option ‘v’ will shows the verbose output of all the 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	jar cf jar-file-name input-file-name(s)
To view contents of a .jar file	jar tf jar-file-name
To extract contents of a .jar file	jar xf jar-file-name
To run the application packaged into the .jar file. Manifest.txt file is required with Main-class header attribute.	java -jar jar-file-name

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 commandline 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.

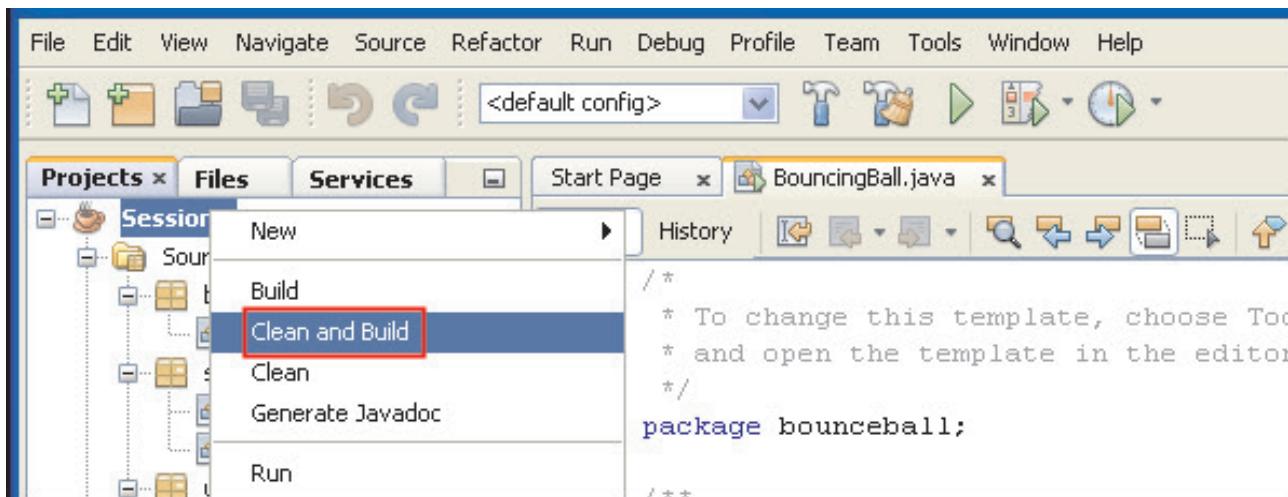


Figure 6.11: Clean and Build an Application

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.

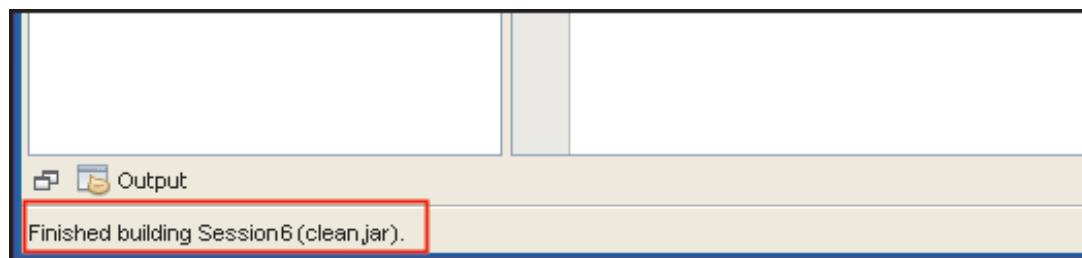


Figure 6.12: Status Message After .jar Generation

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.

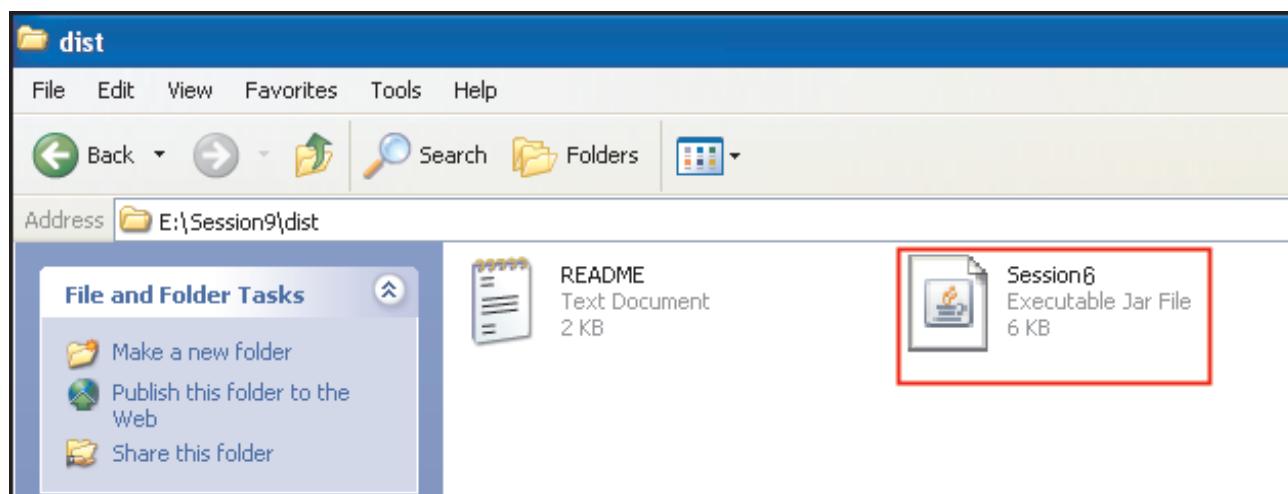


Figure 6.13: Session6.jar File

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 the 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 the 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

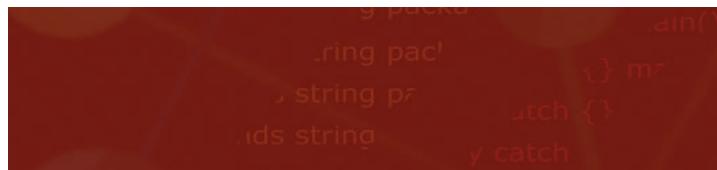
6.5.1 Answers

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

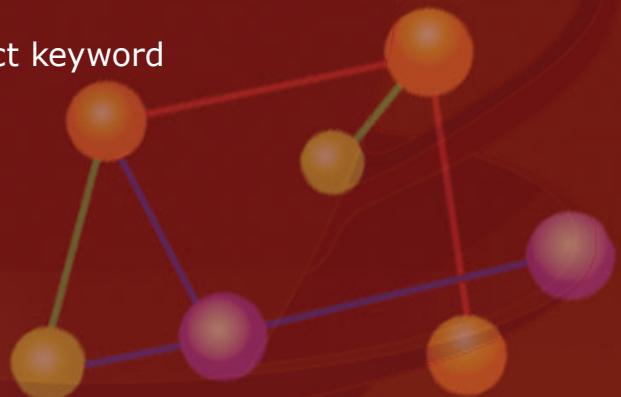
Inheritance and Polymorphism

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
- Differentiate 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.

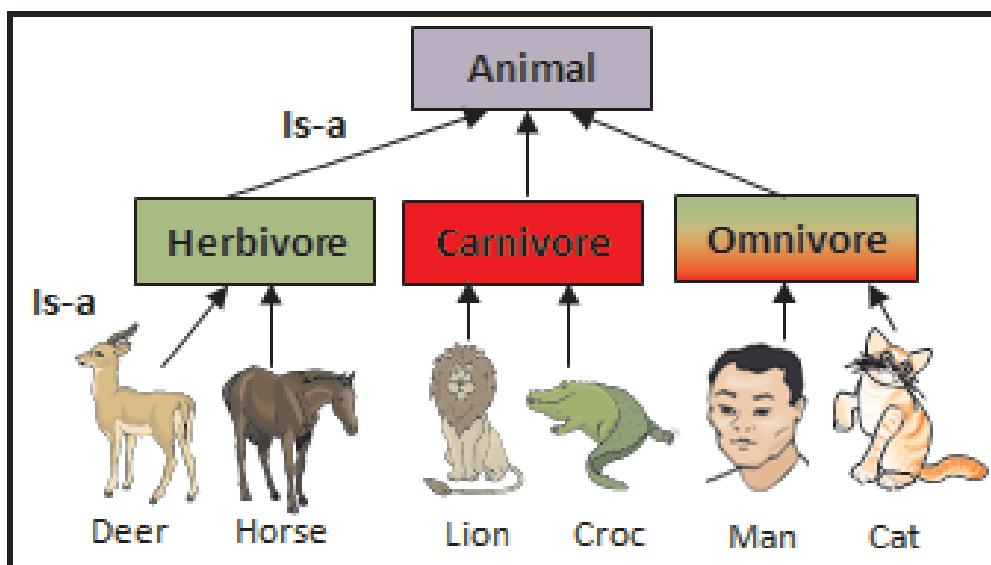


Figure 7.1: Is-a Relationship Between Real World Entities

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 and behavior. The classes Deer, Horse, Lion, and so on inherit properties from the classes Herbivore, Carnivore, and so on.

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.

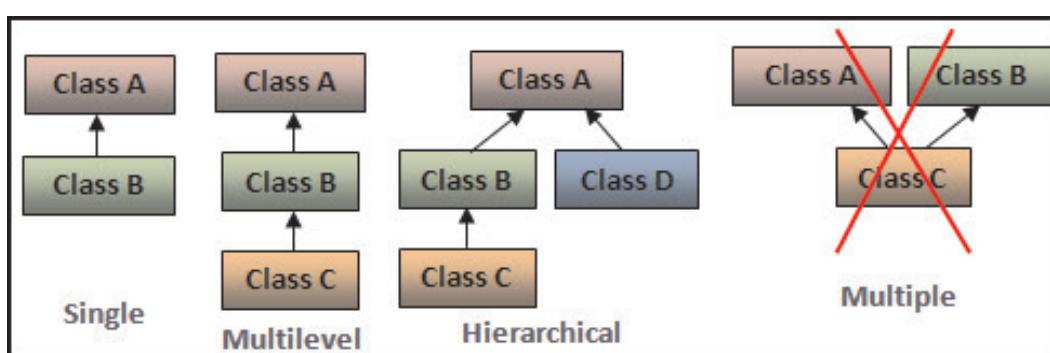


Figure 7.2: Types of Inheritance in Java

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 just like 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) x
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 superclass accelerate() method
    super.accelerate(speed);
    System.out.println("Maximum acceleration:"+ speed + " kmph");
}

}

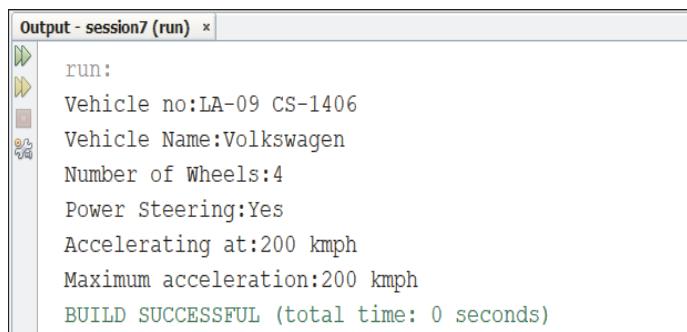
/***
 * Define the TestVehicle class
*/
public class TestVehicle {

    /**
     * @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.

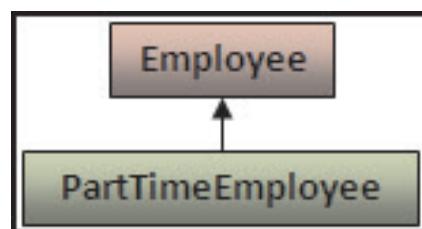


Figure 7.7: Single Inheritance Between Employee and PartTimeEmployee

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 superclass 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 run time 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.

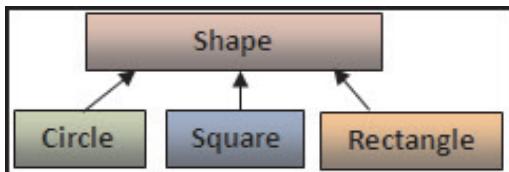


Figure 7.9: Using Abstract Class

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.

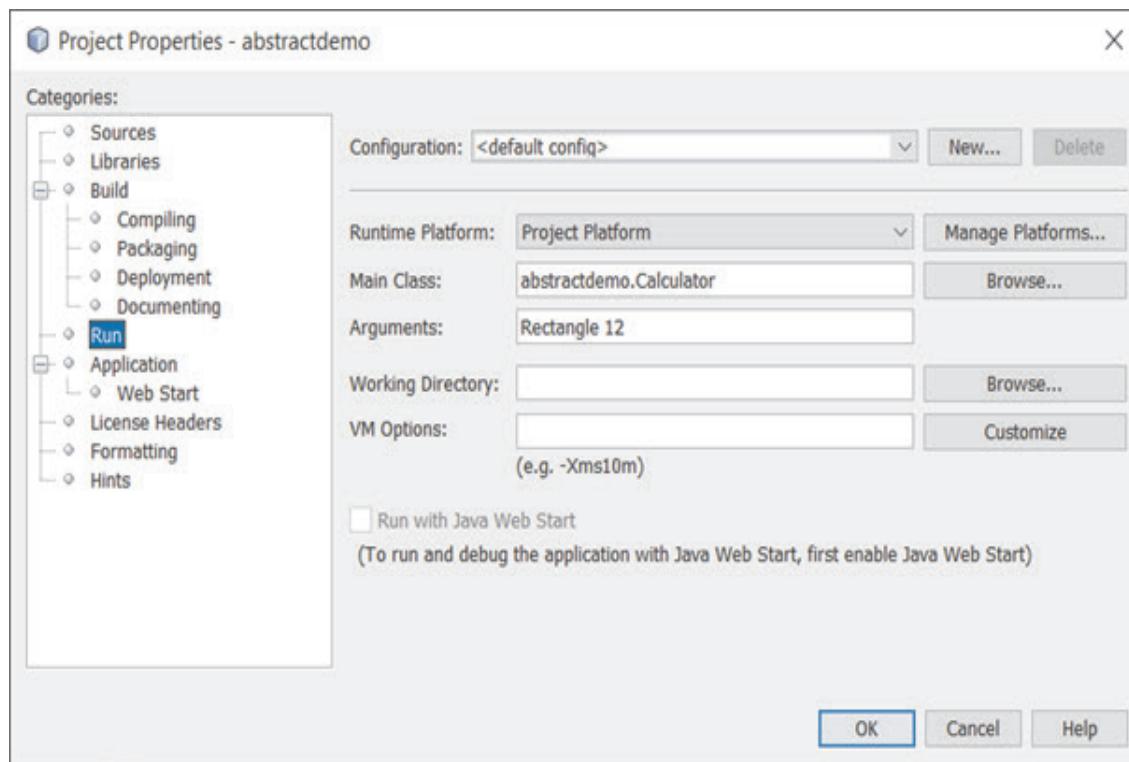
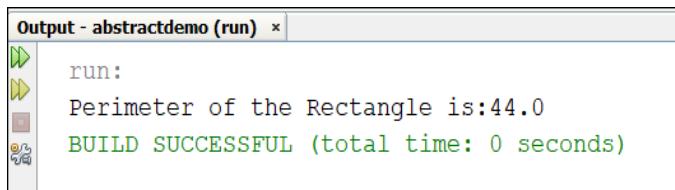


Figure 7.10: Setting Command Line Arguments

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 the 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 the 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 the 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

- 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
- Explain the purpose of interfaces
- Explain implementation of multiple interfaces
- Describe private methods in interfaces
- Describe Abstraction
- Explain Nested class
- Explain Member class
- Explain Local class
- Explain Anonymous class
- Describe 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 I Enumerable
interface I Comparable
```

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.

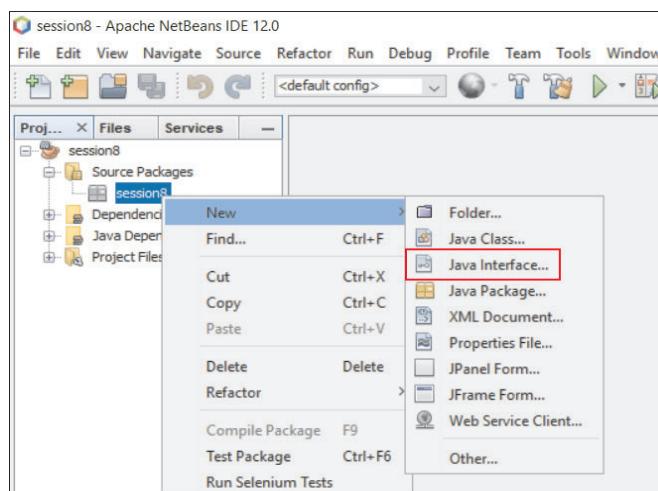


Figure 8.1: Creating a New Java Interface

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.

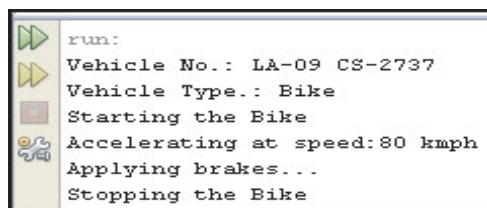


Figure 8.2: Output of `TwoWheeler.java`

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.

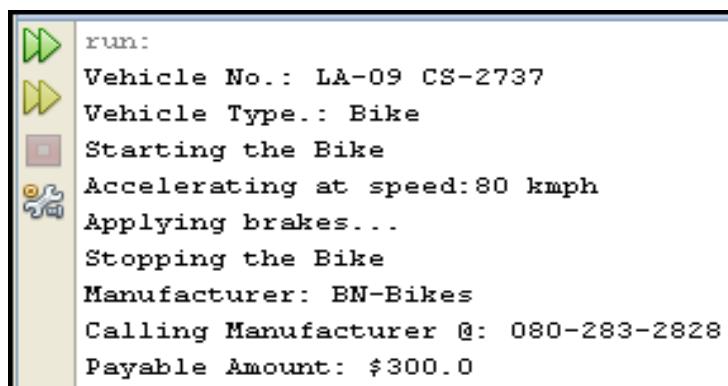


Figure 8.3: Output of TestVehicleNew.java Class

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

Java 9 introduced private methods and private static method in interfaces. These methods are visible only inside the interface that they are declared in, so it is recommended to use private methods 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.

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**, we define a method **printInterface** 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, tyres, ignition key, and may 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 private, protected, and so on.	Members of an interface are public by default.
An abstract class is inherited using extends keyword.	An interface is implemented using implements 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.

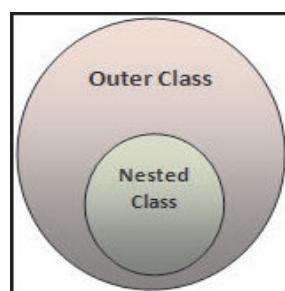


Figure 8.4: Nested Class

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 innerclass using outerclass 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.

```
run:
Connecting to Server at:101.232.28.12:8080
BUILD SUCCESSFUL (total time: 1 second)
```

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.

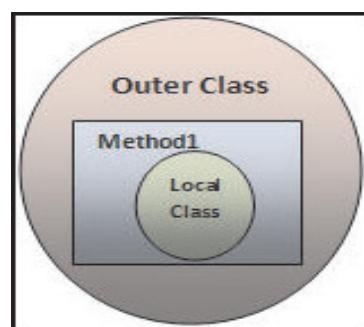


Figure 8.6: Local Inner Class

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.

```
run:
Getting Rank of employee: E001
Employee rank is:A
Status: Eligible for upgrade
```

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.

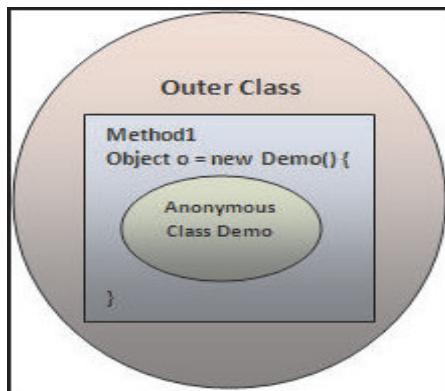


Figure 8.8: Anonymous Inner Class

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 is verified and accordingly 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.

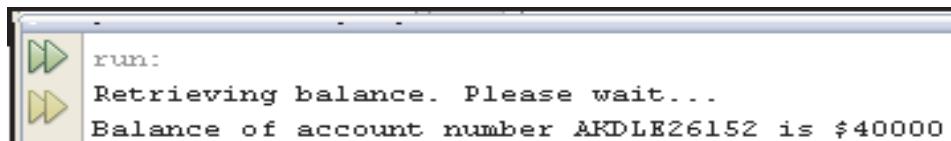


Figure 8.9: Output of `Authenticate.java` That Uses Anonymous Class

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.

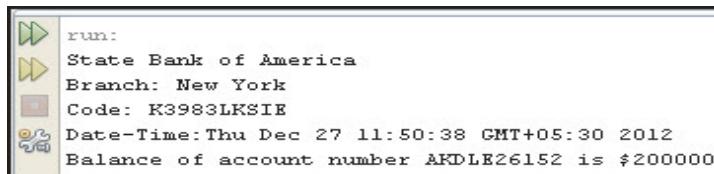


Figure 8.10: Output of `AtmMachine.java` That Uses Static Nested Class

Notice that the output of date and time shows the default format as specified in the implementation of the `getTime()` method. The format can be modified according to the user requirement using the `SimpleDateFormat` class of `java.text` package. `SimpleDateFormat` is a concrete class used to format and parse dates in a locale-specific manner. `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.

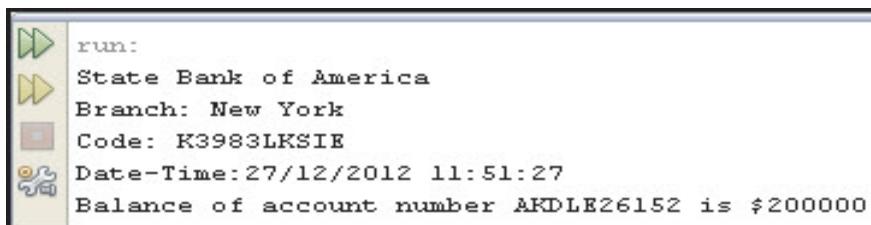


Figure 8.11: Output of Modified `AtmMachine` Class Using `SimpleDateFormat`

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 the 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 the 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	(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 the 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

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.drawString("Hello World", 100, 100);
    }
}

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[]) {
        double angle = Double.parseDouble(args[0]);
        System.out.println("Rotating Circle by " + angle + " 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
- ➔ Describe the Exception class
- ➔ Describe exception handling
- ➔ Explain try-catch block
- ➔ Explain finally block
- ➔ Explain execution flow of exceptions
- ➔ Explain 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. An exception can occur for different reasons such as when the user enters invalid data, a file that must be opened cannot be found, a network connection has been lost in the middle of communications, or the JVM has run out of memory. 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.

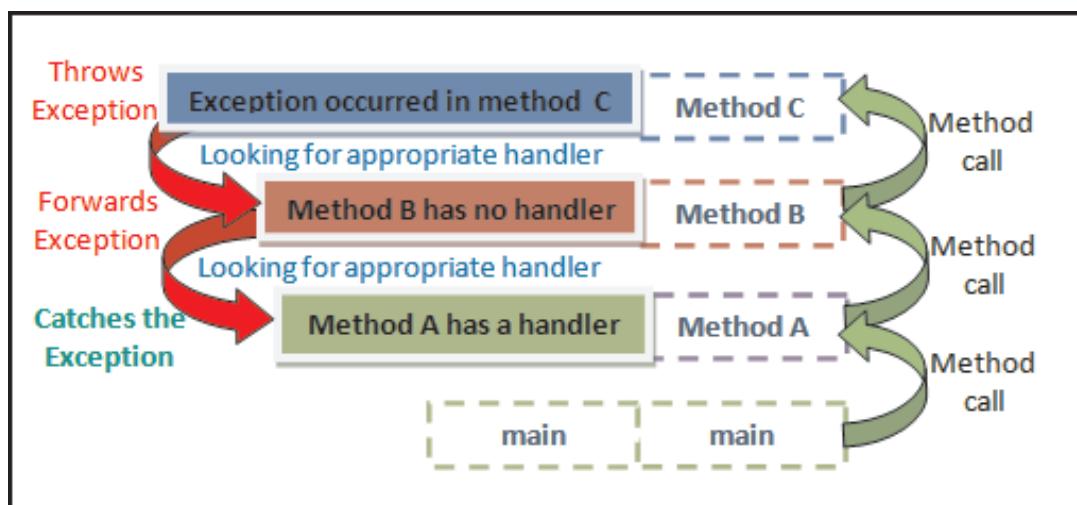


Figure 9.1: Runtime Call Stack and Exception Handling

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, trying to open a file that does not exist, loss of network connection in the middle of a transaction, or the JVM runs out of memory while performing a task. 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.

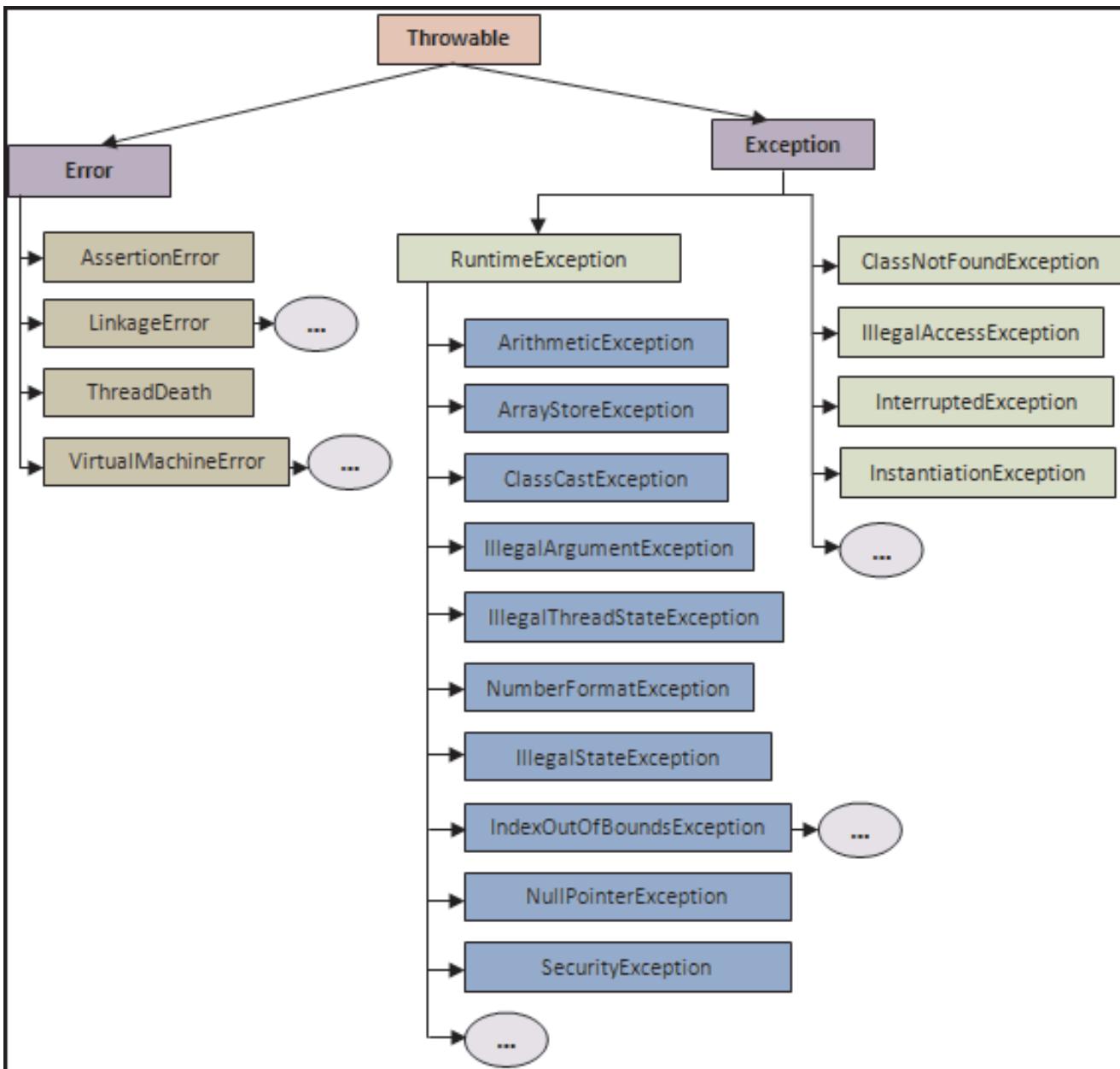


Figure 9.2: `Throwable` Class Hierarchy

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.
<code>InterruptedException</code>	Occurs when a thread is interrupted.
<code>NoSuchMethodException</code>	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 null 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> .

Exception Class Constructor	Description
Exception(String message, Throwable cause)	Constructs a new exception with error message set to the specified strings message and cause.
Exception(Throwable cause)	Constructs a new exception with the specified cause. The error message is set as per the evaluation of cause == null?null:cause.toString(). That is, if cause is null, it will return null, else it will return the String representation of the message. The message is usually the class name and detail message of cause.

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
public String getMessage()	Returns the details about the exception that has occurred.
public Throwable getCause()	Returns the cause of the exception that is represented by a Throwable object.
public String toString()	If the Throwable object is created with a message string that is not null, it returns the result of getMessage() along with the name of the exception class concatenated to it. If the Throwable object is created with a null message string, it returns the name of the actual class of the object.
public void printStackTrace()	Prints the result of the method, toString() and the stack trace to System.err, that is, the error output stream.
public StackTraceElement [] getStackTrace()	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.
public Throwable fillInStackTrace()	Fills the stack trace of this Throwable 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(ArithmeticException e) { //catch block for ArithmeticException
        // 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 commandline arguments
     */
    public static void main(String[] args)
    {
        // Check the number of commandline 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) ×
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 1, 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.

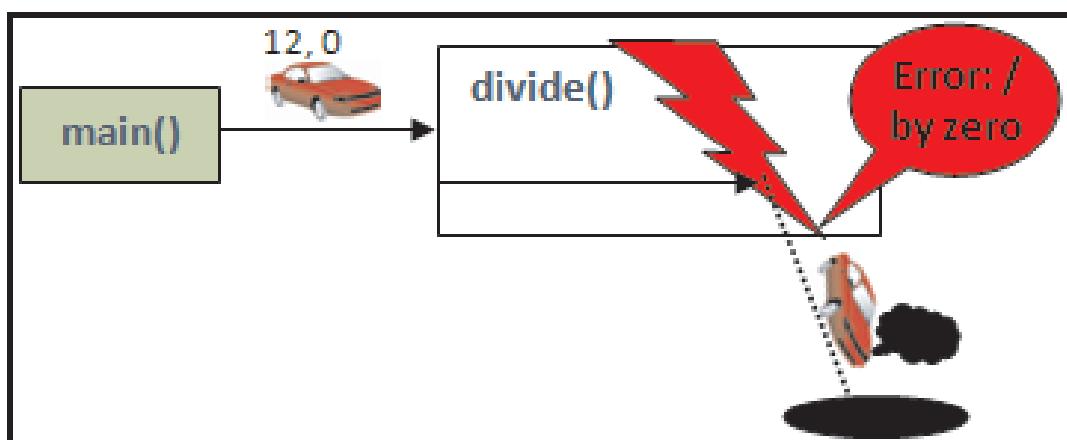


Figure 9.4: Execution Without try-catch Block

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.

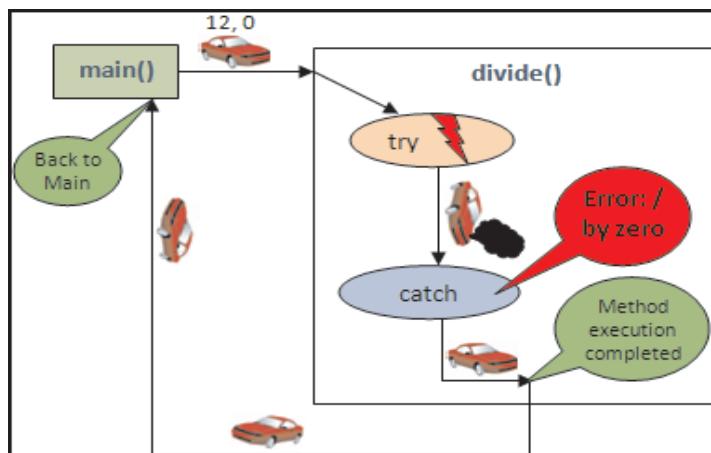


Figure 9.5: Execution of try-catch Block

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());
            }
        }
    }
}

```

```

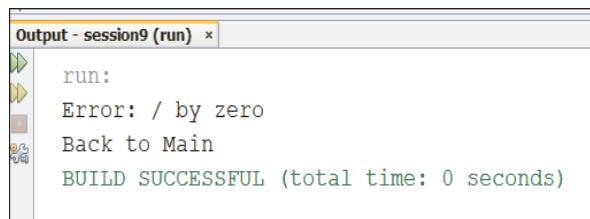
        else{
            System.out.println("Usage: java Mathematics <number1><number2>");
        }
        System.out.println("Back to Main");
    }
}

```

The method `divide(int,int)` now includes the `throws` clause that informs the calling method that `divide(int,int)` may throw an `ArithmaticException`. Within `divide(int,int)`, the code checks for the value of `num2`. If it is equal to zero, it creates an instance of `ArithmaticException` 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 `ArithmaticException` 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.



The screenshot shows the 'Output - session9 (run)' window. It displays the following text:
 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 `ArithmaticException`. 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.

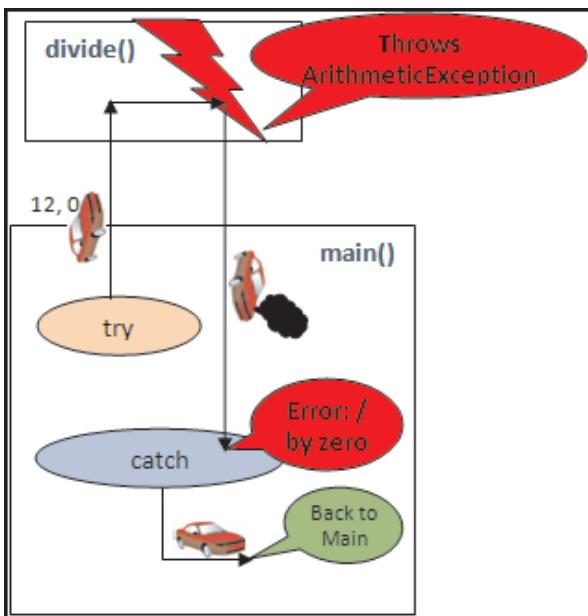


Figure 9.7: Execution of Code Using `throw` and `throws` Keywords

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]);
                System.out.println("Division is: " + num3);
            }
            catch (ArithmaticException e) { // Catch the ArithmaticException
                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, `ArithmaticException` 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.

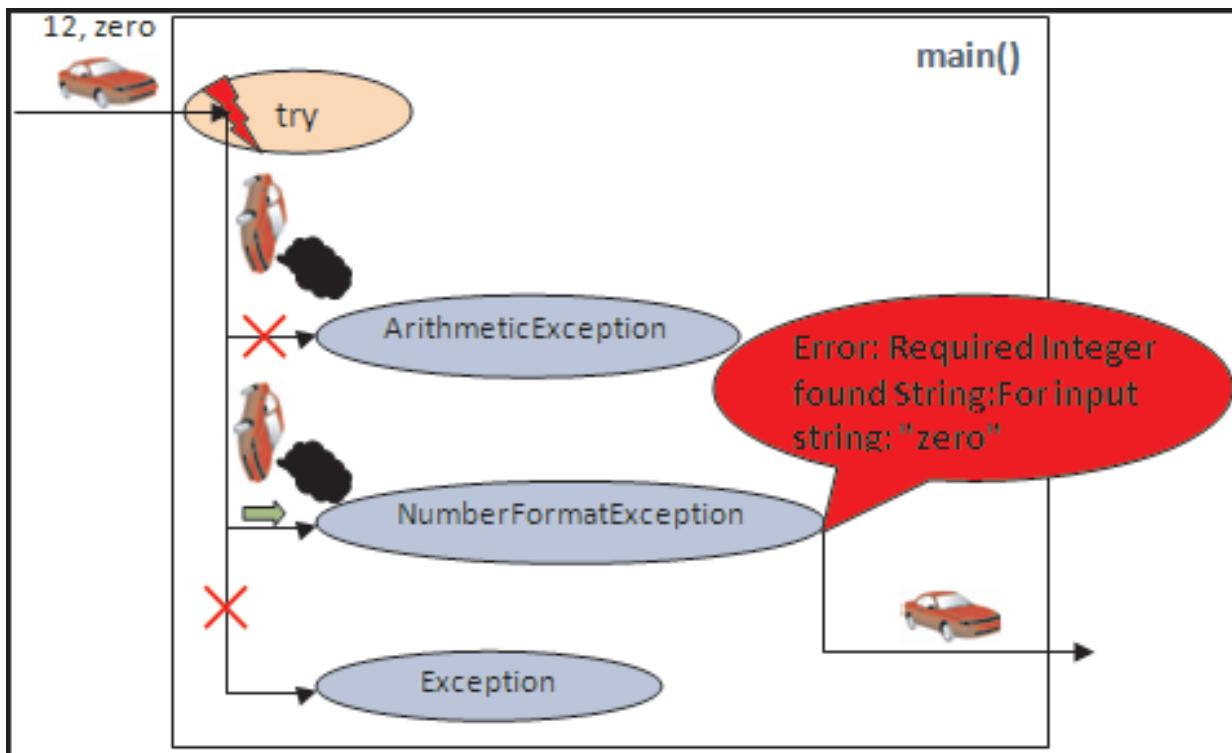


Figure 9.10: Execution of Code Using Multiple `catch` Blocks

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());
    }

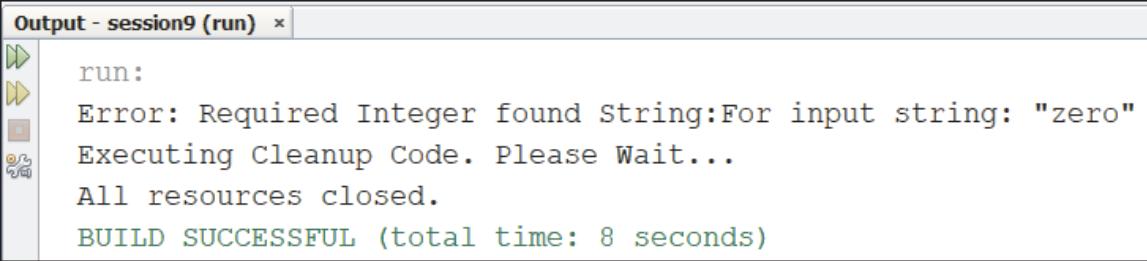
    catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }

    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.

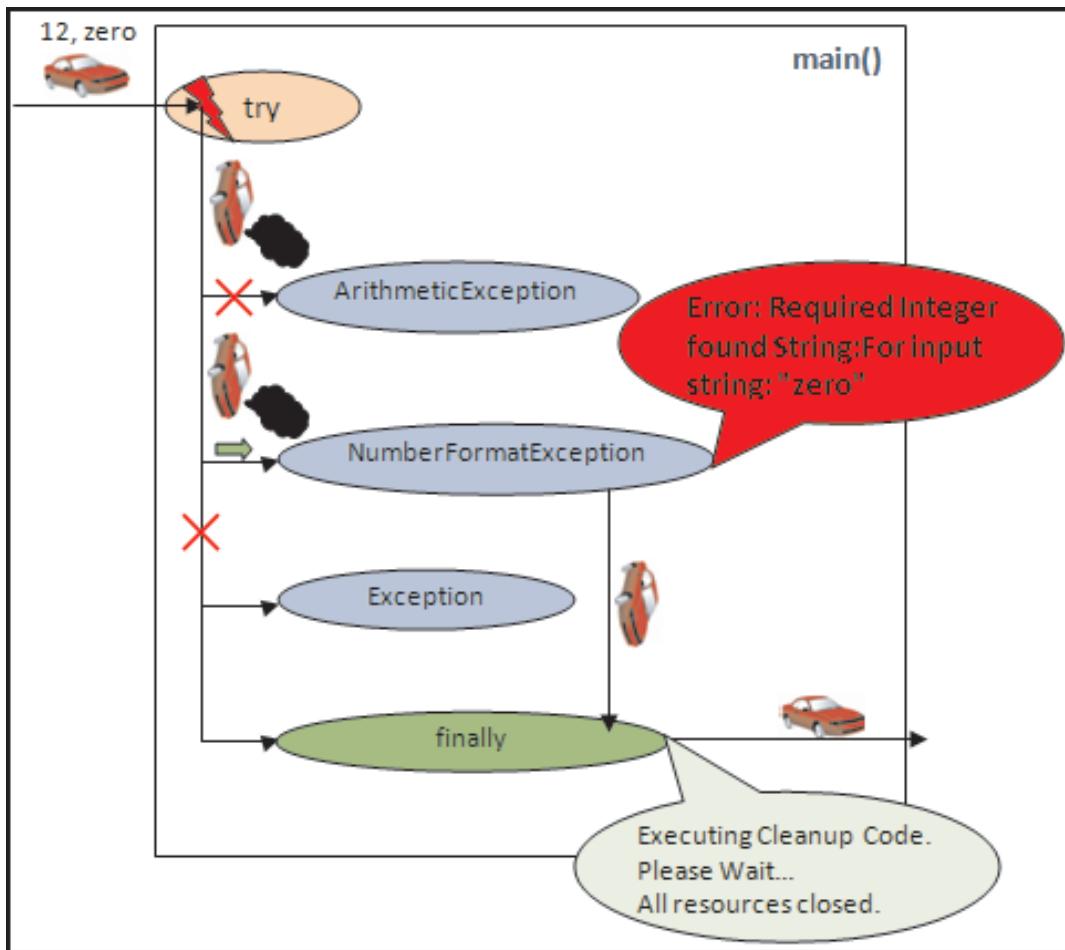


Figure 9.12: Execution of Code Using `finally` Block

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 the 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 the 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 the 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 the following message:

Unreachable array index

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 8 and later versions. Finally, the session describes support for backward compatibility in the new API.

In this Session, you will learn to:

- Explain classes of the Date and Time API in Java 8 and later versions
- 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 to commute to work, a bus to go to college or school, completing tasks at home, or delivering orders to customers, many such everyday tasks require us to monitor and track time. 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.

However, a new Date-Time API is introduced from Java 8 onwards, which offers a solution for many unaddressed drawbacks in the earlier API.

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

Class
Period
Year
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 ensuing 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 an 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 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);
// current local time

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 InstC1 {
    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. This enumeration will be described in detail in a later section.

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: 25

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 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 (): 2021
 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("2021-02")
            .getMonthValue()); // To display only the month value
        System.out.println("Year alone:"+YearMonth.parse("2021-02"
            .getYear());// to display the year value alone
        System.out.println("This year is a Leap year:"
            +YearMonth.parse("2021-02").isLeapYear());// leap year check
    }
}
```

```
}
```

Output:

The Present Year Month: 2021-02

Month alone: 2

Year alone: 2021

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 as shown in Code Snippet 19. `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:

```
ZoneId sampleZoneIdA = ZoneId.of("America/New_York");
ZoneId sampleZoneIdB = 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("2021-02-03T10:15:30+08:00[Asia/Singapore]");
        System.out.println("Present day of the year:" + sampleZoDT.getDayOfYear());
        System.out.println("Present year:" + sampleZoDT.getYear());
    }
}
```

Output:

2021-02-06T06:03:51.787+08:00[Etc/UTC]

Present day of the year: 24

Present year: 2021

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.

The Date-Time API in Java SE 8 onwards supports several new and useful enumerations.

`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 java8enum = new EnumDateCalculation();
```

```

        java8enum.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: 2021-02-16
After 2 weeks: 2021-03-02
After 2 months: 2021-04-16
After 2 years: 2023-02-16
Date after twenty years: 2041-02-16

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: 2021-02-16

Next Wednesday on: 2021-02-17

Second Sunday of the month on: 2021-02-14

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(2021, Month.MAY, 5);
        LocalDate randomDateB = LocalDate.of(2021, 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 2021-05-05 is 2021-05-07

Upcoming Odd Day for 2021-05-07 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= Tue Feb 16 20:52:48 EDT 2021

Desired Current Local date= 2021-02-16T20:52:48.932

Desired Current Zoned date= 2021-02-16T07:32:58.769-04:00[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 samplDate = 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("2021-02-16T10:15:30+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: 2021-02-16T10:15:30+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 01 March 2021.

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 01 March 2021
Stream<LocalDate> dates = LocalDate.now().datesUntil(LocalDate.
parse("2021-03-01"));
dates.forEach(System.out::println);
}
```

The code prints all the dates between current date and 01 March 2021. 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:

2021-02-15

2021-02-16

2021-02-17

2021-02-18

2021-02-19

2021-02-20

2021-02-21

2021-02-22

2021-02-23

2021-02-24

2021-02-25

2021-02-26

2021-02-27

2021-02-28

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`?

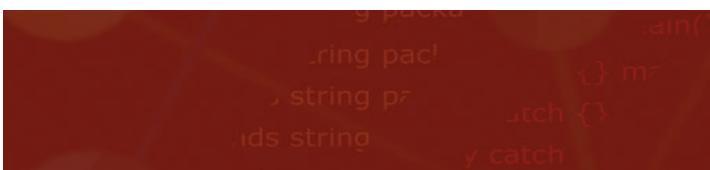
(A)	plusDays()	(C)	getDayofMonth()
(B)	ofInstant()	(D)	toNanos()

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

Annotations and Base64 Encoding

Welcome to the Session, **Annotations and Base64 Encoding**.

This session explains annotations and the predefined annotation types. It also defines Type annotation and repeating annotations in Java 8. Finally, this session describes about the Base64 encoding.

In this Session, you will learn to:

- Explain declaring an annotation type in Java
- Describe predefined annotation types
- Explain Type annotations
- Explain Repeating annotations
- Describe Base64 encoding



11.1 Introduction

Annotations or metadata processing in Java source code was introduced in Java for the first time in version 5.

Annotations are comments, notes, remarks, or explanations. In Java, annotations help to associate this additional information (also called metadata) to the program elements. Annotations affect the way a program is treated by tools and libraries which in turn can affect program semantics. Annotations can be read from source files or class files. Comments in code are replaced by annotations in many places.

Annotation represents the specific use of the type. In other words, annotation can be compared to an instance of the class. An annotation declaration consists of '@' (at) followed by the annotation type.

Annotation type helps to define an annotation. This is used by the programmer when a custom annotation is created. An annotation type can be compared with a class. An annotation type definition consists of '@' (at) symbol, followed by the interface keyword and the annotation name.

Annotations can be added at the class level, field level, and method level. Java 5 allowed annotation processing during compilation of code as well as during execution.

Java SE 6 introduced an enhanced feature in annotations, called the Pluggable Annotation Processing API. This API helps application developers to write Customized Annotation Processor that can be plugged-in to the code dynamically to operate on the set of annotations that appear in a source file. Hence, this API is also known as pluggable.

Java 8 introduces changes to annotations that will help developers to create better code and with more accuracy for automated code analysis to ensure quality.

11.2 Uses of Annotations

Instructions to Java compiler can be given using built-in Java annotations. For example, you can provide build-time annotations that will be used for building the software project. If you use automatic build tools to build your projects, they can scan your code for certain specific annotations and create source code (or other files) based on the comments.

Although Java annotations will not be present in Java code after compilation, you can declare specific annotations to be available at runtime through the Reflection API and thus, provide instructions for the programs.

Most common uses of annotations are as follows:

- ➔ **Information for the compiler:** Compiler uses annotations to produce notification or errors based on different rules. For example, consider the `@FunctionalInterface` annotation. This annotation helps the compiler to confirm the annotated class and verify whether it is a functional interface or not.
- ➔ **Documentation:** Software applications (for example, FindBugs) use annotations to decide the quality of code or generate reports automatically such as Jenkins.

- **Code generation:** Annotations can be used to create code or XML files automatically using metadata information available in the code (for example, JAX library).
- **Runtime processing:** Annotations that are observed in runtime are used for different purposes such as unit testing, dependency injection, validation, logging, data access, and so on.

11.3 Declaring Annotations

The shortest manner in which you can declare or apply a Java annotation is by prefixing an @ as follows:

Syntax:

@<name>

Here,

@ (at) symbol signals to the compiler that this is an annotation. The name following the @ symbol is the name of the annotation.

Example:

@Item

Here, the annotation name is Item.

11.4 Predefined Annotation Types

Java contains three built-in annotations that are used to provide instructions to the Java compiler. These annotations are as follows:

- @Deprecated
- @Override
- @SuppressWarnings

@Deprecated

The @Deprecated annotation is used for deprecating or marking a class, method, or field as deprecated, signifying that the part of code will no longer be used. If any that code uses deprecated classes, methods, or fields is compiled, and a warning message will be generated by the compiler. Code Snippet 1 demonstrates usage of @Deprecated Java annotation.

Code Snippet 1:

```
@Deprecated
class SampleContent {
}
```

```
public class Test {
    public static void main(String args[]) {
        SampleContent c = new SampleContent();
        ...
    }
}
```

The use of @Deprecated Java annotation directly before the class declaration turns the class as deprecated. Hence, when you attempt to create an instance of the class in main, you will get a compiler warning.

In an IDE such as NetBeans, you will be able to view the compiler warnings for deprecated items only if it is set on. Figure 11.1 shows NetBeans options to set it to ON.

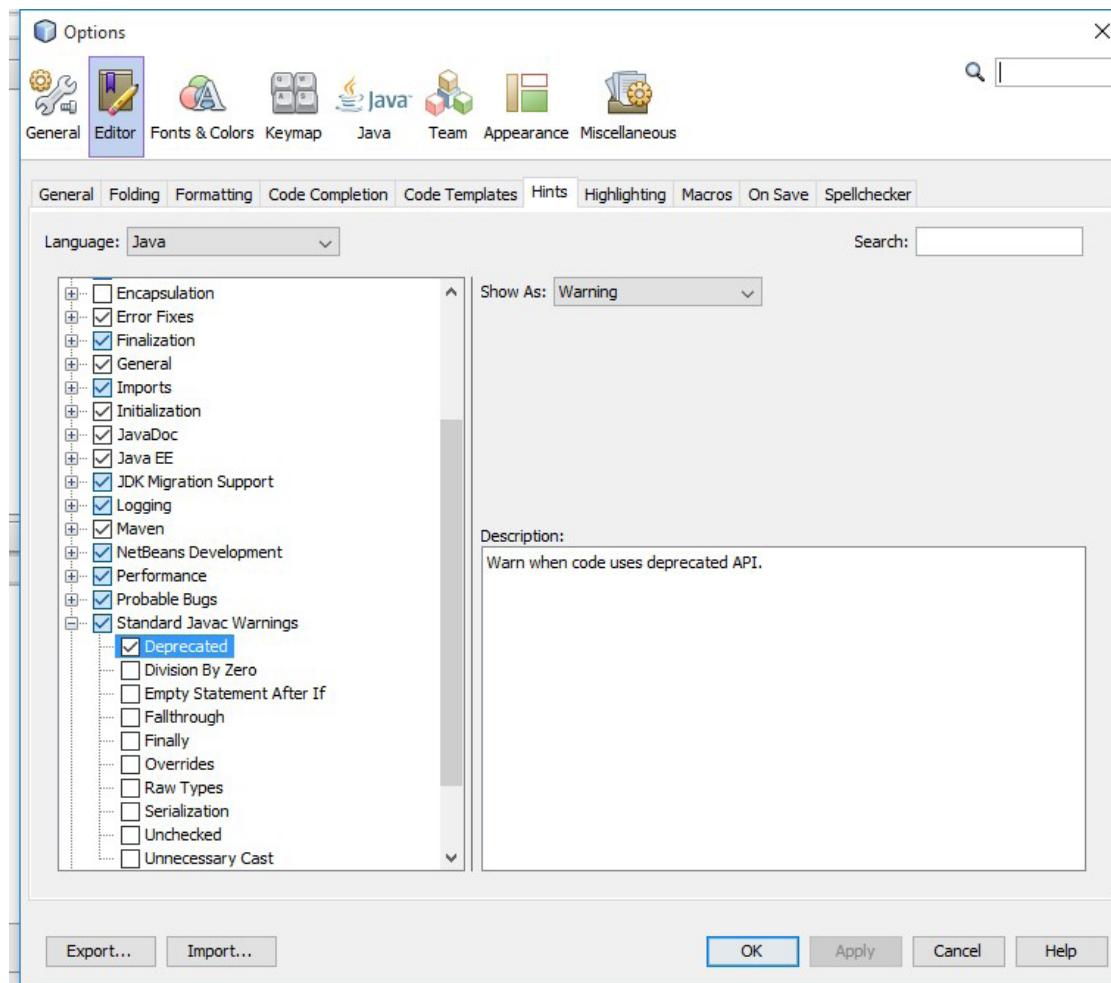


Figure 11.1: Configuring NetBeans to Show Warnings

An example to deprecate an interface is shown in Code Snippet 2.

Code Snippet 2:

```
@Deprecated
interface Hotel {
    // ...
}
```

An example to use this annotation to deprecate a method is shown in Code Snippet 3.

Code Snippet 3:

```
class SampleClass {
    @Deprecated
    public void doSpecificAction() {
        // method
    }
    public void doRandomAction() {
        // new, alternate method
    }
}
```

If the method `doSpecificAction()` is used in the code, it will generate a compiler warning.

An example to deprecate a member variable is shown in Code Snippet 4.

Code Snippet 4:

```
class SampleConstant {
    @Deprecated
    public static final int MAX_SIZE = 2048;
    // new
    public static final int MAX_UPLOAD_SIZE = 2048;
}
```

An example to deprecate a constructor is shown in Code Snippet 5.

Code Snippet 5:

```
class Gadget {
    @Deprecated
    Gadget(String color, int height, int width)
```

```
{
}

// new

Gadget(Style style)

{
}

}
```

@Override

The `@Override` Java annotation is used to create a compile time check to indicate that a method is being overridden.

The compiler returns an error, if the method does not match a program in superclass.

The `@Override` annotation is not essential in order to override a method in a superclass. If the overridden method name is altered in superclass, then its subclass method will no longer override it.

Based on the `@Override` annotation, the compiler indicates that the method in subclass is not overriding any method in the superclass.

Example 1 shows code where `ClassOne` has a method named `show()` and its child class also has a method `show()`, but the child class is not overriding the parent class method, as it has a different parameter type.

Example 1:

```
public class ClassOne
{
    public void show (String testmsg) {
        System.out.println(testmsg);
    }
    public static void main (String args[]) {
        SubClass obj = new SubClass ();
        obj.show ("Good day!!");
    }
}

class SubClass extends ClassOne
{
    @Override
    public void show (int testmsg) {
        System.out.println (" I want to say: "+ testmsg);
    }
}
```

When the code is compiled, it generates a compiler error saying, "method does not override or implement a method from supertype". This is because of `@Override` annotation. It enforces the overriding and

points out an error when there is no overriding.

Example 2 shows the same example, but in this case, the method named `show()` in child class actually overriding a method `show()` defined in the parent class.

Example 2:

```
public class ClassOne
{
    public void show(String testmsg) {
        System.out.println(testmsg);
    }
    public static void main(String args[]) {
        SubClass obj = new SubClass();
        obj.show("Good day!!!");
    }
}
class SubClass extends ClassOne
{
    @Override
    public void show(String testmsg) {
        System.out.println(" I want to say: "+ testmsg);
    }
}
```

Output:

I want to say: Good day

Since, the `@Override` annotation is not utilized, the program runs uninterrupted without any issues and the output is displayed.

Requirement for `@Override` annotation

There are two advantages of using `@Override` annotation:

- If a programmer makes any unintentional error such as wrong method name, parameter types, or so on while overriding, then it results in a compile time error. Using this annotation will instruct compiler that it is overriding this method. However, the subclass method would denote a new method (not the overriding method) in subclass, if the annotation were not utilized wherever required.
- It helps to enhance code readability. Consequently, if the signature of overridden method were altered, all the subclasses that override the specific method would result in a compilation error, which would indicate to you to change the signature in the subclasses. If there are many classes in an application, then this annotation helps to identify the classes that require changes when the signature of a method is changed.

@SuppressWarnings

The `@SuppressWarnings` annotation can suppress compiler warnings in any available method. For example, a warning will be generated, when a deprecated method is called by a method. These warnings are suppressed by annotating the method containing the code with `@SuppressWarnings` annotation.

Code Snippet 6 shows usage of `@SuppressWarnings` Java annotation.

Code Snippet 6:

```
@SuppressWarnings
public void alertMethod() {
}
```

This annotation type can be utilized with one or more warnings in the form of arguments as shown in Code Snippet 7. These warnings are predefined by the compiler.

Code Snippet 7:

```
@SuppressWarnings("unmarked")
@SuppressWarnings({"unmarked", "deprecation"})
```

Example 3 shows an example for using this annotation. If you have a class that implements `Serializable` interface and you forget to declare `SerialVersionUID`, it results in a compiler warning. To suppress this warning, you can use `@SuppressWarnings("serial")` at the class level.

Example 3:

```
import java.io.Serializable; // To use Serializable class

@SuppressWarnings("serial") // Muting the warning
public class MyAlert implements Serializable{
    private String edition;
    public String getEdition(){ return edition; };
    public static void main(String args[]){
        ...
        ...
    }
}
```

There will be no warning upon compilation because `@SuppressWarnings ("serial")` was used here.

11.5 Creating Custom Annotations

Custom annotations can be created in Java. Similar to a Java class or interface, such annotations are defined in their own file as shown in Code Snippet 8.

Code Snippet 8:

```
@interface SampleAnnotate{
    String samplValue();
    String name();
    int age();
    String[] addNames();
}
```

Code Snippet 8 defines an annotation called `SampleAnnotate`, which has four elements. The `@interface` keyword indicates Java compiler that this is a Java annotation definition.

Every element in an annotation is defined just like a method in an interface. It consists of a data type and a name.

@Retention

A custom annotation if required at runtime for inspection through reflection can be marked with the `@Retention` annotation. This annotation tells the compiler how long annotations with the annotated type should be retained.

Code Snippet 9 shows an example.

Code Snippet 9:

```
@Retention(RetentionPolicy.RUNTIME)

@interface SampleAnnotate {
    ...
}
```

Notice the `@Retention` annotation added on top of `SampleAnnotate` definition.

It has following syntax:

Syntax:

```
@Retention(RetentionPolicy.value)
```

where,

`RetentionPolicy` determines the stopping point of annotation. It is an enumeration with three constant values:

- ➔ `RetentionPolicy.RUNTIME`

Denotes to Java compiler as well as JVM that the annotation is to be retained until runtime so that it can be manageable over reflection.

- ➔ `RetentionPolicy.CLASS`

Means that the annotation is stored in the `.class` file and is not accessible at runtime. Annotations under this type will be recorded in the class file by the compiler, but will not be retained by the VM

at run time. This is the default retention policy.

- `RetentionPolicy.SOURCE`

Means that the annotation is only available in the source code and not accessible in both the class files and runtime.

@Target

Specific Java elements can be mentioned in a custom annotation to be annotated. Code Snippet 10 shows an example for `@Target` Java annotation.

Code Snippet 10:

```
@Target({ElementType.METHOD})
public @interface SampleAnnotate{...}
```

Here, the target indicates that a method will be annotated.

`ElementType` enumeration values (for targets) are as follows:

- `ElementType.ANNOTATION_TYPE`
- `ElementType.CONSTRUCTOR`
- `ElementType.FIELD`
- `ElementType.LOCAL_VARIABLE`
- `ElementType.METHOD`
- `ElementType.PACKAGE`
- `ElementType.PARAMETER`
- `ElementType.TYPE`

The `ANNOTATION_TYPE` target means Java annotation definitions. Hence, the annotation can only be used to annotate further annotations such as `@Target` and `@Retention` annotations.

The `TYPE` target denotes any type such as a class, interface, `enum`, or annotation.

@Inherited

The `@Inherited` annotation indicates that a custom Java annotation in a class is inherited by subclasses inheriting from that class. Code Snippet 11 shows an `@Inherited` annotation example.

Code Snippet 11:

```
import java.lang.annotation.Inherited;
@Inherited
public @interface SampleAnnotate {
}
@SampleAnnotate
class Person { ... }
public class Employee extends Person { ... }
```

In this example, the class `Employee` inherits the annotation `@SampleAnnotate` because `Employee` inherits from `Person` and the class `Person` has been given a `@SampleAnnotate` annotation of `@Inherited` type.

@Documented

`@Documented` annotation is utilized for informing the JavaDoc tool that custom annotations have to be visible in the JavaDoc for classes that are using custom annotation. Code Snippet 12 shows an example for `@Documented` Java annotation.

Code Snippet 12:

```
import java.lang.annotation.Documented;
@Documented
@interface TestAnnotate {
...
}
@TestAnnotate
public class Employee { ... }
```

While creating JavaDoc for the `Employee` class, the `@TestAnnotate` is included in the JavaDoc.

Annotation Elements

A custom Java annotation can have elements for which you can set values. A Java element is similar to a parameter or an attribute.

A Java annotation with an element is as shown in Code Snippet 13.

Code Snippet 13:

```
@Entity (tableName = "gadgets")
```

In this example, the annotation contains a single element named `Entity`, with `gadgets` as its value. Elements are positioned inside the parentheses following the annotation name. Annotations with zero elements do not require parentheses.

An annotation can have multiple elements. A multiple element Java annotation is shown in Code Snippet 14.

Code Snippet 14:

```
@Entity (tableName = "gadgets", primaryKey = "id")
```

If an annotation contains a single element, the standard approach is to name the element `value` as shown in Code Snippet 15.

Code Snippet 15:

```
@InsertNew (value = "yes")
```

If an annotation contains a single element named `value`, you can provide the value without the element name. An annotation of element with a value alone is shown in Code Snippet 16. The element name is implied here, but is not provided.

Code Snippet 16:

```
@InsertNew ("yes")
```

Annotation Placement

Java annotations are placed on top of classes, interfaces, methods, method parameters, fields, and local variables. An example of annotation added before a class definition is shown in Code Snippet 17.

Code Snippet 17:

```
@Entity
public class Gadget { }
```

Declaring an Annotation Type

Various annotations can substitute comments in coding. Consider an example scenario.

Code Snippet 18 displays an instance where a Web developer customarily starts the structure of every class with comments, giving vital information.

Code Snippet 18:

```
public class SampleInfo {

    // Lead Designer: Mark Lee
    // Last modified: 06/01/2021
    // By: Mark Lee
    // devTeam: Parker, Kevin, and Anna
    // Business Logic Functionality begins here
}
```

You can achieve the same outcome via annotations. Define the annotation type first, to add the same metadata with an annotation as shown in Code Snippet 19.

Code Snippet 19:

```
@interface ClassWebDevelopment {
    String leadDesigner();
    int recentUpdatds() default 1;
    String lastEdited() default "N/A";
    String lastEditedBy() default "N/A";

    String[] devTeam();
}
```

Once the comment type is defined, use annotations of that type with the values as shown in Code Snippet 20.

Code Snippet 20:

```
/*Annotation Sample
*Aptech Java*/
@ClassWebDevelopment (
leadDesigner = "Mark Lee",
lastEdited= "6/04/2012",
lastEditedBy= "Mark Lee",
// devTeam
devTeam= {"Parker", "Kevin", "Anna"}
)
public class SamplInfo {
// rest
}
```

11.6 Type Annotations

Java SE 8 makes annotations even more powerful and better. Annotations have now become multipurpose. Type annotations are formed to maintain better analysis of Java programs, ensuring better type checking.

For example, to ensure that a particular variable in program is never assigned to null or to avoid triggering a `NullPointerException`, a custom plugin can be written. Then, modify code to annotate that particular variable specifying that it is never assigned to null. Following syntax shows the variable declaration.

Syntax:

```
@NonNull String str;
```

where,

`str` is a `String` variable in the annotation.

Compiler shows a warning if it notices a potential problem, allowing you to modify the code to avoid the error while calculating the `NonNull` module at the command line during code compilation.

After editing the code to remove all warnings, this specific error will not occur when the program runs.

Multiple type-checking can be used in modules in which each module monitors a different type of error. Similarly, we can build on top of the Java type system, balancing with precise checks as and when it is required.

Usage of type annotations and the presence of plugins will help you to write code that is more robust and less prone to errors.

Here are some examples:

→ **Class instance creation expression:**

Syntax:

```
new @Interned MyObject()
```

where,

MyObject is a class

→ **Typecast:**

Syntax:

```
myString = (@NotNull String) str;
```

where,

myString is a string variable

11.7 Repeating Annotations

It may be required in some circumstances to apply the same annotation to a declaration or type use. Java version 8 onwards provides repeating annotations feature to enable you to do this.

For instance, consider that you require to use a timer service that enables to run a method `scorePapers()` at a given time or on a particular schedule, on the last day of the month, and on every Wednesday at 09:00 p.m. To set the timer to run, create a `@ScoreSchedule` annotation and use it two-fold to the `scorePapers()` method.

Code Snippet 21 demonstrates the code for this.

Code Snippet 21:

```
@ScoreSchedule(dayOfMonth="last")
@ScoreSchedule(dayOfWeek="Wed", hour="21")
public void scorePapers() { ... }
```

Here, the annotation `@ScoreSchedule` has been applied twice, thus, it is a repeating annotation. Repeating annotations can be applied not just to methods, but to any item that can be annotated.

Code Snippet 22 shows an example where a class is defined for handling unauthorized entry exceptions.

The class is annotated with one `@Alert` annotation for publishers, one `@Alert` for editors, and another for authors.

Code Snippet 22:

```
@Alert(role="Publishers")
@Alert(role="Editors")
@Alert(role="Authors")
public class UnauthorizedAccessException extends { ... }
```

For compatibility purposes, repeating annotations are loaded in a container annotation generated by the Java compiler. For the compiler to perform this, two declarations are necessary in the code as follows:

→ Declare a Repeatable Annotation Type

This annotation type mentioned with the `@Repeatable` meta-annotation. Code Snippet 23 demonstrates a custom repeatable annotation type.

Code Snippet 23:

```
/*
 *Aptech Java8
 *Java tester
 */
import java.lang.annotation.Repeatable;
@Repeatable(ScoreSchedules.class)
public @interface ScoreSchedule {
    String monthDay() default "1st";
    String weekDay() default "Monday";
    int hour() default 12;
}
```

The Java compiler generates an annotation type to store repeating annotations which is the value of the `@Repeatable` annotation.

→ Declare Containing Annotation Type

The containing annotation type contains an array type element value. The array type value must be the repeatable annotation type. The containing annotation type declaration is given in Code Snippet 24.

Code Snippet 24:

```
public @interface ScoreSchedules {
    ScoreSchedule[] value();
}
```

Here, `ScoreSchedules` is the class for `@interface` annotation.

Code Snippet 25 shows the complete code.

Code Snippet 25:

```
import java.lang.annotation.Repeatable;

@Repeatable(ScoreSchedules.class)
@interface ScoreSchedule {
    String monthDay() default "1st";
    String weekDay() default "Monday";
    int hour() default 12;
}

@interface ScoreSchedules {
    ScoreSchedule[] value();
}

public class RepeatingDemo {

    public static void main(String args[]) {} 
    @ScoreSchedule(monthDay="last")
    @ScoreSchedule(weekDay="Fri", hour=23)
    public void scorePapers() { }
}
```

11.8 Processing Annotations Using Reflection

Reflection API of Java can be used to access annotations on any type such as class or interface or methods.

There are several methods in the Reflection API to retrieve annotations. The conduct of the methods that generate a single annotation, such as `AnnotatedElement.getAnnotationByType (Class <T>)` remains unchanged and returns only a single annotation when one annotation of the requisite type is available. If one or more annotation of the required type is available, it can be acquired by getting the container annotation.

An example is shown in Code Snippet 26.

Code Snippet 26:

```
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.WildcardType;
import java.util.List;
/*
 *Aptech Java
 *Reflection Sample */
public class SampleRefl {
    public static void main(String[] args)
        throws NoSuchMethodException, SecurityException {
        Method sampMeth = SampleRefl.class.getMethod("sampMeth",
            new Class[0]);
        System.out.println("Annotation on SampMeth is " +
            sampMeth.getAnnotation(Annotation.class));
    }
}
```

```

List.class); //here Method defined as sampMeth

ParameterizedType sampleLiTy = (ParameterizedType) sampMeth.
getGenericParameterTypes()[0]; //here List Type defined as SampleLiTy

ParameterizedType sampleClTy = (ParameterizedType) sampleLiTy.
getActualTypeArguments()[0]; //here Class Type defined as SampleClTy

WildcardType sampleGenTy = (WildcardType) sampleClTy.
getActualTypeArguments()[0];
//here generic Type defined as SampleGenTy

Class<?> SampleGenCl = (Class<?>) sampleGenTy.getUpperBounds()[0];
//here generic Class defined as sampleGenCl

boolean isException = Exception.class.
isAssignableFrom(SampleGenCl);
//to display whether the statement is true or false
System.out.println("This Class extends RuntimeException: " +
isException);

boolean isRuntimeException = RuntimeException.class.
isAssignableFrom(SampleGenCl);
// to display whether the statement is true or false
System.out.println("This Class extends RuntimeException: " +
isRuntimeException);
}
public void sampMeth(List<Class<? extends Exception>>
exceptionClasses) {

}
}

```

Output:

This Class extends RuntimeException:true

This Class extends RuntimeException:false

Here, SampleGenCl class is reflected through sampMeth() method.

@Functional Interface

A functional interface is an interface that has one abstract method (not default). The compiler will operate the annotated element as a functional interface and generates an error if the element does not comply with the requirements.

Code Snippet 27 shows the usage of this annotation.

Code Snippet 27:

```
@FunctionalInterface
interface MyCustomInterface
{
    ...
}
```

11.9 Known Libraries Using Annotations

Many libraries these days are utilizing annotations for various reasons such as code quality analysis, unit testing, XML parsing, dependency injection, and so on. Some of these libraries include JAXB, Junit, and FindBugs.

11.10 Base64 (`java.util.Base64`) Encoding

Java 8 encloses an inbuilt encoder and decoder for Base64 encoding. There are three types of Base64 encoding:

- **Simple** – Output is limited to a set of characters between A-Z, a-z, 0-9, and +.

The encoder do not add any line feed in output and the decoder declines any input other than A-Z ,a-z, 0-9, and +/.

A sample of this basic encoding is as shown in Code Snippet 28.

Code Snippet 28:

```
import java.util.Base64;
...
// Encodesample
String asB64 = Base64.getEncoder().encodeToString("AptechJava8".
getBytes("utf-8"));
System.out.println(asB64); // desired output is QXB0ZWNoSmF2YTg=

// Decode sample
byte[] asBytes = Base64.getDecoder().decode("QXB0ZWNoSmF2YTg=");
System.out.println(new String(asBytes, "utf-8")); // desired output
// is: AptechJava8
...
```

- **URL** – The final output is safe from filename and URL and is limited to a set of characters between A-Z, a-z, 0-9, and +_.

Sample of this URL Encoding is shown in Code Snippet 29.

Code Snippet 29:

```

String sampleBasicEncod = Base64.getEncoder() .
    encodeToString("subjects?abcd".getBytes("utf-8"));
System.out.println("Using Basic Alphabet: " + sampleBasicEncod);
String sampleUrlEncod = Base64.getUrlEncoder() .
    encodeToString("subjects?abcd".getBytes("utf-8"));
System.out.println("Using URL Alphabet: " + sampleUrlEncod);

```

- **MIME** – Output is limited to MIME friendly format. Output is denoted in lines with maximum 76 characters each and uses a carriage return '\r' followed by a linefeed '\n' as the line separator. The output will be line separator free.

Nested Classes

There are two nested classes in Base64:

- static class Base64.Encoder
- static class Base64.Decoder

Methods

Add the respective static `Base64.encoder` or static `Base64.Decoder` method that is appropriate for the code.

- **static Base64.Decoder getDecoder()**
Displays a Base64.Decoder that processes using the Basic type base64 decoding scheme.
- **static Base64.Encoder getEncoder()**
Displays a Base64.Encoder that processes using the Basic type base64 encoding scheme.
- **getMimeDecoder()**
Displays a Base64.Decoder that process using the MIME type base64 decoding scheme.
- **getMimeEncoder()**
Displays a Base64.Encoder that process using the MIME type base64 encoding scheme.
- **getMimeEncoder(int lineLength, byte[] lineSeparator)**
Displays a Base64.Encoder that processes using the MIME type base64 encoding scheme with required line length and line separators.

→ **getUrlDecoder()**

Displays a Base64.Decoder that processes using the URL and Filename safe type base64 decoding scheme.

→ **getUrlEncoder()**

Displays a Base64.Encoder that processes using the URL and Filename safe type base64 encoding scheme.

Base64 Encoding Example

Following example illustrates how to encode a string to base 64, then, decode the same String back to a base64 encoded output stream as shown in Code Snippet 30.

Code Snippet 30:

```
import java.util.Base64;
import java.util.UUID;
import java.io.UnsupportedEncodingException;

/**Aptech Java
*Base64 example in detail
*/

public class HelloWorld {
    public static void main(String args[]) {
        try {
            // Encoding a string using Base64
            String sampleBase64EncoStr = Base64.getEncoder() .
            encodeToString("AptechJava8".getBytes("utf-8"));
            System.out.println("Encoded String (Basic) looks like this: " +
sampleBase64EncoStr);

            // Decoding a string using Base64
            byte[] base64decodedBytes = Base64.getDecoder() .
            decode(sampleBase64EncoStr);

            System.out.println("Decoded String is : " + new
String(base64decodedBytes, "utf-8"));
            sampleBase64EncoStr = Base64.getUrlEncoder() .
            encodeToString("AptechJava8".getBytes("utf-8"));
            System.out.println("Encoded String (URL) looks like this: " +
sampleBase64EncoStr);
            StringBuilder strBuild = new StringBuilder();

            for (int j = 0; j < 10; ++j) {
                strBuild.append(UUID.randomUUID().toString());
            }
        }
    }
}
```

```

byte[] sampleMimeBytes = strBuild.toString().getBytes("utf-8");
String sampleMimeEncStr = Base64.getMimeEncoder().
    encodeToString(sampleMimeBytes);
System.out.println("Encoded String (MIME) looks like this: " +
sampleMimeEncStr);

} catch (UnsupportedEncodingException u) { // to display error
    System.out.println("Unsupported Encoding Error: " +
    u.getMessage());
}
}
}
}

```

Output:

Encoded String (Basic) looks like this: QXB0ZWNoSmF2YTg=

Decoded String is: AptechJava8

Encoded String (URL) looks like this: QXB0ZWNoSmF2YTg=

Encoded String (MIME) look like this:

YzNINmYyNjctY2JkNi00OTFiLThiMzQtZTMwZTJkZjBkZTU2YTljM2FIN2YtZTJlMi00YmVlWI2
NDQtNjFjMGZiYml4NDc5NWFmOTA3YjktYTc5Ni00MDBkLTg2Zjl0TII0GRhMDJkODE4YmZmZmYx
YzQtNjA1Yy00M2ZhLWJkNTMtNzEwOWI5NzYwMGY4NGE0YTzkZjUtZWQ5Zi00NWY0LWE3YTAtYmZi
MDU5MWRjMGM2OWVjOTYyNDEtZGU1Yi00YWFiLTlIMmYtM2MyYjg3MjY4ZTYzNDg0MGMzYzctZjEy
MS00OTJiLThjM2MtZTQzMDExNmQzM2RIMDdIYjMyNTYtNmI2Mi00NjVmLTgyYzItOWY5OGQ3OTdh
ZmJlMzJiYzNhNWEtZmM1Mi00ZmI1LTg5NDAtZjdjZGjjOWFhYWVjNTg4Y2YxODktYjk2NC00YTJk
LTlhMWYtYmNmNGRmOTQ5MTI4

11.11 Check Your Progress

1. Which of the following is not in use by annotations?

(A)	Compiler information	(C)	Code generation
(B)	Documentation	(D)	Variable declaration

2. Which of the following is not a predefined annotation type in Java?

(A)	@Compiler	(C)	@Deprecated
(B)	@SuppressWarnings	(D)	@Override

3. Which type of annotation can be used to annotate another annotation definition?

(A)	@Deprecated	(C)	@Target
(B)	@Retention	(D)	@Autowired

4. Which of these two come under ElementType class?

(A)	ANNOTATION_TYPE	(C)	SURFACE
(B)	SURFACEVIEW	(D)	PARAMETER

5. Which of the following is not a kind of Base64 encoding?

(A)	URL	(C)	Compiler
(B)	Simple	(D)	MIME

11.11.1 Answers

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

Summary

- Annotations are comments, notes, remarks, or explanations. In Java, annotations help to associate these additional information (also called metadata) to the program elements. Annotations affect the way a program is treated by tools and libraries which in turn can affect program semantics.
- Annotations can be determined from source files or class files at runtime.
- The @Deprecated annotation is used for deprecating or marking a class, method, or field as deprecated, signifying that the part of code no longer will no longer be used.
- The @SuppressWarnings annotation can suppress the compiler warnings in any available method. For example, a warning will be generated, when a deprecated method is called by a method.
- Since, annotation types are piled up and stored in byte code files such as classes, the annotations reverted by these methods can be enquired as any systematic Java object.
- Base64 encoding has an in-built encoder and a decoder. In Java 8, there are three types of Base64 encoding namely, Simple, URL, and MIME.

Try it Yourself

1. Define an annotation type for a prototype of an automobile with elements id, mileage, model, and year. Specify the default value as unassigned for model and 2021 for year.
(Hint: Name the interface as Prototype)
2. Create classes Person, Employee, and Manager wherein Employee and Manager inherit from Person. Create appropriate member variables and methods in each of the classes. Use annotations to ensure that the subclass methods are overriding the parent class method.

**LEARN
@
YOUR PACE
YOUR
DEVICE**

Onlinevarsity

Session - 12

Functional Programming in Java

Welcome to the Session, **Functional Programming in Java**.

This session defines Functional Programming in Java and explains how lambda expressions can facilitate it. The session also explains method references and functional interfaces. Finally, the session explains default methods and static methods in interfaces.

In this Session, you will learn to:

- Explain lambda expressions
- Describe method references
- Explain functional interfaces
- Explain default methods



12.1 Introduction

One of the new features introduced in Java 8 is functional programming. Functional programming is a type of programming approach. It emphasizes utilization of functions and writing code that does not change state. Using functional programming, you can pass functions as parameters to other functions and return them as values. Functional programming provides many benefits to programmers - it makes programs easier to test, it is thread-safe, and is more modular.

12.2 Lambda Expressions

An important addition to Java 8 is the introduction of lambda expressions. They facilitate functional programming and make application development easier. A lambda expression is similar to the concept of an anonymous function. Origin of the word lambda is Greek; It is the 11th letter of Greek alphabet.

Lambda expressions (also called lambdas in short) have various features that set them apart. A lambda expression is a function that expects and accepts input parameters and may return output. It is a component of a functional interface. A functional interface is an interface with one method and is used as the type of a lambda expression.

A lambda expression creates possibility of passing functions in a code, which is similar to passing parameters and data in general. A lambda expression is like a shorthand expression that allows executing a method exactly where it is required. Lambda expressions eliminate the requirement of a new method declaration as well as a distinct method to a containing class.

12.2.1 Syntax of Lambda Expressions

The syntax of a lambda expression is as follows:

Syntax:

parameters -> body

where,

parameters are variables

-> is the lambda operator

body is parameter values

12.2.2 Rules for Lambda Expressions

Few rules of lambda expressions are as follows:

- **Type declarations are optional:** The parameter type declaration is optional.
- **Parentheses around parameters can be omitted:** Parentheses usage is optional with single parameter cases.
- **Curly braces may or may not be present:** Similarly, curly braces are optional in single parameter cases.

- **The return keyword may or may not be present:** The `return` keyword usage is optional in single parameter cases.
- **Statement body may contain varied number of statements:** The body of the lambda expressions can contain zero, one, or more statements.

12.3 Single Method Interface and Lambdas

Functional programming can be used to create event listeners. In Java, event listeners are frequently defined as Java interfaces with a single method. An example of a single method interface is shown in Code Snippet 1. Assume that a class `State` is previously declared.

Code Snippet 1:

```
interface StateChangeListener {
    public void onStateChange(State previousState, State presentState);
}
```

In Code Snippet 1, a single method call will occur on each state change event.

Traditionally, adding an event listener would be done as shown in Code Snippet 2.

Code Snippet 2:

```
public class StateTest {
    public static void main(String args[]) {
        StateTest objStateTest = new StateTest();
        objStateTest.addStateListener(new StateChangeListener() {
            public void onStateChange(State previousState, State presentState) {
                // action statements.
                System.out.println("State change event occurred");
            }
        });
    }
}
```

Code Snippet 2 shows adding an event listener using an anonymous method implementation.

Code Snippet 3 shows adding an event listener using a lambda expression.

Code Snippet 3:

```
public class StateTest {
    public static void main(String args[]) {
        StateTest objStateTest = new StateTest();
```

```

objStateTest.addStateListener((previousState, presentState) ->
    System.out.println("State change event occurred"));
}
}

```

This is a much simpler and compact way to add an event listener. The lambda expression within the `addStateListener` method is: `(previousState, presentState) -> System.out.println("State change event occurred")`

Here, `previousState` and `presentState` are parameters of the event. The lambda expression and the parameter type of `addStateListener()` method have to match each other. If the type and expression is matched, lambda expression turns into a function, which creates the same interface as a parameter. This interface contains a single method. Thus, the lambda expression is matched successfully.

12.4 Lambda Parameters

As Java lambda expressions are like anonymous methods, they can also handle parameters similar to methods. Code Snippet 2 shown earlier, explained `(previousState, presentState)` as a part of lambda expression. These parameters must match with parameters of the single method interface.

12.4.1 Zero Parameters

Parentheses with no comments indicates that the lambda takes no parameters.

The lambda expression is written as shown in Code Snippet 4 through a simple example. Here, the method does not take any parameters.

Code Snippet 4:

```
() ->System.out.println("Zero parameter lambda");
```

At this point, the parentheses contain no content, which states that the lambda does not have any parameters.

12.4.2 One Parameter

If the method takes one parameter, then the lambda expression will be given as shown in Code Snippet 5.

Code Snippet 5:

```
(param) ->System.out.println("One parameter: "+param); //with parentheses
```

Here, the parentheses contains a value, which means the lambda receives one parameter.

As mentioned earlier, lambda expression with a single parameter requires no parentheses. A statement to illustrate this is shown in Code Snippet 6.

Code Snippet 6:

```
param->System.out.println("One parameter: " + param); //without parentheses
```

12.4.3 Multiple Parameters

If the method returns with multiple parameters match, then parameters must be added within the parentheses as shown in Code Snippet 7.

Code Snippet 7:

```
(pA, pB) ->System.out.println("Multiple parameters defined: " + pA + ", " + pB);
```

Here, the parentheses contain two values, indicating that the lambda expression will take two parameters.

12.4.4 Parameter Types

Defining parameter types for a lambda expression is necessary in some cases, where the compiler is inconclusive about the parameter type match between functional interface method and lambda.

Code Snippet 8 shows a Java lambda parameter with a specified custom type.

Code Snippet 8:

```
(Phone smartphone) ->System.out.println("The smartphone is: " + smartphone.  
getName());
```

In Code Snippet 8, the type (`Phone`) of the parameter (`smartphone`) is specified along with the parameter name. It is similar to declaring a parameter in a method or making an anonymous implementation of an interface.

Lambda function body has to be enclosed within {} bracket similar to regular Java code in case of multiple line lambda expressions.

Code Snippet 9 shows multiple line lambda.

Code Snippet 9:

```
(previousState, presentState) -> { //bracket usage for multiple lines  
System.out.println("The result as Previous state: " + previousState);  
System.out.println("The result as Present state: " + presentState);  
}
```

12.5 Returning a Value

The procedure for returning values from Java lambda expressions is similar to that in a regular Java method. Code Snippet 10 illustrates this.

Code Snippet 10:

```
(pA) -> {
    System.out.println("The output will be: " + pA);
    return "result value"; // return statement
}
```

A return statement for specific calculations can be added in a shortened form. Consider an example. Code Snippet 11 shows a regular form while Code Snippet 12 shows its condensed form.

Code Snippet 11:

```
(iA, jB) -> {return iA>jB; }
```

Code Snippet 12:

```
(iA, jB) ->iA>jB;
```

The compiler identifies that the statement `iA>jB` is the return value of lambda expression.

12.6 Lambdas as Objects

Java lambda expression is a sort of object too. It can be assigned as a regular object to a variable and can be passed. It is similar to other object usage in Java. Code Snippets 13 and 14 demonstrate an example for this.

Code Snippet 13:

```
public interface SampleComparator {
    public boolean compare(int iA, int iB);
}
```

Code Snippet 14 displays the implementation of lambda.

Code Snippet 14:

```
SampleComparator sampleComparator = (iA, iB) -> return iA>iB;
boolean result = sampleComparator.compare(3, 6);
```

Code Snippet 14 illustrates how the lambda object is assigned to a variable and passed as an object.

Code Snippet 15 shows a lambda used to sort strings by its length.

Code Snippet 15:

```
Arrays.sort(sampleStrArr,
    (String strA, String strB) ->strB.length() - strA.length());
```

Here, a lambda is applied in string operation and performs sort operation in the given array, `sampleStrArr`.

12.7 Advantages and Uses of Lambda Expressions

Lambda expressions provide several unique advantages.

Some of the advantages of lambda expressions are as follows:

- More readable code
- Rapid fast coding specifically in Collections
- Much easier parallel processing

Uses

Code Snippet 16 shows a complete program utilizing lambda.

Code Snippet 16:

```
public class SampleLambda {
    public static void main(String args[]) {
        SampleLambda perform = new SampleLambda();
        //to receive results with type declaration
        MathOperation add = (int ab, int xy) -> ab + xy;
        // to receive results without type declaration
        MathOperation subtr = (ab, xy) -> ab - xy;
        // to receive results with return statement along with curly braces
        MathOperation multi = (int ab, int xy) -> { return ab * xy; };
        // to receive results without return statement and curly braces
        MathOperation div = (int ab, int xy) -> ab / xy;
        System.out.println("Addition operation with Type declaration : 20 + 5 = "
                + perform.operate(20, 10, add));
        System.out.println("Subtraction operation without Type declaration:
                20 - 5 = " + perform.operate(20, 10, subtr));
        System.out.println("Multiplication with return statement : 20 x 5 = " +
                perform.operate(20, 10, multi));
        System.out.println("Division operation without return statement : 20 /
                5 = " + perform.operate(20, 10, div));
    }
    interface MathOperation {
        int operation(int ab, int xy);
    }
}
```

```

    }
    private int operate(int ab, int xy, MathOperation mathOperation) {
        return mathOperation.operation(ab, xy);
    }
}

```

Output:

Addition operation with Type declaration: $20 + 5 = 30$

Subtraction operation without Type declaration: $20 - 5 = 10$

Multiplication with return statement: $20 \times 5 = 200$

Division operation without return statement : $20 / 5 = 2$

Here, Code Snippet 16 performs basic math operations using lambda expressions.

Code Snippet 17 shows a complete program that uses a lambda expression to display messages of greetings.

Code Snippet 17:

```

public class SampleLambda2 {

    public static void main(String args[]) {
        SampleLambda2 testLambda = new SampleLambda2 () ;

        //passing without parentheses
        GreetingService greetService1 = message -> System.out.println("Hi " +
message);

        //passing with parentheses
        GreetingService greetService2 = (message) ->
System.out.println("Hi " + message);

        greetService1.sayMessage("James");
        greetService2.sayMessage("Mary");
    }
}

interface GreetingService {
    void sayMessage(String message);
}

```

Output:

Hi James

Hi Mary

Here, Example 2 performs string related operations using lambda and displays the greeting messages.

12.8 Scope for Lambda Expressions

Code Snippet 18 uses lambda expressions with `Runnable` interface.

Code Snippet 18:

```
import static java.lang.System.out;
/**
 *Scope of Lambda example*
public class MyWishes {
    Runnable dA= () ->out.println(this);
    Runnable dB= () ->out.println(toString());
    public String toString() { return "Happy New Year!"; }
    public static void main(String args[]) {
        new MyWishes().dA.run(); //Happy New Year
        new MyWishes().dB.run(); //Happy New Year
    }
}
```

In Code Snippet 18, both the `dA` and `dB` lambdas call the `toString()` method of the `MyWishes` class. This shows the scope available to the lambda.

Final variables or 'effectively final' variables can also be referred within a lambda expression. A final variable which is assigned only once in the entire code is called an 'effectively final' variable.

12.9 Method References

Method references let us refer constructors or methods without performing an execution. They are a new feature in Java 8. The usage of method references is similar to lambdas and both require a target type that consists of a compatible functional interface.

Method references represent easy-to-read lambda expressions for methods already having a name.

Following are six types of method references:

- **TypeName::static Method:** This type refers to a static method of a class, an enum, or an interface.
- **TypeName.super::instance Method:** This type refers to an instance method from the supertype of an object.
- **ObjectRef::instance Method:** This type refers to an instance method.

- **ClassName::instance Method:** This type refers to an instance method of a class.
- **ClassName::new:** This type refers to the constructor from a class.
- **ArrayTypeName::new:** This type refers to the constructor of the specified array type.

For example, consider a case that required frequent filter of a list of files based on file types. Code Snippet 19 shows a set of methods to determine a file type.

Code Snippet 19:

```
public class FileFilters { // to filter files
    public static boolean fileIsJpeg(File file) /*sample code*/
    public static boolean fileIsTiff(File file) /*sample code*/
    public static boolean fileIsPng(File file) /*sample code*/
}
```

Method reference can be useful in file filtering cases, as shown in Code Snippet 20. Here, a method is predefined as `getFiles()` that returns a Stream.

Code Snippet 20:

```
Stream<File> Jpegs = getFiles().filter(FileFilters::fileIsJpeg);
Stream<File> Tiffs = getFiles().filter(FileFilters::fileIsTiff);
Stream<File> Pngs = getFiles().filter(FileFilters::fileIsPng);
```

The role of method reference is to point the methods by method names. A double colon (:) symbol is used to describe the method reference. Possible types of methods are as follows:

- Static methods
- Instance methods
- Methods on particular instances
- Constructors

12.9.1 Static Method References

A static method reference facilitates use of a static method as a lambda expression.

Static methods can be defined in an enum, a class, or an interface. Code Snippet 21 demonstrates static method reference.

Code Snippet 21:

```
import java.util.function.Function;
public class MainTest {
    public static void main(String[] args) {
```

```
// To retrieve result with a lambda expression
Function<Integer, String> funcA = x -> Integer.toBinaryString(x);
System.out.println(funcA.apply(11));
// To retrieve result with a method reference
Function<Integer, String> funcB = Integer::toBinaryString;
System.out.println(funcB.apply(11));
}
```

Output:

1011

1011

The first lambda expression `funcA` is created by defining an input value `x` and providing a lambda expression body. This is the normal way of creating a lambda expression.

The second lambda expression `funcB` is created by referencing a static method from `Integer` class.

12.9.2 Instance Method References

Code Snippet 22 demonstrates an instance method reference.

Code Snippet 22:

```
import java.util.function.Supplier;
public class MainTest2 {
    public static void main(String[] args) {
        Supplier<Integer> sampleSupA = () ->"Aptech".length();
        System.out.println(sampleSupA.get()); // display result
        Supplier<Integer> sampleSupB = "Aptech"::length;
        System.out.println(sampleSupB.get()); // display result
    }
}
```

Output:

6

6

12.10 Functional Interface

This is another Java 8 addition. As mentioned earlier, a functional interface is an interface with one method and is used as the type of a lambda expression. Following types of methods are not applicable for functional interface declaration:

- Static methods
- Default methods
- Methods inherited from the Object class

A functional interface type with an abstract method takes a String as a parameter and returns an integer value.

New functional interfaces included in the Java 8 package, `java.util.function`, are as follows:

- **Predicate<T>** - returns a Boolean value based on input of type T.
- **Supplier<T>** - returns an object of type T.
- **Consumer<T>** - performs an action with given object of type T.
- **Function<T, R>** - gets an object of type T and returns R.
- **BiFunction** – similar to Function, but with two parameters.
- **BiConsumer** - similar to Consumer, but with two parameters.

Functional interface also contains several corresponding interfaces for primitive types, such as:

- IntSupplier
- IntFunction<R>
- IntPredicate
- IntConsumer

Functional interface can process almost any type of parameter and similarly returns to another type. For example, String to integer or String to String.

Code Snippet 23 demonstrates use of a functional interface.

Code Snippet 23:

```
//Functional Interface sample use case
Function<String, Integer>sampleLengthA = (name) -> name.length(); //as int
Function<String, String>atr = (name) -> {return "@" + name; }; //as string
Function<String, Integer>sampleLengthB = String::length; //as int
```

In Code Snippet 23, the second line defines a function that represents '@' symbol to a String. The first and last code blocks define the same goals and process the same operation of calculating a length of a string,

but using different approaches.

For instance, you can convert a reference method (`String length`) into a functional interface function, where the applied method gets a `String` value and returns an integer value.

Code Snippet 24 shows this process.

Code Snippet 24:

```
for (String str : args) out.println(samplLengthB.apply(str));
```

This code block displays the length of the given strings.

Code Snippet 25 illustrates use of a functional interface.

Code Snippet 25:

```
import java.util.Arrays; //to use array
import java.util.List; //to use list
import java.util.function.Predicate; //to use function
public class SampleFuncInter {
    public static void main(String args[]) {
        List<Integer> list = Arrays.asList(5, 10, 15, 20, 25, 30, 35, 40,
        45);
        //In this example, a functional interface is utilized
        //to receive numbers lesser than 45
        System.out.println("Display number lesser than 45:");
        eval(list, n->n < 45);
    }
    public static void eval(List<Integer> list, Predicate<Integer> predicate)
    {
        for(Integer n: list) {
            if(predicate.test(n)) {
                System.out.println(n + " ");
            }
        }
    }
}
```

Output:

Display numbers lesser than 45:

5
10
15
20
25

30

35

40

12.11 Default Methods

Default method is another new Java 8 feature that allows default implementation for methods in an interface. In simple words, interfaces can provide constructive implementation for methods. As a result, existing classes implementing an interface will automatically inherit the default implementations. For example, `List` or `Collection` interfaces contain `forEach()` method declaration by default. Thus, implementing such methods will break the collection framework implementations. Using the new 'default method' feature, the `List/Collection` interface can have a default implementation of `forEach()` method and the classes implementing these interfaces are not required to implement the methods.

Default methods are also known as Virtual extension method or Defender methods.

Following are features of default methods:

- Default methods reduce the difference among interfaces and abstract classes.
- These methods eliminate require of utility classes.
- One of the foremost reasons to introduce default methods within interfaces is to enhance the collection API and makes them lambda expression friendly.
- Default methods can assist in the base implementation class removal. You can provide default implementation and allow the implementation classes to automatically decide which one to override.
- Default methods become useless if any class in the hierarchy contains a method having same signature.
- A default method cannot override a method from `java.lang.Object`. The reason is that `Object` is the base class for all the other Java classes.
- These methods extend interfaces without breaking implementation classes.

Code Snippet 26 demonstrates default methods.

Code Snippet 26:

```
class Test{
    public static void main(String args[]) {
        Book book = new Novel();
        book.print();
    }
}
```

```

interface Book {
    default void print() {
        System.out.println("This is a book");
    }
    static void turnPages() {
        System.out.println("Turning pages.");
    }
}

interface Journal {
    default void print() {
        System.out.println("This is a journal");
    }
}

class Novel implements Book, Journal {
    public void print() {
        Book.super.print();
        Journal.super.print();
        Book.turnPages();
        System.out.println("This is a novel");
    }
}

```

Overriding default methods is an effective option that can be applied whenever required.

The `Iterable` interface with a default method is shown in Code Snippet 27.

Code Snippet 27:

```

public interface SampleIterable<T> {
    public default void forEach(Consumer<? super T> consumer) {
        for (T a : this) {
            consumer.accept(a);
        }
    }
}

```

12.11.1 Default Method and Regular Method

Default method contains `default` modifier - that is the main difference between a regular method and default method. In addition to that, methods in classes can access and modify method arguments and also fields of their class. However, a default method can only access its arguments, interfaces do not have any state.

Finally, default methods enable adding new functionality to existing interfaces without impacting the current implementation of these interfaces. An interface have one or more default methods and still be functional.

Code Snippet 28 demonstrates another example of default methods.

Code Snippet 28:

```
public class Java8Tester {
    public static void main(String args[]) {
        Gadget gadget = new SmartGadget();
        gadget.print();
    }
}

interface Gadget {
    default void print() {
        System.out.println("This is a Gadget!");
    }
    static void call() {
        System.out.println("With Calling feature!");
    }
}

interface TextMessage {
    default void print() {
        System.out.println("With Text Messaging feature!");
    }
}

class SmartGadget implements Gadget, TextMessage {
    public void print() {
        Gadget.super.print();
        TextMessage.super.print();
        Gadget.call();
        System.out.println("It is a Smartphone!");
    }
}
```

Output:

This is a Gadget!

With calling feature!

With Text Messaging feature!

It is a Smartphone!

12.11.2 Multiple Defaults

A Java class can implement one or more interfaces and each interface can state default method through same method signature. Eventually, the inherited methods conflict with one another and cause errors. Thus, Java throws a compilation error, if it is not sure whether the class implements two or more interfaces defining the same default method. In such a case, you should override the method and select from one of the methods.

Code Snippet 29 demonstrates multiple default methods.

Code Snippet 29:

```
public interface Green {
    default void defaultMethod() {
        System.out.println("Green default method");
    }
}

public interface Red {
    default void defaultMethod() {
        System.out.println("Red default method");
    }
}

public class Impl implements Green, Red{}
```

Compiling Example 7 will result in an error:

```
java: class Impl inherits unrelated defaults for defaultMethod() from types Green and Red
```

In order to fix this, we require to provide explicit default method implementation as shown in Code Snippet 30.

Code Snippet 30:

```
public class Impl implements Green, Red {
    public void defaultMethod() {
        ...
    }
}
```

Further, to invoke default implementation provided by any of super interfaces, the coding can be as shown in Code Snippet 31.

Code Snippet 31:

```
public class Impl implements Green, Red {
    public void defaultMethod() {
        // remaining code...
        Green.super.defaultMethod();
    }
}
```

12.12 Static Methods on Interfaces

In addition to default methods, static methods can be defined in interfaces. This makes it easy to organize and access helper methods in libraries; these methods eliminate the requirement of a separate class, accordingly static methods specific to an interface can be assigned in the same interface.

For example, the new `Stream` interface contains many static methods. Thus, accessing 'helper' methods is more easier, since they are defined directly on the interface, instead of a different class such as `StreamUtil` or `Streams`.

Similar to static methods in classes, a method definition in an interface is marked as a static method by using the `static` keyword at the beginning of the method signature. By default, all method declarations in an interface, including static methods, are implicitly `public`, hence, the `public` modifier is not required to be given. Code Snippet 32 shows a simple example of a static method in an interface.

Code Snippet 32:

```
public interface ProductInfo {
    ...
    static ProductId getProductId (String ProductString) {
        ...
    }
    ...
}
```

12.13 Local-Variable Syntax for Lambda Parameters

In Java 11, the Local-Variable Syntax for Lambda Parameters was enhanced. According to this enhancement, `var` can now be used similar to local variables when declaring formal parameters of implicitly typed lambda expressions.

In earlier versions of Java, such as Java 10, a lambda expression could be written as follows:

```
(n1, n2) -> n1.compute(n2)
```

Here, the expression is implicitly typed which means that the types of all its formal parameters are inferred automatically.

In Java 10, you can also use implicit typing available for local variables. Consider some examples:

1. var n1 = new DisplayText();
2. try (var n1 = // some logic...) { ... } catch {...}

In both these cases, `var` is used for implicit typing of local variables.

Hence, in Java 11, to ensure consistency with local variables, it was decided to allow '`var`' for formal parameters of an implicitly typed lambda expressions.

```
(var n1, var n2) -> n1.compute(n2) // implicit typed lambda expression
```

The benefit of this is that it makes it easier for developers to annotate a lambda's implicit parameters or add access modifiers to implicitly-typed lambda local variables. Another benefit is that developers will experience less errors on account of using `var` in the wrong places.

The usage of annotations within lambda statements can help developers make use of reflection and gather some information about the annotated object at runtime as well as to depict behaviors such as generated source code or other hints for tools at compile time.

Note: You cannot use `var` for some parameters and skip for others. It must be uniformly used.

Consider an example.

Prior to Java 11, you may have written:

```
IDisplay dis = (@ADisplay SomeVeryLongClassName n1, final
SomeVeryLongClassName n2, final SomeVeryLongClassName n3,) -> .... ;
```

From Java 11 onwards, you can write:

```
IDisplay dis = (@ADisplay var n1, final var n2, final var n3) -> .... ;
```

Code Snippet 33 shows a full example of using local-variable syntax for lambda parameters.

Code Snippet 33:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Book{
    int id;
    String title;
    float price;
```

```
public Book(int id, String title, float price) {  
    super();  
    this.id = id;  
    this.title = title;  
    this.price = price;  
}  
}  
  
public class VarDemo{  
    public static void main(String[] args) {  
        List<Book> list=new ArrayList<Book>();  
        //Adding Products  
        list.add(new Book(1,"Harry Potter and the Chamber of Secrets",250f));  
        list.add(new Book(3,"Keyboard Ninjas",300f));  
        list.add(new Book(2,"The Three Investigators Club",150f));  
        System.out.println("Sorting on the basis of title...");  
        // implementing lambda expression  
        Collections.sort(list,(var p1, var p2)->{  
            return p1.title.compareTo(p2.title);  
        });  
        for(Book p:list){  
            System.out.println(p.id+" "+p.title+" "+p.price);  
        }  
    }  
}
```

Output:

Sorting on the basis of title...

1 Harry Potter and the Chamber of Secrets 250.0

3 Keyboard Ninjas 300.0

2 The Three Investigators Club 150.0

12.14 Check Your Progress

3. What type of support do lambda expressions provide to Java 8?

(A)	Language level support	(C)	API level support
(B)	APP level support	(D)	Method level support

4. Which among the following is often created using functional programming?

(A)	Method Listeners	(C)	Code Listeners
(B)	Event Listeners	(D)	Method References

5. Which among the following options is also known as defender methods?

(A)	Default Variables	(C)	Default Methods
(B)	Custom Variables	(D)	Custom Implementation

6. What is the main difference between default methods and regular methods?

(A)	Use of Default variables	(C)	Use of Default class
(B)	Use of Custom class	(D)	Use of Default modifier

7. Given the code:

...

```
Supplier<String>test= "Aptech"::toUpperCase;
```

```
System.out.println(test.get());
```

Which of the following denotes the correct statement?

(A)	The code uses a static method reference	(C)	The code uses an instant method reference
(B)	The code uses a default method	(D)	The code uses a regular method reference

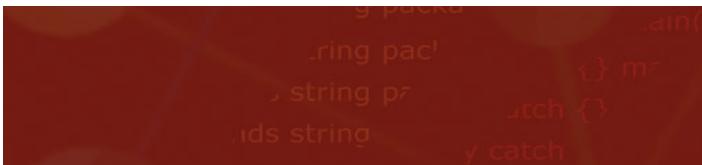
12.14.1 Answers

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

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

Summary

- Functional programming is a type of programming approach that emphasizes utilization of functions and writing code that does not change state.
- Using functional programming, you can pass functions as parameters to other functions and return them as values.
- A lambda expression is a compact expression that does not require a separate class/function definition. It facilitates functional programming.
- Depending on the parameters being passed to the lambda expression, you will use/omit parentheses.
- Default method is a new feature in Java 8 that allows default implementation for methods in an interface.
- In addition to default methods, static methods can be defined in interfaces that makes it easy to organize and access helper methods in libraries.
- From Java 11 onwards, local-variable syntax, that is, usage of var, is supported for lambda parameters.



Try it Yourself

1. The Welfare Association of Paradise Housing Society has requested children living in the society to start a lending library with unused books at home. The innovative idea had an overwhelming response from children leading to piling of hundreds of books at society's office. Hence, the secretary of the society wanted to automate the task of managing the library.

As a software developer, you take up the initiative of developing an application using Java for the same and do the following tasks:

- i. Create an interface called Books having two abstract methods: getAuthor and getTitle. Show the title of the books using the method, display().
- ii. Include an arraylist for book titles. Titles can be set as Think and Grow Rich, Global Economy, The Intelligent Investor, You Can Win, The Art of War, Habit Makes Perfect, Gulliver's Travel, The Black Sheep, War and Peace, The Odyssey. Add Predicate interface to filter title of book that starts with the letter T.
- iii. Create another interface called Publishers having methods: getName() and getAddress(). Use the method display() to show the name of the publisher.
- iv. Add a class called BookInfo that implements interfaces, Books and Publishers and overrides the display() method of the interface.

Session - 13

Stream API

Welcome to the Session, **Stream API**.

This session explains about the new Stream API in Java 8 and later versions. This API provides classes and interfaces that enable you to perform functional-style tasks on streams of data. The session describes the difference between collections and streams and then, explains functional interfaces in the Stream API. The session also concentrates on various operations supported by the Stream API through classes and interfaces such as `Optional` and `Spliterator` respectively.

In this Session, you will learn to:

- Describe the Stream API
- Outline the differences between collections and streams
- Explain the classes and interfaces in Stream API
- Describe how to use functional interfaces with Stream API
- Describe the `Optional` class and `Spliterator` interface
- Explain stream operations
- Discuss the limitations of Stream API



13.1 Introduction

Stream API is one of the notable inclusions in Java 8 and later versions, besides lambda expressions. The new Stream API allows parallel processing in Java. It supports many sequential and parallel aggregate operations to process the data, while completely abstracting out the low-level multithreading logic.

Streams can help to express efficient, SQL-like queries and manipulations on data. They can also use lambda expressions, thus, producing more compact code.

`java.util.stream` package contains all the Stream API interfaces and classes. The `Stream` interface and `Collectors` class form the foundation of the Stream API. Primitive data type operations in collections are time consuming. To avoid this time delay, there are new classes available in the API. Some of the interfaces in the API include `IntStream`, `LongStream`, and `DoubleStream`.

Figure 13.1 shows a pictorial overview of the Stream interfaces.

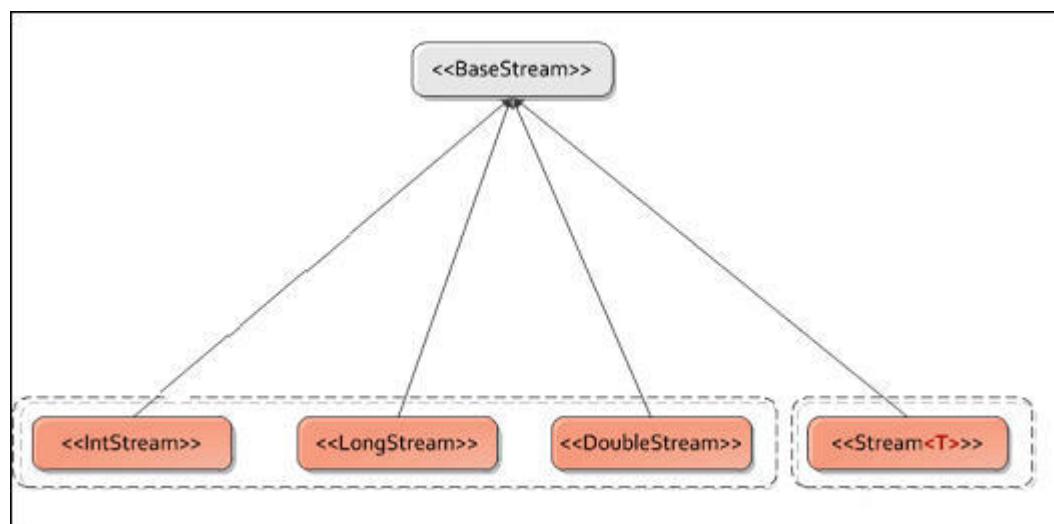


Figure 13.1: Stream Interfaces

13.2 Collections and Streams

In simplest terms, a stream is a series or set of elements that support sequential and parallel aggregate operations such as filtering, sorting, and so on.

A collection is a set of data, in the form of objects or elements. These data elements may also be filtered, sorted, and so on similar to stream elements. So, what is the difference between collections and streams?

Table 13.1 lists the differences between streams and collection.

Streams	Collections
Streams are fixed data structures that are computed on-demand.	A collection is an in-memory data structure to store values and it is mandatory that all values must be generated, before user access.

Streams	Collections
Streams are lazily implemented collections; a stream operates on user demand basis.	The characteristics of collections are completely opposite to the streams and they are a set of active computed values (irrespective of the user demand).
Streams focus on aggregations and computations.	Collections merely focus on holding data.
Data storage is not available in Stream; it functions on the source data structure (collection and array) and returns sequential data or data elements in series on which you can perform further operations. For example, a Stream can be created from a list and applied condition based filtering.	Contrary to streams, collections are actual data structures that contain or hold data in memory.
Streams are based on pipelining, formed through a data source and intermediate and terminal operations performed on the data.	Collections may not support pipelining.
Stream operations do not iterate explicitly, the iteration takes place behind the scenes.	Collections are iterated explicitly.
Stream operations are functional interface friendly; this makes functional programming using lambda expressions possible. However, this may also make processing slower.	Processing collection data is faster in most cases as compared to streams.

Table 13.1: Differences Between Streams and Collections

13.3 Generating Streams

There are many options available to generate a Stream in Java. One of these is to generate a Stream from a data source such as a class or interface inheriting from `java.util.Collection`. The Collection interface provides `stream()` and `parallelStream()` methods which are inherited by all implementing classes and sub-interfaces.

These methods are briefly described as follows:

- **stream():** Is used to get a sequential Stream with the collection as its source.

Code Snippet 1 shows the usage of `stream()`.

Code Snippet 1:

```
Stream<String> str = list.stream();
```

- **parallelStream():** Is used to get a possibly-parallel Stream lateral to the collection as its source.

Code Snippet 2 shows the usage of `parallelStream()`.

Code Snippet 2:

```
Stream<String> parStr = list.parallelStream();
```

The order of the `Stream` relies on the respective collection similar to an Iterator.

Some of the other ways to generate a `Stream` include static factory methods on the stream classes and interfaces, such as `Stream.of()`, `IntStream.range()`, and so on, `BufferedReader.lines()` and `Random.ints()`.

Consider some examples.

`BufferedReader` class of `java.io` package includes the `lines()` method that returns a `Stream`, as shown in Code Snippet 3.

Code Snippet 3:

```
import java.util.stream.*;
import java.io.*;
public class BufferedReaderDemo {
    public static void main(String[] args) throws FileNotFoundException,
    IOException {
        FileReader sampleFR = new FileReader("C:\\Misc\\rfile.txt");
        BufferedReader sampleBR = new BufferedReader(sampleFR);
        sampleBR.lines().forEach(System.out::println);
    }
}
```

Here, `SampleBR` is created as a `BufferedReader` instance that uses `lines()` method for a simple stream operation - reading and displaying data from a text file.

To read a file as a `java.util.stream.Stream` object using `Files.lines(Path filePath)`, the code will be as shown in Code Snippet 4. The `Files` class belongs to the `java.nio.file.Files` package.

Code Snippet 4:

```
try (Stream sampleST = Files.lines(Paths.get("D:\\random_file.txt"))) {
    sampleST.forEach(System.out::println);
}
```

Here, `sampleST` is the `Stream` instance. The `lines()` method is applied to get data from a given file path. As you can observe, this is a lazy implementation. Hence, it may not read the entire file when it is called.

Static methods on the `Files` class assist in navigating file trees using a `Stream`. Some of these are listed in table 13.2.

Method	Explanation
static Stream<Path> list(Path dir)	This method retrieves a Stream, whose elements include files in the specific directory.
static Stream<Path> walk(Path dir, FileVisitOption options)	This method retrieves a Stream that created by traversing the file tree starting at a specific file. FileVisitOption is an enumeration that defines file tree traversal options.
static Stream<Path> walk(Path dir, int maxDepth, FileVisitOption options)	This method retrieves a Stream that created by traversing the file tree depth-first starting at a specific file.

Table 13.2: Static Methods in the Files Class

You can stream text patterns using the Pattern class that contains a method, `splitAsStream(CharSequence)`. This method generates a Stream.

Code Snippet 5 shows a program that demonstrates use of the `splitAsStream()` method to generate a stream.

Code Snippet 5:

Code Snippet 5 generates a Stream from a simple text pattern that contains a comma as separator and separates the text into a Stream by using the `splitAsStream()` method. Then, each element in the Stream is printed out using a `forEach` loop. This was a simple case. In practical scenarios, you can use similar code to match and display large collections of strings.

Output:

Nathan

Ethan

Hank

Dennis

Sarah

13.4 Infinite Streams

An infinite stream is a sequence or collection of elements that has no limit. Infinite streams can be created using the `generate()` or `iterate()` static methods on Stream class. For example, an infinite quantity of objects can be created with `generate()` method. Code Snippet 6 shows this.

Code Snippet 6:

```
Stream.generate(() -> "*").forEach(System.out::println);
```

The `iterate()` method is similar to `generate()` with one difference, it gets the initial value and a function modifies that value. For example, a series of integers can be iterated as shown in Code Snippet 7.

Code Snippet 7:

```
Stream.iterate(2, n->2+2) // iterating stream
    .forEach(System.out::print); // result
```

Here, `iterate()` method is applied and the initial value 2 is iterated with the addition function until the program is stopped. This code produces the output '2468 10 12 14 ...' infinitely (until the program is forcibly stopped).

13.5 Stream Range

This approach allows creating ranges of numbers as streams.

The newly included primitive stream called `IntStream` can be used for Stream range calculation.

Code Snippet 8 describes the usage of static method, `range()` on the `IntStream` interface.

Code Snippet 8:

```
// to produce Stream range and display result
IntStream.range(2, 18).forEach(System.out::println);
```

Here, static method `range()` produces numbers between the given values in the stream operation. This code produces numbers between 2 to 18.

All primitive Stream interfaces such as `IntStream`, `DoubleStream`, and, `LongStream` contain corresponding `range` methods.

13.6 Operations on Streams

Tasks on streams are categorized as intermediate and terminal operations.

13.6.1 Intermediate Operations

In intermediate operations, operators (intermediate operators) apply logic, thus, the inbound Stream generates another stream. A Stream can contain 'n' number of intermediate operators, which has no limitations.

These operators can perform many operations such as filters, maps, and so on. Intermediate operations are lazy in nature, which means that they do not actually take place right away; rather they generate new Stream elements and send it to the next operation. The new Stream element is traversed when a terminal operation is encountered.

13.6.2 Terminal Operations

Intermediate operators can start a pipeline of Stream elements, however, to execute the process further, terminal operators are required in the stream. A terminal operator can be found at the end of the call stack and it performs the final operation to consume the Stream, which is the terminal operation. These operations return a result or produce a side effect.

Following are commonly used terminal methods:

- forEach
- toArray
- min
- max
- findFirst
- findAny
- anyMatch
- allMatch
- noneMatch

For example, `distinct()` is an intermediate operation. It retrieves a stream consisting of the distinct elements of the specified stream. However, traversal begins only when a `forEach()` iteration or similar terminal operation is performed.

Code snippet 9 demonstrates this.

Code Snippet 9:

```
public static void main(String[] args) {
    Stream.of("the", "cat", "sat", "on", "the", "mat", "on", "the", "floor")
        .distinct()
        .forEach(w -> System.out.println("Printing: " + w));
}
```

The output will be:

```
Printing: the
Printing: cat
Printing: sat
Printing: on
Printing: mat
Printing: floor
```

13.6.3 Short-circuiting Operations

These operations are not standalone operations such as intermediate operations or terminal operations. If an operation (intermediate or terminal) generates a finite Stream in an infinite Stream, it is known as short-circuiting operation. `limit()` and `skip()` are short-circuit intermediate operations.

If a terminal operation terminates in a limited time in an infinite Stream, it is called a short-circuiting terminal operation. For example, `anyMatch`, `noneMatch`, `allMatch`, `findFirst`, and `findAny` are short-circuiting terminal operations.

13.7 Map/Filter/Reduce with Streams

Map/Filter/Reduce methods implementations are allowed in lambda expressions and default methods. However, the standard Java library has already implemented these methods. You will now see what are these methods one by one.

Map:

This method is applied for mapping all the elements to its output.

Filter:

As the name implies, this method applied to choose a set of elements and to eliminate other elements based on the given instructions.

Reduce:

This method is applied to reduce elements based on given instructions.

Java supports many terminal operations including average, max, sum, and so on that return one value by combining the contents of a stream. Such operations are called reduction operations.

A simple example is shown in Code Snippet 10.

Code Snippet 10:

```
String outcome = scores.stream()
    .reduce((acc, score) -> acc + " " + score)
    .get();
```

Here, a set of scores is reduced to a single value.

Consider a more complex example where you require to find a member with highest leading score in a sports team. This can be done as shown in Code Snippet 11 using Stream.reduce() method.

Code Snippet 11:

```
MemberPoints highestMember =
names.stream().map(memberName -> new MemberPoints(memberName,
getPoints(memberName)))
    .reduce(new MemberPoints("", 0.0),
(sA, sB) -> (sA.memberPoints > sB.memberPoints) ? sA : sB);
```

13.8 Streams and ParallelArray

Array class of Java contains many functionalities for various array operations such as sort. The newly introduced Stream API (and updated Java library) allows all array operations through parallel arrays. For example, parallelSort(). Code Snippet 12 shows the sample usage of the parallel approach.

Code Snippet 12:

```
Arrays.parallelSort(sampleArray);
```

13.8.1 Limit

The `limit()` method can be applied to limit a `Stream` to a specified number of elements. This method is best recommended for sequential stream pipelines.

For instance, `limit(12)` can be applied to limit numbers below 12 as shown in Code Snippet 13.

Code Snippet 13:

```
Random sampleRand = new Random();
sampleRand.ints().limit(12)
.forEach(System.out::println); // to display the results
```

Here, `sampleRand` is used to return random integer values and `limit()` method is applied to limit the numbers.

The code is limited to display only 12 random numbers.

13.8.2 Sort

In addition to facilitating these operations, Stream API also contains another method to sort the `Stream`, the `sorted()` method. This method is similar to other intermediate methods on `Stream` such as `map`, `filter`, and `peek`.

Important characteristics of the `sorted()` method is 'lazy' execution. No process is started until a terminal operation (such as `reduce` or `foreach`) is called. A limiting operation must be called before the sorting operation on an infinite `Stream`. Code Snippet 14 demonstrates a sort operation.

Code Snippet 14:

```
sampleRand.ints().limit(12).sorted() // limit before sort
.forEach(System.out::println); // to display output
```

Here, the returning values are sorted using `sorted()` method and this method should be placed after the limit to produce the outputs.

Further, `sort()` can be applied after a `filter` call operation, Code Snippet 15 shows this in detail.

Code Snippet 15:

```
import java.io.*;
import java.util.*;
import java.nio.file.*;
public class Demo {
    public static void main(String[] args) throws IOException {
        Files.list(Paths.get(".")) // folder
            .map(Path::getFileName) // extended path
            .map(Path::toString) // String conversion
            .filter(name -> name.endsWith(".Jpeg"))
```

```

        .sorted() // sort the files alphabetically
        .limit(6) // first 6
        .forEach(System.out::println); // displays output
    }
}

```

This code displays the first six .Jpeg file names in the specific folder/directory.

The code operations are as follows:

- List the files in the specific folder/directory
- Map the files to names
- Find names that end with '.jpeg'
- Takes only the first six (sorted alphabetically)
- Displays output

13.9 Collectors

Streams are lazily implemented and support parallel operations. Thus, a specific approach is required to merge the elements as single output. This approach is called as a collector.

There are three different elements in a collector as follows:

- First, a supplier of an initial value.
- Second, an accumulator that adds to the initial value.
- Third, a combiner that combines two outputs as a single output.

There are two methods to implement this:

- collect(supplier, accumulator, combiner)
- collect(Collector)

There are several default collectors available. The statement to import collectors is shown in Code Snippet 16.

Code Snippet 16:

```
import static java.util.stream.Collectors.*; // to import collectors
```

13.9.1 Simple Collectors

Code Snippet 17 shows a complete example using simple collectors. Consider that you have a list of movie names and their release years and you want to display only the names from this list. This can be achieved using a collector.

Code Snippet 17:

```

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
class Movie {
    String name;
    int year;
    public Movie(String name, int year) {
        super();
        this.name = name;
        this.year = year;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getYear() {
        return year;
    }
    public void setYear(int year) {
        this.year = year;
    }
}
public class SimpleCollectorDemo {
    public static void main(String args[]) {
        // Create list of movies
        List<Movie> listOfmovies = createListOfMovies();
        // Use map(), collect(), and toList() to get a list of movie names
        List<String> listOfmovieNames = listOfmovies.stream()
            .map(s -> s.getName())
            .collect(Collectors.toList());
        listOfmovieNames.forEach(System.out::println);
    }
    public static List<Movie> createListOfMovies() {
        List<Movie> listOfmovies = new ArrayList<>();
        Movie m1 = new Movie("Coma", 1996);
        Movie m2 = new Movie("Peter Kong Goes to the Mall", 1975);
        Movie m3 = new Movie("Martin Eden", 2020);
        Movie m4 = new Movie("Clouds of Sils Maria", 2018);
        listOfmovies.add(m1);
        listOfmovies.add(m2);
        listOfmovies.add(m3);
        listOfmovies.add(m4);
        return listOfmovies;
    }
}

```

The code defines a simple collector using `toList()`. Here, `map` is an intermediate operation and consumes single element from the input Stream and produces single element to output Stream.

The output will be as follows:

```
Coma
Peter Kong Goes to the Mall
Martin Eden
Clouds of Sils Maria
```

13.9.2 Joining

Joining collector is similar to `StringUtil.join`. It merges the Stream using a provided delimiter.

Code Snippet 18 builds on the earlier example. Here, a semicolon separates the given set of names and they are combined as a single string using the joining collector.

Code Snippet 18:

```
String movieNames=listOfmovies.stream()
    .map(s -> s.getName())
    .collect(Collectors.joining("; "));
System.out.println(movieNames);
```

Here, `movieNames` is a string variable that will be assigned a string value based on the stream in which names are mapped and joined. Each movie name is separated by a semicolon and a space and is added to the `movieNames` string.

The output will be as follows:

```
Coma; Peter Kong Goes to the Mall; Martin Eden; Clouds of Sils Maria
```

13.9.3 Statistics

These types of collectors evaluate the provided values and produce a single value as output. For example, an average collector produces the average set of values.

Code Snippet 19 calculates the average length of non-empty lines in the file '`rfile.txt`'.

Code Snippet 19:

```
import java.util.*;
import java.util.stream.Collectors;
import java.nio.file.*;
import java.io.IOException;
public class StatisticsCollectors {
    public static void main(String args[]) throws IOException{
        System.out.println("Here's the Avg length value:");//displays result
        System.out.println(Files.lines(Paths.get("c:\\\\misc\\\\rfile.txt"))
            .map(String::trim)
            .filter(p -> !p.isEmpty())
            .collect(Collectors.averagingInt(String::length)))
        ;//averaging the lines
```

```

    }
}
}
```

In some cases, multiple statistics are required in a single collection. Streams are usually consumed when `collect` is called, it calculates multiple statistics at once. In such instances, using `IntSummaryStatistics` is recommended. This class can be imported using a simple import as shown in Code Snippet 20.

Code Snippet 20:

```
import java.util.IntSummaryStatistics; // import summary
```

Code Snippet 21 shows an example of `summaryStatistics` usage.

Code Snippet 21:

```
import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
public class AptechJavaStreamAPI {
    public static void main(String args[]) {
        List<Integer> aptechInt = Arrays.asList(77, 66, 888, 22, 33, 7,
        121, 89, 55);
        IntSummaryStatistics aptechStats =
            aptechInt.stream().mapToInt((x) ->x).summaryStatistics();
        System.out.println("A list of Random numbers: " + aptechInt);
        System.out.println("Highest number in the lot -" +
            aptechStats.getMax());
        System.out.println("Lowest number in the lot -" +
            aptechStats.getMin());
        System.out.println("Combined value of All: " +
            aptechStats.getSum());
        System.out.println("Average value of all numbers: " +
            aptechStats.getAverage());
    }
}
```

Output:

A list of Random numbers: [77, 66, 888, 22, 33, 7, 121, 89, 55]

Highest number in the lot -888

Lowest number in the lot -7

Combined value of All: 1358

Average value of all numbers: 150.88888888888889

Here, `aptechInt` is a `List` instance takes array of integers as a list and `aptechStats` is a `summaryStatistics` instance. This code returns the average value and also calculates the min, max, and count of the values from the set of elements.

Similarly, a `Stream` can be mapped to a primitive type and then, `summaryStatistics()` can be called as shown in Code Snippet 22.

Code Snippet 22:

```
IntSummaryStatistics sampleStats = Files.lines(Paths.get("c:\\\\misc\\\\rfile.txt"))
    .map(String::trim)
    .filter(p -> !p.isEmpty())
    .mapToInt(String::length)
    .summaryStatistics();
```

13.10 Grouping and Partitioning

Grouping (`groupingBy`) collector groups elements based on a given function. For instance, grouping a set of elements by the first letter of names is shown in Code Snippet 23.

Code Snippet 23:

```
import java.util.stream.Collectors;
import java.util.*;
class Movie {
    String name;
    int year;
    public Movie(String name, int year) {
        super();
        this.name = name;
        this.year = year;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getYear() {
        return year;
    }
    public void setYear(int year) {
        this.year = year;
    }
}
public class GroupingCollectorDemo {
    public static void main(String args[]) {
        // Create list of movies
        List<Movie> listOfmovies = createListOfMovies();
        // Perform grouping based on first character
        Map<Character, List<Movie>> objMap = listOfmovies.stream()
            .collect(Collectors.groupingBy(movie -> movie.getName().charAt(0)));
        // Iterate through the results and display the grouped entries
        for (Map.Entry<Character, List<Movie>> me : objMap.entrySet()) {
            Character key = me.getKey();
            List<Movie> valueList = me.getValue();
        }
    }
}
```

```

        System.out.println("\nKey: " + key);
        System.out.println("Values: ");
        for (Movies : valueList) {
            System.out.println(s.getName() + " ");
        }
    }
}

public static List<Movie> createListOfMovies() {
    List<Movie> listOfmovies=new ArrayList<>();
    Movie m1=new Movie("Coma",1996);
    Movie m2=new Movie("Peter Kong Goes to the Mall",1975);
    Movie m3=new Movie("Martin Eden",2020);
    Movie m4=new Movie("Clouds of Sils Maria",2018);
    listOfmovies.add(m1);
    listOfmovies.add(m2);
    listOfmovies.add(m3);
    listOfmovies.add(m4);
    return listOfmovies;
}
}

```

In Code Snippet 23, movie names from the list are grouped based on the first letter of each names. The outcome of the grouping is stored in a map comprising character and movie list values. To iterate through this map, we make use of `Map.entrySet()` method and we get both, key and value, in one go. As you can observe in the output, there are two movies beginning with C and they are grouped together.

Output:

```

=====
Key: P
Values:
Peter Kong Goes to the Mall
=====
Key: C
Values:
Coma
Clouds of Sils Maria
=====
Key: M
Values:
Martin Eden

```

Partitioning

Partitioning (`partitioningBy`) method is parallel to Grouping method and creates a map with a boolean key as shown in Code Snippet 24.

Code Snippet 24:

```
...
List<Movie> listOfmovies = createListOfMovies();
Map<Boolean, List<Movie>> objMap = listOfmovies.stream()
    .collect(Collectors.partitioningBy(movie -> movie.getYear() > 2000));
for (Map.Entry<Boolean, List<Movie>> me : objMap.entrySet()) {
    Boolean key = me.getKey();
    List<Movie> valueList = me.getValue();
    System.out.println("\nKey: " + key);
    System.out.println("Values: ");
    for (Movie s : valueList) {
        System.out.println(s.getName() + " ");
    }
}
...

```

Here, the code partitions the data into two sets based on whether a movie released prior to 2000 or not. This code can be used with the earlier example shown in Code Snippet 23.

Output:

```
Key: false
Values:
Coma
Peter Kong Goes to the Mall
Key: true
Values:
Martin Eden
Clouds of Sils Maria
```

13.10.1 Parallel Grouping

Parallel Grouping (`groupingByConcurrent`) executes grouping in parallel (without ordering). The Stream must be unordered to allow parallel grouping.

13.11 Using Functional Interfaces with Stream API

Functional interfaces can be used with several new APIs in Java including Date-Time and Stream.

Commonly used functional interfaces with Stream API are as follows:

- ➔ **Function and BiFunction:** Function denotes a function that gets one type of element and produces another type of element.

Function is the basic (generic) form, in which T is the input type and R is the output (result) type of the function. There are specific Function interfaces available to handle primitive types. Each of these represents a functional interface and hence, can be given as the assignment target in a

lambda expression or method reference.

Some of these functions (or functional interfaces) are as follows:

- ToIntFunction
- ToIntBiFunction
- ToLongFunction
- ToLongBiFunction
- LongToIntFunction
- LongToDoubleFunction
- ToDoubleFunction
- ToDoubleBiFunction
- IntToLongFunction

Following are the Stream methods in which Function or its primitive specialization is applied:

- <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)
- <R> Stream<R> map(Function<? super T, ? extends R> mapper)
- IntStreamflatMapToInt(Function<? super T, ? extends IntStream> mapper) – same for long and double
- <A> A[] toArray(IntFunction<A[]> generator)
- IntStreammapToInt(ToIntFunction<? super T> mapper) – same for long and double producing primitive specific.

- **Predicate and BiPredicate:** They denote a predicate against which arguments of the Stream are tested. They are applied to filter the arguments from the Stream. Similar to Function, there are primitive specific interfaces for int, long, and double.

Following are the Stream methods in which Predicate or BiPredicate methods are used:

- boolean noneMatch(Predicate<? super T> predicate)//to filter no match
- boolean anyMatch(Predicate<? super T> predicate)//to filter any match
- Stream<T> filter(Predicate<? super T> predicate)//filter in Stream
- Boolean allMatch(Predicate<? super T> predicate)//to filter all match

- **Consumer and BiConsumer:** It denotes an operation that accepts a single input element and produces no output. Some action can be performed with this on all the elements of the Stream.

A simple example is shown here in Code Snippet 25.

Code Snippet 25:

```

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
public class SampleDemo {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(
            new Employee("John Simmons", 350000),
            new Employee("Mark Smith", 413000),
            new Employee("Jane Weston", 344000),
            new Employee("Gillian Bush", 690000)
        );
        displayAllEmployee(employees, e -> {
            e.salary *= 1.5;
        });
        System.out.println("Salaries after increment:");
        displayAllEmployee(employees, e ->
            System.out.println(e.empname + ":" + e.salary));
    }
    public static void displayAllEmployee(List<Employee> emp,
                                         Consumer<Employee> printer) {
        for (Employee e : emp) {
            printer.accept(e);
        }
    }
    public static void display(List<Employee> emp,
                             Consumer<Employee> printer) {
    }
}
class Employee {
    public String empname;
    public long salary;
    Employee(String name, long sal) {
        this.empname = name;
        this.salary = sal;
    }
}

```

Following are Stream methods in which Consumer, BiConsumer, or its primitive interfaces are used:

- void forEach(Consumer<? super T> action)
- Stream<T> peek(Consumer<? super T> action)//peek action
- void forEachOrdered(Consumer<? super T> action)

- **Supplier:** It represents an operation that can generate new values in the Stream. The methods in Stream that gets Supplier elements are as follows:
- public static<T> Stream<T> generate(Supplier<T> s) // supplier in

Stream generation

- <R> R collect(Supplier<R>supplier, BiConsumer<R, T>accumulator, BiConsumer<R, R> combiner)//supplier in collect operation

Code Snippet 26 demonstrates the predicate interface.

Code Snippet 26:

```
import java.util.Arrays;//to use arrays
import java.util.List;//to use list
import java.util.function.Predicate;//to use Predicate
public class FuncInterExample {
    public static void main(String args[]) {
        List<Integer> newList = Arrays.asList(5, 10, 15, 20, 25, 30,
            35, 40, 45); // newList as integer values
        System.out.println("Displays the array list:");
        // passing the parameter as x
        eval(newList, x->true);
        System.out.println("Displays the odd numbers from the list:");
        eval(newList, x->x%1 == 0); // to display odd numbers from
        // the newList
        System.out.println("Displays the numbers smaller than 40:");
        eval(newList, x->x< 40); // to display the numbers below 40
    }
    public static void eval(List<Integer> newList,
        Predicate<Integer> newPred) {
        for(Integer x: newList) {
            if(newPred.test(x)) {
                System.out.println(x + " ");
            }
        }
    }
}
```

Output:

Displays the array list:

5
10
15
20
25
30
35
40
45

Displays the odd numbers from the list:

```
5
10
15
20
25
30
35
40
45
```

Displays the numbers smaller than 40:

```
5
10
15
20
25
30
35
```

13.12 Optional and Spliterator API

The `Optional` class and `Spliterator` interface defined in `java.util` package can be used with Stream API.

`Optional` is a container object that optionally contains a value (non-null). If it contains a value, `isPresent()` shows true and `get()` returns the value. Following are Stream terminal operations that return an `Optional` object:

- ➔ `Optional<T> min(Comparator<? super T> comparator) // minimum`
- ➔ `Optional<T> max(Comparator<? super T> comparator) // maximum`
- ➔ `Optional<T> reduce(BinaryOperator<T> accumulator) // to reduce`
- ➔ `Optional<T> findFirst() // to find first`
- ➔ `Optional<T> findAny() // to find any`

`Spliterator` interface is used to support the parallel execution. `Spliterator(trySplit)` method produces a new `Spliterator` that manages a subset of the elements of the original `Spliterator`.

13.13 Parallelism

Parallel computing includes splitting a task into sub-tasks, running those tasks simultaneously (in parallel, with each sub-task running in an individual thread) and then, merging the outputs of the sub-tasks into

a single output. Java includes the fork/join framework that aids stress-free implementation of parallel computing in various applications. In this framework, the user must specify the partition of tasks. Aggregate operations are automated; the Java runtime performs this partitioning and merging of tasks automatically.

Implementing parallelism in collection-based applications involves a possible difficulty. Since, the collections are not thread-safe, manipulating collection with multiple threads may cause thread interference errors or memory consistency errors.

To make the collections thread-safe, the Collections Framework provides synchronization wrappers that enables automatic synchronization to a collection and makes it thread-safe. However, synchronization includes thread contention and it prevents parallel processing of threads. Thus, it must be avoided.

As a solution for this, aggregate operations and parallel streams makes it possible to implement parallelism in non-thread-safe collections.

Faster performance of parallelism also depends on other external factors such as the processor. Implementing parallelism can be achieved easily by aggregate operations, however, the user must decide whether an application requires parallelism or not. Since, Java version 8 onwards supports parallelism, it executes with parallel streams. It divides streams into multiple sub-tasks and merge them as explained earlier. Invoking a parallel command on a Stream can implement parallelism in multi-cores.

Adding a `parallel()` method to the Stream instructs the library to deal with the complexities of threading. Thus, the library controls the process of forking.

13.13.1 Executing Streams in Parallel

Aggregate operations are implemented to combine the results. This process is known as concurrent reduction. Following conditions must be true for performing a collect operation in the process:

- The Stream must be parallel.
- The parameter of the collect operation, the collector, contains the characteristic `Collector.Characteristics.CONCURRENT`.
- Stream must be unordered or the collector must contain the `Collector.Characteristics.UNORDERED`.

Code Snippet 27 shows a complete program with various Stream API operations in detail.

Code Snippet 27:

```
import java.util.Arrays;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.stream.Collectors;
public class AptechJavaStreamAPI {
    public static void main(String args[]) {
        List<String> clientList=Arrays.asList("Amazon", "Etsy", "Frodos",
        "Wendys", "", "", "MaBeats", "Miniclip");
        System.out.println("The newClientList: "+clientList);
```

```

// To receive empty list count
long emptyCount = clientList.stream().filter(cList -> cList.
isEmpty()).count();
System.out.println("Result1:no. of Empty Strings: " + emptyCount);
// To receive company with character letter count length > 5
long lengthCount = clientList.stream().filter(x ->x.length() >
5).count();
System.out.println("Result2:no. of clients with name length > 5: " +
lengthCount);
// To receive the client name starts with letter 'A'
// and display count
long startCount = clientList.stream().filter(x ->x.startsWith("M")).count();
System.out.println("Result3:no. of clients which name starts with
letter M: " + startCount);
// To eliminate all empty Strings from List
List<String> removeEmptyStrings = clientList.stream().filter(x ->!x.
isEmpty()).collect(Collectors.toList());
System.out.println("Result4:no. New Client List without empty list" +
removeEmptyStrings);
// To display the client names with > 8 characters
List<String> newList = clientList.stream().filter(x ->x.length() >
8).collect(Collectors.toList());
System.out.println("Result5: New client list with letter count > 8: " +
newList + "\n");
List<Integer> aptechInt = Arrays.asList(77, 66, 888, 22, 33, 7, 121,
89, 55);
IntSummaryStatistics aptechStats = aptechInt.stream().mapToInt((x) ->
x).summaryStatistics();
System.out.println("^ A list of Random numbers: " + aptechInt);
System.out.println("Highest number in the lot --" + aptechStats.
getMax());
System.out.println("Lowest number in the lot --" + aptechStats.
getMin());
System.out.println("Combined value of All: " + aptechStats.getSum());
System.out.println("Average value of all numbers: " + aptechStats.
getAverage() + "\n");
// To convert a Message in UPPERCASE and join them
// using space
List<String> aptechTips = Arrays.asList("java8", "has", "some",
"great", "features");
String joinList = aptechTips.stream().map(x ->x.toUpperCase()).collect(Collectors.joining(" "));
System.out.println("- To Join and Display the message with UPPERCASE: " +
joinList);
// To display the cube value of the numbers
List<Integer> numbers = Arrays.asList(5, 10, 15, 20, 25);

```

```

List<Integer> cubes = numbers.stream().map(myInt -> myInt * myInt *
myInt).distinct().collect(Collectors.toList());
System.out.println("- Display the cube value of the numbers : " + cubes +
"\n");
}
}

```

Output:

The new Client List: [Amazon, Etsy, Frodos, Wendys, , , MaBeats, Miniclip]
Result1: no. of Empty Strings: 2
Result2: no. of clients with name length > 5: 5
Result3: no. of clients which name starts with letter M: 2
Result4: no. New Client List without empty list[Amazon, Etsy, Frodos,
Wendys, MaBeats, Miniclip]
Result5: New client list with letter count > 8: []
^ A list of Random numbers: [77, 66, 888, 22, 33, 7, 121, 89, 55]
Highest number in the lot -888
Lowest number in the lot -7
Combined value of All: 1358
Average value of all numbers: 150.88888888888889
- To Join and Display the message with UPPERCASE: JAVA8 HAS SOME GREAT
FEATURES
- Display the cube value of the numbers : [125, 1000, 3375, 8000, 15625]

Code Snippet 28 demonstrates use of Stream API for text manipulation and display. This example shows a set of stream operations with Australian state list.

Code Snippet 28:

```

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class JavaStreamAPIDemo {
    public static void main(String args[]) {
        // To diplay the number of empty values that contains no
        // state names
        List<String> stateList = Arrays.asList("Victoria", "", "Queensland",
        "", "Tasmania");
        long count = stateList.stream().filter(x -> x.isEmpty()).count();
        System.out.printf("Australian state List %s has %d empty values %n",
        stateList, count);
        // To display the state names with character length
        // more than 9
        long num = stateList.stream().filter(x -> x.length() > 9).count();
        System.out.printf("Australian state List %s has %d strings of length
        more than 9 %n", stateList, num);
    }
}

```

```

// To display the number of Australian states that starts
// with the letter V
count = stateList.stream().filter(x ->x.startsWith("V")).count();
System.out.printf("Australian state List %s contains %d state names
which starts with the letter 'V' %n", stateList, count);
// To display the Australian states list without empty
// values
List<String> filtered = stateList.stream().filter(x ->!x.
isEmpty()).collect(Collectors.toList());
System.out.printf("Australian state List : %s, without Empty any empty
values: %s %n", stateList, filtered);
// To display and convert BRIC nations to Lowercase and // merge them
with comma
List<String> BRIC = Arrays.asList("BRAZIL", "INDIA", "RUSSIA",
"CHINA");
String BRICNations = BRIC.stream().map(x ->x.toLowerCase()) .
collect(Collectors.joining(","));
System.out.println("BRIC Nations in Lowercase: " +BRICNations);
}
}

```

Here, `stateList` is a `List` instance and contains a string value as array. The code counts the empty string values in the list. `num` is a `long` variable applied to count the string values more than nine characters. `filter` method is applied to filter the results and display the states beginning with V.

Output:

```

Australian state List [Victoria, , Queensland, , Tasmania] has 2 empty
values
Australian state List [Victoria, , Queensland, , Tasmania] has 1 strings
of length more than 9
Australian state List [Victoria, , Queensland, , Tasmania] contains 1
state names which starts with the letter 'V'
Australian state List: [Victoria, , Queensland, , Tasmania], without
Empty any empty values: [Victoria, Queensland, Tasmania]
BRIC Nations in Lowercase: brazil, india, russia, china

```

13.14 Limitations of Java Stream API

Stream API includes many new methods to execute aggregate operations on list and arrays; however, there are also some limitations in this API. They are described as follows:

- ➔ Stateless lambda expressions: If parallel Stream and lambda expressions are stateful, it will produce a random set of output. Code Snippet 29 demonstrates this in detail.

Code Snippet 29:

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

```

```

import java.util.stream.Stream;
public class AptechJavaStreamLimit {
    public static void main(String[] args) {
        // A set of numbers in a proper order
        List<Integer> randomset =
            Arrays.asList(31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45);
        List<Integer> result = new ArrayList<Integer>();
        Stream<Integer> stream = randomset.parallelStream();
        // To display the number set in a random manner
        stream.map(set -> {
            synchronized (result) {
                if (result.size() < 40) {
                    result.add(set);
                }
            }
            return set;
        }).forEach(change -> {});
        System.out.println("The jumbled number set: " + result);
    }
} // the number set order will be displayed completely changed

```

Output:

The jumbled number set: [40, 41, 39, 38, 44, 45, 42, 43, 35, 34, 31, 32, 33, 37, 36]

This code example produces a jumbled set of numbers, as it is based on the Stream implementation and there is no specific order instructed for parallel. Moreover, this issue can be avoided in the sequential Stream.

- Once a Stream is consumed, it cannot be used later on as shown in Code Snippet 30.
- There are a vast number of methods in the Stream API and many of these are overloaded as well. This makes learning time-consuming and cumbersome.

13.15 Improvements and New Features in Stream API

Java 9, Java 12, and later versions introduced several improvements and new features in Stream API.

13.15.1 Stream API Improvements

Several new methods were introduced in Java 9. Since Stream is an interface, methods added to it are default and static. Table 13.3 lists these methods along with examples.

Assume that following import statement is used in all the examples shown in Table 13.3:

```
import java.util.stream.*;
```

Method	Description	Example
dropWhile()	This is a default method and drops all elements of the stream until given predicate fails.	<pre>public class StreamDemo { public static void main(String[] args) { Stream<Integer> mystream = Stream.of(18, 72, 55, 90, 100); mystream.dropWhile(num -> num < 50).forEach(num -> System.out.println(num)); } }</pre>
takeWhile()	This is a default method and works opposite to dropWhile(). This method takes all elements of the stream in the resulted stream until the predicate fails. In short, when the predicate fails, it drops that element and all the elements that comes after that element in the stream.	<pre>public class StreamDemo { public static void main(String[] args) { Stream<Integer> mystream = Stream.of(18, 72, 55, 90, 100); mystream.takeWhile(num -> num < 50).forEach(num -> System.out.println(num)); } }</pre>
iterate()	This is a static method and has three arguments, namely: → Initializing value: the returned stream starts with this value. → Predicate: the iteration continues until this predicate returns false. → Update value: updates the value of previous iteration.	<pre>public class StreamDemo { public static void main(String[] args) { IntStream.iterate(1, num -> num < 30, num -> num*5).forEach(num -> System.out.println(num)); } }</pre>
ofNullable()	This is a static method and is introduced to avoid NullPointerException. This method returns an empty stream if the stream is null. It can also be used on a non-empty stream where it returns a sequential stream of single element.	<pre>public class StreamDemo { public static void main(String[] args) { Stream<String> stream1 = Stream.ofNullable(null); stream1.forEach(str -> System.out.println(str)); Stream<String> stream2 = Stream.ofNullable("Oranges"); stream2.forEach(str -> System.out.println(str)); } }</pre>

Table 13.3: New Methods in Java 9 for Stream API

13.15.2 Teeing Collector

The teeing collector was introduced in Java 12. A Collector that is a composite of two downstream collectors is the return value of the teeing collector. Every element passed to the resulting collector is processed by both downstream collectors and then, their results are merged using specified merge function into the final result. In simple words, it is just a helper method added to `java.util.stream.Collectors` class which helps in reducing the verbosity of code when you want to combine collectors.

Syntax:

```
public static <T,R1,R2,R> Collector<T,?,R> teeing(Collector<? super T,?,R1>
downstream1, Collector<? super T,?,R2> downstream2, BiFunction <? super R1,?
super R2,R> merger)
```

Code Snippet 30 demonstrates use of a teeing collector.

Code Snippet 30:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collector;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class TeeingDemo {
    public static void main(String[] args) {
        Collector<CharSequence, ?, String> joiningCollector =
            Collectors.joining(":");
        Collector<String, ?, List<String>> listCollector =
            Collectors.toList();
        Collector<String, ?, String[]> compositeCollector =
            Collectors.teeing(joiningCollector, listCollector,
                (joinedString, strings) -> {
                    ArrayList<String> list = new ArrayList<>(strings);
                    list.add(joinedString);
                    String[] array = list.toArray(new String[0]);
                    return array;
                });
        String[] strings = Stream.of("Pink", "Blue", "Yellow", "Red") .
            collect(compositeCollector);
        System.out.println(Arrays.toString(strings));
    }
}
```

Here, in the code, we first build the logic to join any given string parameters using a combination of several different methods. Then, four strings representing color names are passed as a stream to the collect method, which in turn, invokes the logic and returns the joined string as well as individual strings as an array.

Output:

```
[Pink, Blue, Yellow, Red, Pink:Blue:Yellow:Red]
```

13.16 Check Your Progress

1. Which among the following defines parallel processing?

(A)	Adding tasks in sequence	(C)	Dividing a bigger task into a smaller sub-tasks and executing them at the same time
(B)	Sequence of bigger tasks	(D)	Sequence of smaller tasks

2. What is an infinite stream?

(A)	Sequence or collection of elements that has no limit	(C)	Stream which can only perform string operations
(B)	Sequence or collection that contains limited elements	(D)	Stream which can perform only parallel operations

3. Which among the following is an operation that accepts a single input element and produces no output?

(A)	Function and BiFunction	(C)	Supplier
(B)	Consumer and BiConsumer	(D)	Predicate and BiPredicate

4. From the following, which option does Spliterator interface support in the Stream API?

(A)	Sequential execution	(C)	Parallel execution
(B)	Intermediate operation	(D)	Short-circuiting operation

5. Which among the following is a definition for short-circuiting operation?

(A)	Produce new stream elements and send it to the next operation	(C)	Final operation to consume the stream
(B)	Accept a single input element and produces no output	(D)	Produce a finite stream in an infinite stream

13.16.1 Answers

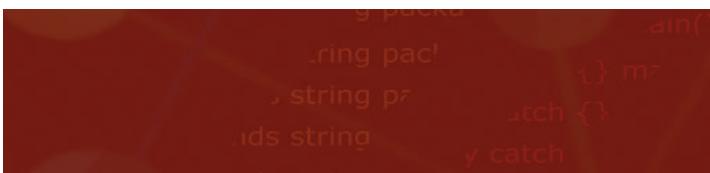
1.	B
2.	A
3.	A
4.	D

```
g package;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        List<String> stringList = Arrays.asList("apple", "orange", "banana", "kiwi");
        stringList.stream()
            .filter(s -> s.startsWith("a"))
            .map(s -> s + " juice")
            .forEach(s -> System.out.println(s));
    }
}
```

Summary

- The new Stream API in Java allows parallel processing. It supports many sequential and parallel aggregate operations to process the data.
- `java.util.stream` package contains all the Stream API interfaces and classes.
- The Stream interface and Collectors class form the foundation of the Stream API. Some of the interfaces in the API include IntStream, LongStream, and DoubleStream.
- There are several differences between collections and streams.
- Streams are lazily implemented and support parallel operation. Thus, it requires a specific approach to merge elements as single output, this approach is called as collector.
- Function denotes a function that gets one type of element and produces another type of element. Function is the basic form, in which T is the input type and R is the output type of the function.
- Tasks on streams are categorized as intermediate and terminal operations.
- The Optional class and Spliterator interface defined in `java.util` package can be used with Stream API.
- Commonly used functional interfaces with Stream API include Function and BiFunction, Predicate and BiPredicate, Consumer and BiConsumer, and Supplier.
- Newer Java versions from 9 onwards have introduced improvements and new features in the Stream API.



Try it Yourself

1. Given a list of names, display the total number of letters in all the names with more than five letters. Use the Stream API classes/interfaces to do this.
2. You are given a list of person names and their ages. Determine the oldest person in the list using streams.
3. You are given a list of person names and their ages. Partition these people into adults and kids.

Session - 14

More on Functional Programming

Welcome to the Session, **More on Functional Programming**.

This session explores advanced concepts of functional programming in Java 8. It explains features such as immutability and concurrency. Finally, this session describes about recursion in Java.

In this Session, you will learn to:

- Explain functional interfaces
- Describe immutability in Java
- Define and explain concurrency in Java
- Explain Recursion in Java



14.1 Introduction to Functional Interfaces

Functional interfaces are one of the key elements to implement functional programming in Java. The newly introduced `java.util.function` package in Java 8 defines functional interfaces that can supply target types for elements such as lambda expressions and method references. A functional interface has only one abstract method (which is called as a functional method) and zero or more default methods.

14.1.1 Function

`Function<T, R>` is one of these in-built functional interfaces included in the package. The main purpose of `Function<T, R>` is to map scenarios, for example, when an object of a specific type is given as input and it is converted (or mapped) into different types. One of the common uses of `Function` is in streams wherein the `map()` function of a stream takes an instance of `Function` to convert the stream of one type to a stream of different type.

As `Function<T, R>` is a functional interface, it can be applied as an assignment target for a lambda expression or a method reference.

A Function Descriptor is a term that describes the signature of the abstract method of a functional interface. It is important to note that the signature of this method is similar in syntax to the signature of a lambda expression.

Function Descriptor of `Function<T, R>`:

The function descriptor for `Function<T, R>` will be `T -> R`.

Here, object of type `T` is given as input to the lambda and it produces an object of type `R` as output value.

Following points are to be noted for the `Function` interface:

- Method `apply()` is the prime abstract functional method of `Function` interface. It takes a parameter `t` of type `T` as input and yields an output object of type `R`.
- `Function<T, R>` contains two default methods.
- The first default method is `compose()`, that combines the function on which it is activated (which is also mentioned as the current function) with another function that is named `before()`.

If the `before` function is applied after the `combined` function is applied, then the input type will change from type `V` to type `T`. Then, the current function alters the type `T` object to type `R` as output. Accordingly, the combined function is returned as a result, whereas, `compose()` utilizes both the functions, in the course of converting type `V` to `R`.

- The second default method is `andThen()`, combines the function on which it is activated (current function) with other functions named `after`, so that while the combined function is called, at that point initially the current function is activated which changes the input type `T` to type `R`. Then, the `after` function is applied which changes from type `R` to `V`. Accordingly, the combined function is attained by using `andThen()` default method utilizes both functions within the process to convert type `T` to type `V`.
- `Function <T, R>` also consists of a static method `identity()` which is a plain function, that returns the input as it is. As the name implies, functional programming is created on functions as a

first-class feature.

- The Function interface (and other associated interfaces such as IntFunction, DoubleFunction, LongFunction, BiFunction, and so on that are defined in the java.util.function package) lets the functions to be accepted as arguments, stored as variables, and be returned by methods.

To summarize once again, default methods of the Function interface are as follows:

- **andThen(Function):** Returns a composed function that initially applies this function to its input and later applies the specified function to the output.

Syntax:

```
default <V> Function<T,V> andThen(Function<? super R,> ? extends V> after)
```

where,

T – Is the type of input to the function

R – Is the type of result of the function

V – Is the type of output of the after function and of the composed function

after – Is the function to apply after this function is applied

- **compose(Function):** Comparable to andThen(), however in reversed sequence (formerly applies the specified function to its input and later this function).

Syntax:

```
default <V> Function<V,R> compose(Function<? super V,> ? extends T> before)
```

where,

R – Is the type of input to the function

V – Is the type of output of the before function and input to the function

T- Is the output of the composed function

before – Is the function to apply before this function is applied

- **identity():** Returns a function which constantly returns its input argument.

Syntax:

```
static <T> Function<T,T> identity()
```

where,

T- is the type of input and output to the function

These methods can be used as a chain for creating a function as shown in Code Snippet 1.

Code Snippet 1:

```
Function<integer, String> sampleF = Function.<integer>identity()
    .andThen(i -> 2 * i).andThen(i -> "sampleStr" + i);
```

Resultant function will acquire an integer, multiply it by 2 and finally, prepend "sampleStr" to it.

To create a single function, `andThen` function can be applied countless times. Similarly, functions can be passed as well as returned from methods. Code Snippet 2 shows the new Date-Time API being used with `Function`.

Code Snippet 2:

```
public Function<LocalDate, LocalDateTime> displayDateTime(
    final Function<LocalDate, LocalDate> test)
{
    return test.andThen(d -> d.atTime(5, 4));
}
```

This method would take in a function that works on a `LocalDate` and change it into a function `LocalDateTime` (with a time of 5:04 am) as output.

The complete program defining and utilizing this method `displayDateTime()` is shown in Example 1.

Example 1:

```
import java.util.function.*;
import java.util.stream.*;
import static java.util.stream.Collectors.*;
import java.time.*;

public class FunctionDemo {

    public static void main(String[] args) {
        Function<LocalDate, LocalDateTime> plusTwoM = Function.<LocalDate>
            identity()
            .andThen(displayDateTime(d -> d.plusMonths(2)));
        System.out.println(Stream.iterate(LocalDate.now(), d ->
            d.plusDays(1))
            .limit(10)
            .map(plusTwoM)
            .map(Object::toString)
            .collect(joining(", ")))
    }
}

public static Function<LocalDate, LocalDateTime> displayDateTime(
    final Function<LocalDate, LocalDate> test) {
    return test.andThen(d -> d.atTime(2, 2));
}
```

The code iterates through, manipulates, and displays dates and times two months from today's date using functional interfaces and a variety of methods such as `limit()`, `map()`, and `collect()`.

14.1.2 Currying

Currying is a concept related to functional programming in which a function f of two arguments (consider a and b) is created as an alternative for a function g of one argument that returns a function. The output returned by the latter function is the same as the value of the original function.

In simple terms, currying is the process of transforming a function having multiple arguments into a function with a single argument. This argument is the value of the first argument from the original function. The function returns another function that has only one argument. This will receive the second original argument and itself return a function that has only one argument. This chain continues over the number of arguments of the original. The last function in the chain will get access to all arguments and can perform whatever must be done.

Code Snippet 3 demonstrates a simple example.

Code Snippet 3:

```
f(a, b) = (g(a))(b)
```

Here, f is a function, a and b are arguments.

If fewer arguments (not all) are passed to a function, it is referred as partially applied function. Unit conversions are the perfect example for this.

Unit conversion constantly encompasses a conversion factor from time to time and a baseline adjustment factor. For example, the formula to convert Kilograms to Pounds, that is, kg to lbs (x) = $x * 2.2046$.

Following is the basic pattern of all unit conversions:

- Multiply by the conversion factor
- Adjust the baseline if relevant

Code Snippet 4 expresses the basic pattern of unit conversion.

Code Snippet 4:

```
static double converter(double a, double e, double y) {
    return x * e + y;
}
```

Conversion methods requires many conversions between the same pair of units, for example, Celsius to Fahrenheit. However, these methods have to perform calculations with three values between two units. This can cause manual input errors in some cases.

However, an easy way to benefit from the existing logic is to enhance it with a curry converter. Code Snippet 5 shows a curry-converter usage. In Java 8, currying can make use of lambda expressions.

Code Snippet 5:

```
static DUOcurryConverter(double e, double b) {
    return (double a) -> a * e + b;
}
```

Here, the code is more flexible and it reuses the existing conversion logic. Instead of passing all the arguments `a`, `e`, and `b` all at once to the converter method, the arguments `e` and `b` are called to return another function, which when given an argument returns `a * e + b`. The method enables to reuse the conversion logic and create different functions with different conversion factors.

Conversion factor and baseline (`e` and `b`) is passed to the code which returns a function (of `a`) to do exactly what is expected.

For example, one can utilize converters as follows:

- DoubleUnaryOperator convert kgtolbs = curriedConverter(0, 2.2046);
- DoubleUnaryOperator convert GBPtoEUR = curriedConverter(1.268, 0);
- DoubleUnaryOperator convert CtoF = curriedConverter(9.0/5, 32);

Code Snippet 6 shows an example. `DoubleUnaryOperator` defines a method `applyAsDouble()` which is used here.

Code Snippet 6:

```
double gbp = convertUSDtoGBP.applyAsDouble(1000);
```

Example 2 defines a `compose()` method that uses currying.

Example 2:

```
import java.util.function.Function;
public class AptechJavaCompose {
    public static<x,y,z> Function<x,z> compose(Function<y,z> g,
    Function<x,y> h) {
        return o->g.apply(h.apply(o));
    }
    public static void main(String[] args) {
        //Create instance named sin_asin
        Function<Double,Double> sin_asin = compose(Math::sin, Math::asin);
        //To display the result, call apply on the instance
        System.out.println(sin_asin.apply(0.6));
    }
}
```

Here, `compose()` is defined as a static method that takes in a `Function` and returns a `Function`. Within the method, the `apply()` method is invoked on the argument `g` and then, on `h` using a lambda expression. In the `main()` method, the `compose()` function is called.

Output:

0.6

Example 3 further demonstrates `compose()` in detail.

Example 3:

```
import java.util.function.BiFunction;
import java.util.function.Function;
public class JavaCurry {
    public void curryfunction() {
        // Create a function that adds 2 integers
        BiFunction<Integer, Integer, Integer> adder = (x, y) ->x + y;

        Function<Integer, Function<Integer, Integer>> currier = x ->y
        -> adder.apply(x, y);

        Function<Integer, Integer> curried = currier.apply(5);

        // To display Results
        System.out.printf("Curry : %d\n", curried.apply(2));
    }
    public void compose() {
        // Function to display the result with + 4
        Function<Integer, Integer> addFour = (x) ->x + 4;
        // function to display the result with * 5
        Function<Integer, Integer> timesFive = (x) ->x * 5;
        // to display the result with n number of times using compose
        Function<Integer, Integer> compose1 = addFour.compose(timesFive);
        // to display the result with add
        Function<Integer, Integer> compose2 = timesFive.compose(addFour);
        // To display the end result
        System.out.printf("Times then add: %d\n", compose1.apply(7));
        // (7 * 4) + 5
        System.out.printf("Add then times: %d\n", compose2.apply(7));
        // (7 + 5) * 4
    }
    public static void main(String[] args) {
        new JavaCurry().curryfunction();
        new JavaCurry().compose();
    }
}
```

Output:

```
Curry: 7
Result as Times then add: 39
Result as Add then times: 55
```

14.2 Immutability

State cannot always be determined safely and for that reason, it is avoided whenever possible, in functional programming. Hence, immutable (unchangeable) data structures are chosen.

Immutability is the capability of an object to resist or prevent change. It is one of the core concepts in functional programming. If the state of an object cannot change after it is constructed, it is considered immutable. For example, String is an immutable type in Java.

Immutable objects are specifically valuable in concurrent applications. They cannot be corrupted by thread interference or perceived in an unpredictable state, as they cannot change state.

Example 4 demonstrates the concept of immutability with a basic example.

Example 4:

```
public class Main {
    public static void main(String[] args) {
        String sample = "immutable";
        System.out.println(sample); // immutable
        change(sample);
        System.out.println(sample); // immutable
    }
    public static void change(String str) {
        str = "mutable";
    }
}
```

Output:

```
immutable
immutable
```

The program in Example 4 displays the same String even after the `change()` method has been invoked. So, the String literal remains unchanged. This is because a String object in Java is considered as immutable.

Objects which are immutable cannot have their state changed after they have been created.

Following are the three chief objectives to use immutable objects often, which can also be applied to lessen the number of bugs that are introduced in the code:

- If it is known that an object's state cannot be changed by another method, then it is easier to reason about how your program works.
- Immutable objects are thread-safe by default.
- Immutable objects can be used as keys in a `HashMap` (or similar), as they have the same hash code forever. The table entry in a hash table would be effectively lost, if the hash code of an element in the table changed, as efforts to find it in the table would result in looking in the incorrect place. This is the main reason that String objects are immutable - they are often used as `HashMap` keys.

Java 8 onwards, Date-Time classes are immutable and practically everything added in Java 8 onwards is immutable (Optional and Streams for example).

Immutable class

Immutable class is a class in which the state of its instances does not change while it is constructed.

Following are some of immutable classes in java such as `java.lang.String`, `java.lang.Integer`, `java.lang.Float` and `java.math.BigDecimal`.

Immutable class eradicates the prospect of data turning inaccessible when used as keys in Map and Set. Immutable object must not change its state, when it is in the collection.

Implementing an immutable class

It is easy to implement an immutable class.

Following are some guidelines on implementing an immutable class correctly:

- The class requires to be specified as a final class. `final` classes cannot be extended.
- All fields in the class have to be defined as `private` and `final`.
- Do not define any methods that can alter the state of the immutable object. In addition to Setter methods, it is also applicable to any other methods that can alter the state of the object.

These guidelines must be considered while creating an immutable object. Do not return the mutable references to the caller.

You must take care while using Java's functional patterns to ensure that the code does not accidentally move into the mutable approach. For example, following type of code as shown in Code Snippet 7 has to be avoided.

Code Snippet 7:

```
int[] sampleCount=newint[1];
list.forEach(sampleItems->{
    if(sampleItems.isRed())myCount[0]++;
})
```

This type of code may cause mutation, hence, it should be handled as shown in Code Snippet 8.

Code Snippet 8:

```
list.stream().filter(sampleItems::isRed).count();
```

Whenever, resorting to mutability, consider using some combinations of '`filter`', '`map`', '`reduce`', or '`collect`' instead.

14.3 Concurrency

Concurrency is multiple processes can start, run, and finish in overlapping time periods. Optionally, the processes can finish at the same time. For example, a single-core computer performs multitasking.

Functional programming paves a strong base for concurrent programming, besides which Java also supports concurrency in several diverse ways.

One among those ways is the `parallelStream()` method on Collection. It offers a rapid mode to use a Stream simultaneously. It should be used cautiously as excessive concurrency can slow down the application.

An alternate way in which Java 8 supports concurrency is through the new `CompletableFuture` class. One of its methods is the `supplyAsync()` static method that accepts an instance of the functional interface `Supplier`. It also has the method `thenAccept()` which accepts a `Consumer` that manages completion of the task. The `CompletableFuture` class calls on the specified supplier in a diverse thread and executes the consumer when it is complete.

14.4 Recursion

Recursion is a programming language feature supported by various languages including Java. The practice of outlining something concerning itself is termed as recursion. In Java programming, recursion is the feature that permits a method to call itself. A method that calls itself is known as recursive.

Example 5 displays how to produce Fibonacci numbers with recursive lambda.

Example 5:

```
import java.lang.Math;
import java.util.Locale;
import java.text.NumberFormat;
import java.text.DecimalFormat;
import java.util.stream.IntStream;
import java.util.stream.DoubleStream;
import java.util.function.IntUnaryOperator;

/*Aptech Java FPJ
*Recursion Sample*/

public class FibboRecursion{
    static IntUnaryOperator fibonacci;
    public static void main(String[] args) {
        System.out.println("Fibonacci Number Sequence:");//output
        IntStream.range(0,15)// to display how many fibonacci numbers

        .map(fibonacci = f -> {

            return f == 0 || f == 1
                ? 1
                : fibonacci.applyAsInt(f - 2) + fibonacci.applyAsInt(f - 1);
        })
        .parallel()
        .forEachOrdered(g ->System.out.printf("%s ", g));
    }
}
```

Output:

Fibonacci Number Sequence:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Recursion versus iteration

Typical functional programming languages are not bundled with iterative constructs such as `while` and `for` loops. The reason is that such constructs are often a hidden invitation to use mutation.

For example, the condition in a `while` loop must be updated; otherwise the loop would execute zero or an infinite number of times. However, in many cases, loops work well. It is acceptable to mutate local variables. Using the `while` loop with an iterator in Java is shown in Code Snippet 9.

Code Snippet 9:

```
mangoes { } pass into the Iterator shown here:  
Iterator<Mango> th=mangoes.iterator();  
while (th.hasNext()) {  
    Mango mango=th.next();  
    // ...  
}
```

Here, it is not an issue because the mutations (both changing the state of the Iterator with the `next` method and assigning to the variable `mango` inside the `while` body) are not visible to the caller of the method where the mutations occur.

However, using `for-each` loop, such as a search algorithm, is problematic because the loop body is updating a data structure that is shared with the caller as shown in Code Snippet 10.

Code Snippet 10:

```
public void searchForPlatinum(List<String> l, Stats bunch) {  
    for(String p: l){  
        if("platinum".equals(p)){  
            bunch.incrementFor("platinum");  
        }  
    }  
}
```

The body of the loop has a side effect that cannot be neglected as functional style; it mutates the state of the `stats` object, which is shared with other parts of the program.

This is the main reason that pure functional programming languages avoid such problematic operations completely. In Java, every program can be rewritten to avoid iteration by using recursion instead, which does not require mutability. Using recursion eliminates the requirement of iteration variables.

Code Snippet 11 calculates the factorial function in an iterative approach whereas Code Snippet 12 implements same outcome in recursive approach.

Iterative factorial

Code Snippet 11:

```
static int factIter(int j) { //iterative approach
    int k=1;
    for (int h=1; h<=j; h++) {
        k *= h;
    }
    return k;
} //result
```

Here, the code demonstrates a standard loop-based form; the variables `k` and `h` are updated and iterated.

Recursive factorial

Code Snippet 12:

```
static long factRecur(long i) { //recursive approach
    return i==1 ? 1 : i * factRecur(i-1);
} //result
```

Here, the code demonstrates a recursive definition (the function calls itself) in a simple form.

In general, making a recursive function call is much more expensive than the single machine-level branch instruction required to iterate.

14.5 Check Your Progress

1. What is the main purpose of Function<T, R>?

(A)	Load methods	(C)	Support parallelism
(B)	To map scenarios	(D)	To get scenarios

2. What are the two default methods in Function<T, R>?

(A)	compose() and before()	(C)	hasNext() and andThen()
(B)	after() and andThen()	(D)	hasNext() and after()

3. Which among following options states a scenario where Immutable objects are exclusively advantageous?

(A)	In concurrent applications	(C)	In sequence
(B)	In methods	(D)	In an Interface

4. What type of errors does iteration cause in improper use cases?

(A)	Syntax errors	(C)	Compiler error
(B)	Runtime errors	(D)	Stackoverflow errors

14.5.1 Answers

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

Summary

- Functional interfaces are one of the key elements to implement functional programming in Java.
- Functional interfaces are defined in `java.util.function` package, which is new to Java 8.
- A functional interface has only one abstract method (which is called as a functional method) and zero or more default methods.
- Currying is a concept in which, a function f of two arguments (consider a and b) is realized created as an alternative for a function g of one argument that returns a function.
- State cannot always be determined safely and for that reason, it is avoided whenever possible, in functional programming.
- Immutability is the capability of an object to resist or prevent change. It is one of the core concepts in functional programming.
- In Java programming, recursion is the feature that permits a method to call itself. A method that calls itself is known as recursive.

```
 9. package com.aptech;
10. import java.util.*;
11. public class Main {
12.     public static void main(String[] args) {
13.         String str = "DOMINICA";
14.         System.out.println(moveright(str, 5));
15.     }
16.     public static String moveright(String str, int n) {
17.         if (n < 0) {
18.             return str;
19.         }
20.         if (n == 0) {
21.             return str;
22.         }
23.         return str.substring(n) + str.substring(0, n);
24.     }
25. }
```

Try it Yourself

1. Write a recursive method in Java to move a string by n characters to the right. For example, moveright("DOMINICA",5) must give the output as "ICADOMINI".
2. Write a program using Java functional programming features to demonstrate the use of CompleteableFuture class to print the multiples of the number 100 until 200.

Session - 15

Additional Features of Java

Welcome to the Session, **Additional Features of Java**.

This session lists some of the deprecated or removed features between Java 9 to 15. The session explains additional features of Java such as new mathematical functions in Java, the new module system introduced from Java 9 onwards, and so on. The session also covers hidden and sealed classes.

In this Session, you will learn to:

- List some of the deprecated or removed features between Java 9 to 15
- Explain the new mathematical functions in Java
- Explain Java modules
- Describe hidden and sealed classes



15.1 Deprecated Features in Recent Versions of Java

Between Java 9 to 15, several features that existed in earlier versions have been deprecated or removed now and no longer recommended for use. Some of these include:

→ **Removal of constructors of classes corresponding to primitive types**

Classes such as Boolean, Byte, Short, Character, Integer, Long, Float, and Double are considered as 'box' classes, because they can use autoboxing. The constructors of these classes have been deprecated.

→ **Removal of Nashorn Scripting engine and jjs tool**

Nashorn was the JavaScript scripting engine introduced in Java 8. It was valid for use between Java 8 to 10, but was deprecated in Java 11. The main goal of Nashorn was to provide a lightweight and high-performance JavaScript runtime in Java with a native JVM. jjs was the command line tool used with Nashorn.

→ **Internal garbage collector options**

Several garbage collector options have been removed (such as Concurrent Mark Sweep (CMS) Garbage Collector).

→ **Removal of var as a valid class name**

From Java 10 onwards you cannot use var as a class name.

→ **Removal of underscore as a valid identifier**

From Java 9 onwards, you cannot use underscore as a valid identifier

15.2 Mathematical Methods in Java

The Math class in Java contains several built-in methods to perform basic numeric calculation/operations such as the elementary exponential, logarithm, square root, and trigonometric methods. It is located in the `java.lang` package. Advanced mathematical operations can also be performed with Math class. For example, finding minimum or maximum of given values, rounding values, square root, and so on are also possible.

15.2.1 Basic Math Methods

Let us now see basic math methods one by one.

`Math.abs()`

This method produces the absolute value of the given input in the calculation. `Math.abs()` only returns the positive value, even if the given input contains negative values. It removes the negative values

and returns the positive values as output. Code Snippet 1 demonstrates a complete program with the `Math.abs()` method.

Code Snippet 1:

```
public class BasicMathDemo {
    public static void main(String args[]) {
        int abs1 = Math.abs(10); // abs1 = 10
        int abs2 = Math.abs(-20); // abs2 = 20
        System.out.println("Result A: " + abs1);
        System.out.println("Result B: " + abs2);
    }
} // displays result
```

Here, the code produces output as the same value as the input. However, the negative value (-20) is returned as positive value (20).

The `Math.abs()` method can be overloaded in following four ways:

- int
- double
- long
- float

Usage of these methods may depend on the respective parameters and operations performed.

Math.ceil()

This method rounds a floating-point value close to the integer value. The rounded value is returned as a double. Code Snippet 2 shows use of `Math.ceil()`.

Code Snippet 2:

```
public class BasicMathDemo {
    public static void main(String args[]) {
        double objCeil = Math.ceil(6.454);
        System.out.println("Result as " + objCeil);
    }
} // displays result
```

After executing this code, `objCeil` contains the value 7.0 .

Math.floor()

This method rounds a floating-point value down to the closest integer value. The rounded value is displayed as a double.

Code Snippet 3 demonstrates the `Math.floor()` method.

Code Snippet 3:

```
public class BasicMathFunctons {
    public static void main(String args[]) {
        double objFloor = Math.floor(6.454);
        // will result in objFloor = 6.0
        System.out.println("Result as " + objFloor);
    }
} // displays result
```

After executing this code, `objFloor` contains the value 6.0.

Table 15.1 lists a few more basic Math methods.

Method Name	Description	Example
<code>Math.floorDiv()</code>	This method is similar to <code>floor</code> method, but is combined with division. <code>Math.floorDiv()</code> divides an integer or long value by another one and rounds the resultant value to the closest integer value.	<code>double output = Math.floorDiv(100, 9);</code> Output: 11.0
<code>Math.min()</code>	This method produces the smallest of the given values passed as inputs.	<code>int newMin = Math.min(5, 12);</code> Here, the <code>newMin</code> variable produces output as 5.
<code>Math.max()</code>	This method is similar to the <code>Math.min()</code> method with one difference that it produces the biggest value of the given inputs.	<code>int newMax = Math.max(5, 12);</code> Here, the <code>newMax</code> variable produces output as 12.
<code>Math.round()</code>	This method rounds a <code>float</code> or <code>double</code> to the closest integer. <code>Math.round()</code> method applies common mathematical rules for rounding the values.	<code>double newDecrease = Math.round(44.324);</code> <code>double newIncrease = Math.round(44.654);</code> Here, <code>newDecrease</code> variable produces output as 44.0 and the <code>newIncrease</code> variable produces 45 as output.

Method Name	Description	Example
Math.random()	This method returns a random floating point value between 0 and 1 by default. However, Math.random can be applied to get a random number between 0 and n (any number within 100).	<pre>double newRand1 = Math.random(); double newRand2 = Math.random() * 50D;</pre> <p>Output:</p> <pre>newRand1: 0.7463562032119188 newRand2: 26.360465065790073</pre>

Table 15.1: Basic Math Methods

15.2.2 Exponential and Logarithmic Math Methods

Math class also contains a set of methods to perform exponential and logarithmic operations/calculations. These methods are explained as follows:

Math.exp()

This method produces e (Euler's number) increased to the power of the value given as a parameter.

Code Snippet 4 shows Math.exp() in use.

Code Snippet 4:

```
public class ExpandoLogMathFunctons {
    public static void main(String args[]) {
        double newExpA=Math.exp(4);
        System.out.println("OutputA=" + newExpA);
        double newExpB=Math.exp(5);
        System.out.println("OutputB=" + newExpB);
    }
} //displays result
```

Here, newExpA and newExpB variables produce following output:

OutputA = 54.598150033144236

OutputB = 148.4131591025766

Math.log()

This method provides the logarithm of the given value. Math.log() method functioning under the basis of logarithm is e (Euler's number). This method performs the reverse function of Math.exp(). Code Snippet 5 demonstrates the Math.log() method.

Code Snippet 5:

```
public class ExpoandLogMathFunctons {
    public static void main(String args[]) {
        double newLogA = Math.log(2);
        System.out.println("OutputA = " + newLogA);
        double newLogB = Math.log(100);
        System.out.println("OutputB = " + newLogB);
    }
} //displays result
```

Here, the `newLogA` and `newLogB` are the variables that produce following result:

`OutputA = 0.6931471805599453`

`OutputB = 4.605170185988092`

Math.log10()

This method is similar to the `Math.log()` method with one difference that it takes 10 as a base for calculating the logarithm instead of e (Euler's Number). Code Snippet 6 demonstrates `Math.log10()` method.

Code Snippet 6:

```
public class ExpoandLogMathFunctons {
    public static void main(String args[]) {
        double newLog10A = Math.log10(6);
        System.out.println("OutputA = " + newLog10A);
        double newLog10B = Math.log10(200);
        System.out.println("OutputB = " + newLog10B);
    }
} //displays result
```

Here, `newLog10A` and `newLog10B` are the variables that produce following result:

`OutputA = 0.7781512503836436`

`OutputB = 2.3010299956639813`

Math.pow()

`Math.pow()` method takes two parameters and produces the value of the first parameter raised to the power of the second parameter. Code Snippet 7 demonstrates the `Math.pow()` method.

Code Snippet 7:

```
public class ExpoandLogMathFunctions {
    public static void main(String args[]) {
        double newPowerA = Math.pow(2, 4);
        System.out.println("OutputA as = " + newPowerA);
        double newPowerB = Math.pow(2, 5);
        System.out.println("OutputB as = " + newPowerB);
    }
}
```

Here, newPowerA and newPowerB produce following results:

OutputA as = 16.0

OutputB as = 32.0

Math.sqrt()

This method performs the square root operation for the given parameter. Code Snippet 8 demonstrates Math.sqrt() method.

Code Snippet 8:

```
public class ExpoandLogMathFunctions {
    public static void main(String args[]) {
        double newSrootA = Math.sqrt(8);
        System.out.println("OutputA = " + newSrootA);
        double newSrootB = Math.sqrt(25);
        System.out.println("OutputB = " + newSrootB);
    }
}
```

Here, the newSrootA and newSrootB produce following results:

OutputA = 2.8284271247461903

OutputB = 5.0

15.2.3 Trigonometric Math Methods

The Math class also includes a set of trigonometric methods that can calculate values used in trigonometry, such as sine, cosine, tan, and so on. Math.PI is a constant double that contains a value closest to PI. Math.PI is frequently used in trigonometric operations/calculations.

Frequently used trigonometric methods are listed in table 15.2.

Method	Description	Example
Math.sin()	Performs the sine operation. It calculates the sine value of the given angle value in radians.	<pre>double newSin = Math.sin(Math.PI); System.out.println("The value of sin = " + newSin);</pre> <p>Output:</p> <pre>The value of sin = 1.2246467991473532E-16</pre>
Math.cos()	Performs the cos operation. It calculates the cosine value of the given angle value in radians.	<pre>double newTan = Math.tan(Math.PI); System.out.println("The value of tan = " + newTan);</pre> <p>Output</p> <pre>The value of tan = 1.2246467991473532E-16</pre>
Math.asin()	Performs the arc sine value calculation of a value between 1 and -1.	<pre>double newAsin = Math.asin(Math.PI); System.out.println("The value of Asin = " + newAsin);</pre> <p>Output</p> <pre>The value of Asin = NaN</pre>
Math.acos()	Performs the arc cos value calculation of a value between 1 and -1.	<pre>double newAcos = Math.acos(1.0); System.out.println("The value of acos = " + newAcos);</pre> <p>Output</p> <pre>The value of acos = 0.0</pre>
Math.atan()	Performs the arc tangent value calculation of a value between 1 and -1.	<pre>double newAtan = Math.atan(1.0); System.out.println("The value of Atan = " + newAtan);</pre> <p>Output</p> <pre>The value of Atan = 0.7853981633974483</pre>

Method	Description	Example
Math.sinh()	Performs the hyperbolic sine value calculation of a given value between 1 and -1.	<pre>double newSinh = Math.sinh(1.0); System.out.println("The value of sinh = " + newSinh);</pre> <p>Output</p> <pre>The value of sinh = 1.1752011936438014</pre>
Math.cosh()	Performs the hyperbolic cosine value calculation of a given value between 1 and -1.	<pre>double newCosh = Math.cosh(1.0); System.out.println("The value of Cosh = " + newCosh);</pre> <p>Output</p> <pre>The value of Cosh = 1.543080634815244</pre>
Math.tanh()	Performs the hyperbolic tangent value calculation of a given value between 1 and -1.	<pre>double newTanh = Math.tanh(1.0); System.out.println("The value of tanh = " + newTanh);</pre> <p>Output</p> <pre>The value of tanh = 0.7615941559557649</pre>
Math.toDegrees()	Performs the convert operation of an angle in radians to degrees.	<pre>double newDegrees = Math.toDegrees(Math.PI); System.out.println("Output = " + newDegrees);</pre> <p>Output = 180.0</p>
Math.toRadians()	Performs an reverse operation of Math.toDegrees() method, it performs the convert operation of an angle in degrees to radians.	<pre>double newRadians = Math.toRadians(180); System.out.println("Output = " + newRadians);</pre> <p>Output = 3.141592653589793</p>

Table 15.2: Trigonometric Methods in Math Class

Exact Numeric Operations

One of the new enhancements in Java 8 is that `Math` class now includes a set of methods which raise a `ArithmaticException` when the integer (or long) value exceeds the maximum limit in the result. For example, multiplication between 100000 and 100000 exceeds the maximum limit and will result in this exception being shown. Such operations are called exact numeric operations.

Exact numeric operations can be performed with following methods:

- addExact
- subtractExact
- multiplyExact
- incrementExact
- decrementExact
- negateExact
- toIntExact

Code Snippet 9 shows the usage of `addExact()` method.

Code Snippet 9:

```
public class ExactMethodDemo {
    public static void main(String args[]) {
        int ex1 = 900000000; //
        int ex2 = 1250000000; //
        System.out.println(Math.addExact(ex1, ex2));
    }
} //displays result
```

Ideally, if `ex1` and `ex2` were just added using the `+` sign, the program would show no errors, but produce an inaccurate result as it exceeds the maximum limit for integers. Hence, here, using `addExact()` is recommended so that it would throw an exception and alert the user instead of displaying an incorrect output.

Code Snippet 9 when executed generates following:

```
Exception in thread "main" java.lang.ArithmaticException: integer overflow
at java.lang.Math.addExact(Math.java:790)
```

Similarly, other exact numeric methods can be used.

Next Numeric Operations

Math class also introduces a set of methods to perform numeric operations that are applied where you must display the closest value of a given number. For example, to display a value higher than a given value, you can apply `nextUp()` method.

Following are the next methods:

- `nextUp`
- `nextDown`
- `nextAfter`

Code Snippet 10 demonstrates the `nextDown()` method.

Code Snippet 10:

```
public class ExactMethodDemo {
    public static void main(String args[]) {
        int nextA = 1000; //
        System.out.println(Math.nextDown(nextA));
    }
} // displays result
```

Output:

999.99994

15.3 Other Ways of Creating Objects

Let us look at other ways of creating objects besides the `new` keyword which make use of constructors and default constructors.

→ **Using `newInstance`**

If you know the name of the class and it has a public default constructor, we can create an object using `Class.forName()` and `newInstance()` methods. This method, `Class.forName`, actually loads the class in Java, but does not create any object. To create an object of the class, you must use the `newInstance()` method of the class. Code Snippet 11 demonstrates an example.

Code Snippet 11:

```
public class NewInstanceDemo {
    String name = "objNewInstanceDemo";
    public static void main(String[] args) throws ClassNotFoundException,
    InstantiationException, IllegalAccessException {
        Class cls = Class.forName("NewInstanceDemo");
        NewInstanceDemo obj = (NewInstanceDemo) cls.newInstance();
        System.out.println(obj.name);
    }
}
```

In the code, we assign a string variable and call the `Class.forName()` method with the current class name. The return value of this method is assigned to `cls`, which is an object of `Class`. We then call `newInstance()` on `cls` and create an object, `obj`.

Finally, we print the name of the object. The `main` method may raise some runtime exceptions hence, these exceptions are included in the main declaration. The output of the code will be `objNewInstanceDemo`.

→ **Using clone() method:**

Calling the `clone()` method on any object causes JVM to create a new object and copy all content of the previous object into it. Creating an object using `clone` method does not invoke a constructor. To use `clone()` method on an object, you must implement `Cloneable` and define the `clone()` method in it. Code Snippet 12 demonstrates an example.

Code Snippet 12:

```
import java.util.*;
public class CloneDemo implements Cloneable {
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
    String name = "objCloneDemo";
    public static void main(String[] args) throws CloneNotSupportedException {
        CloneDemo obj1 = new CloneDemo();
        CloneDemo obj2 = (CloneDemo) obj1.clone();
        System.out.println(obj2.name);
    }
}
```

Here, in this code, we create the clone of an existing Object and not a new Object. The code may raise a `CloneNotSupportedException`, hence it is declared with the respective methods.

→ **Using newInstance() method of Constructor class**

This is similar to the `newInstance()` method of a class. There is one `newInstance()` method in the `java.lang.reflect.Constructor` class which we can use to create objects. It can also call parameterized constructor and private constructor by using this `newInstance()` method. Both `newInstance()` methods are known as reflective ways to create objects. Code Snippet 13 illustrates an example.

Code Snippet 13:

```
import java.lang.reflect.*;
import java.util.*;
public class ReflectionDemo {
    private String name;
    ReflectionDemo()
    {
    }
```

```

public void setName(String name) {
    this.name = name;
}
public static void main(String[] args) throws NoSuchMethodException,
    InstantiationException, IllegalAccessException,
    InvocationTargetException {
    Constructor<ReflectionDemo> constructor = ReflectionDemo.class.
        getDeclaredConstructor();
    ReflectionDemo objReflectionDemo = constructor.newInstance();
    objReflectionDemo.setName("objReflectionDemo");
    System.out.println(objReflectionDemo.name);
}
}

```

15.4 Java Modules

In Java 9, a new packaging mechanism was introduced that enables developers to package a Java application or Java API as a separate Java module. This feature is called as a 'module' and can specify which Java packages it contains and which of these should be visible to other Java modules. A Java module is packaged as a modular JAR file. Thus, a module can be defined as a group of closely related packages and resources along with a new module descriptor file.

Packages inside a module are identical to Java packages you have been using so far. Resources can comprise media files, configuration files (XML files), and so on.

Various benefits of Java modules are as follows:

→ Smaller Application Distributables

In earlier versions of Java, prior to version 9, developers had to package all of the Java Platform APIs with their Java application as there was no way to determine exactly what classes their Java application used. This often led to large size of applications during deployment and distribution, with a large amount of Java classes included, many of which the application may not even be using. This was a potential limitation on small devices such as mobile gadgets, Raspberry Pis, and so on.

Since Java 9, Java Platform APIs have now been split up into separate modules. Benefit of this is that you can now specify what modules of Java platform are required by your application. During deployment, Java can package your application comprising only those Java Platform modules that are actually used by your application. The outcome of this is smaller application distributables.

→ Hiding Unwanted Details

A Java module must explicitly indicate which Java packages inside it are to be exported or made visible to other modules that use this module. A Java module can also contain packages which are not exported. Packages that are not exported are also referred to as **hidden** packages, or encapsulated packages. Classes in unexported packages cannot be used by other Java modules.

Such packages can only be used internally in the Java module in which they are defined.

Thus, use of modules can help to hide unwanted and internal details safely and promote better security.

→ **Lesser coupling between components**

Modules make it easy to support less coupling between components.

→ **Faster Detection of Missing Modules**

From Java 9 onwards, as soon as Java VM starts up, it will be able to check the module dependency graph from the application module and further on. In case any required modules are not found at startup, Java VM raises an error about the missing modules and shuts down. This was not possible in versions of Java prior to 9, where missing classes would not be detected until the application actually tried to use the missing class. Detecting missing classes at application startup time is far better than detecting them at runtime during application processing.

15.4.1 Types of Modules

There are four types of modules in the new module system from Java 9 onwards:

- **System Modules:** These are Java SE and JDK system-defined modules. You can view a list of these by running the `list-modules` command. Figure 15.1 shows some of these modules.

```
C:\>java --list-modules
java.base@15.0.2
java.compiler@15.0.2
java.datatransfer@15.0.2
java.desktop@15.0.2
java.instrument@15.0.2
java.logging@15.0.2
java.management@15.0.2
java.management.rmi@15.0.2
java.naming@15.0.2
java.net.http@15.0.2
```

Figure 15.1: Displaying System Modules

- **Application Modules:** These modules are what developers want to build when they decide to use modules. They are named and defined in the compiled `module-info.class` file included in the assembled JAR.
- **Automatic Modules:** You can include unofficial modules automatically by adding existing JAR files to the module path. The name of the module will be derived from the name of the JAR. Automatic modules will have full read access to every other module loaded by the path.

- **Unnamed Module:** When a class or JAR is loaded onto the classpath, but not the module path, it is automatically added to the unnamed module.
- Its purpose is to maintain backward compatibility with previously-written Java code.

15.4.2 Creating a Java Module

Steps to create a module in Java (version 9 onwards) are as follows:

1. **Create a module descriptor file**

To set up a module, you are required to put a special file known as the module descriptor at the root of your packages named `module-info.java`. This file comprises all the data required to build and use your new module. You place it in a folder that has the same name as your module name. Hence, if your module name is `com.test.testmodule`, your module, that is, your file `module-info.java` should be placed in the path `src/com.test.testmodule`.

2. **Create the module declaration**

You can construct the module with a declaration whose body is either empty or made up of module directives. Code Snippet 14 shows an example.

Code Snippet 14:

```
module com.test.testmodule {  
}
```

3. **Add directives**

Different module directives accomplish different purposes. Some examples of directives are described here.

The `requires` module directive allows us to declare module dependencies. In simple terms, the `requires` keyword indicates that this module depends on another module.

Syntax:

```
module my.module {  
    requires module.name;  
}
```

Code Snippet 15 states that our module is dependent on `java.base` module.

Code Snippet 15:

```
module com.test.testmodule {  
    requires java.base;  
}
```

The `requires` directive also has further options:

- By using `requires static` directive, you can create a compile-time-only dependency.
- The `requires transitive` directive forces any downstream consumers to read the

required dependencies.

You can use the `exports` directive to expose all public members of the named package as shown in Code Snippet 16.

Code Snippet 16:

```
module com.test.testmodule {
    exports com.test.testmodule.testpackage;
}
```

By default, all packages are module private. The `exports` keyword indicates that these packages are available to other modules. This means that public classes are default only public within the module unless it is specified within the module info declaration.

Yet another directive is the `uses` directive which indicates that the module uses a service.

4. Set Up the Project

Set up the project structure by creating several directories to organize the files. For example, following may be the structure for `testmodule`:

```
module-demo
src
|_ com
  |_ test
    |_ testmodule - Main.java
```

5. Add Code to the Module

Add the code shown in Code Snippet 17 to a Java file and save it as `Main.java` in the path shown in Step 4.

Code Snippet 17:

```
package com.test.testmodule;
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World from Module! :)");
    }
}
```

6. Compile the Module

Type following command at the command prompt:

```
javac -d module-demo/com.test.testmodule src/com.test.testmodule/module-
info.java src/com.test.testmodule/com/test/testmodule/Main.java
```

This will result in a `Main.class` being produced in the path `module-demo\com.test.testmodule\com\test\testmodule`.

7. Execute the module

To run the application, type following command at the command prompt:

```
java --module-path module-demo --module com.test.testmodule/com.test.testmodule.Main
```

Figure 15.2 shows the commands at the command prompt and the resulting output.

```
C:\>javac -d module-demo/com.test.testmodule src/com.test.testmodule/module-info.java src/com.test.testmodule/com/test/testmodule/Main.java

C:\>java --module-path module-demo --module com.test.testmodule/com.test.testmodule.Main
This example demonstrates a Module!

C:\>
```

Figure 15.2: Output

15.5 Enhanced Switch Statement and Switch Expressions

The traditional switch statement in earlier versions of Java has been enhanced in recent versions to include a new arrow syntax (also called 'switch labeled rules') as shown in Code Snippet 18.

Code Snippet 18:

```
public class EnhancedSwitchStatementDemo{
    public static void main(String[] args) {
        System.out.println("Enhanced Switch Statement:");
        final int integer = 3;
        String numericString;
        switch (integer) {
            case 1 -> numericString = "one";
            case 2 -> numericString = "two";
            case 3 -> numericString = "three";
            default -> numericString = "N/A";
        }
        System.out.println("\t" + integer + " ==> " + numericString);
    }
}
```

The 'arrow' syntax ('label rules') accomplishes the switching without explicit specification of break. This is not only less code, but importantly, it has the advantage of not allowing for switch 'fall-through' which used to happen in earlier switch statements.

Output:

Enhanced Switch Statement:

```
3 ==> three
```

From Java 13 and 14 onwards, you can use switch expressions. A switch expression is a poly expression, which means that if the target type is known, this type is pushed down into each block. The type of a switch expression is its target type, in case it is known; if not known, a standalone type is determined block of the 'case L->' switch label. Code Snippet 19 shows an example.

Code Snippet 19:

```
public class SwitchExprDemo{
    public static void main(String[] args) {
        final int integer = 1;
        System.out.println("Switch Expression with Colons/Breaks:");
        final String numericString = switch (integer)
        {
            case 1 -> "Grade A";
            case 2 -> "Grade B";
            case 3 -> "Grade C";
            case 4 -> "Grade D";
            default -> "N/A";
        };
        System.out.println("\t" + integer + " ==> " + numericString);
    }
}
```

Output:

Switch Expression with Colons/Breaks:

```
1 ==> Grade A
```

As switch can now be used as an expression, you do not require to include break; for each block, nor do you have to worry about finding the missing break; in long strings of blocks while debugging. This is because whenever the case label is matched, it triggers only the applicable line of code, not a fall through of the given block only halted by the break; statements.

15.6 JShell

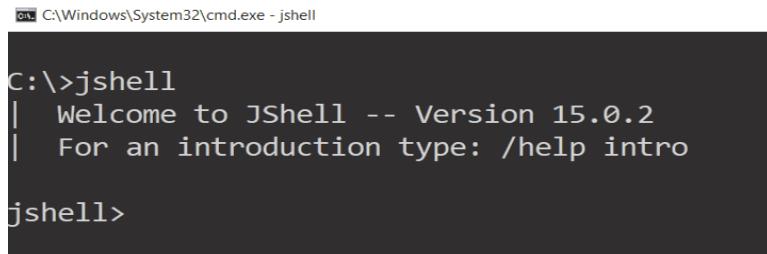
Java Shell tool (JShell) is an interactive tool that helps you try out Java code and easily explore options as you develop your program. You can test individual statements, play around with unfamiliar APIs within the JShell session, and try out different overloads of a method. It is important to remember that JShell does not replace an IDE, instead you can paste code into JShell to try it out, and then paste working code

from JShell into your program editor or IDE.

Start and Stop JShell

JShell is included in JDK from version 9 onwards. To start JShell, type `jshell` on the command line.

Figure 15.3 shows the command and the response from JShell. Text that you enter is shown in bold:



```
C:\>jshell
| Welcome to JShell -- Version 15.0.2
| For an introduction type: /help intro

jshell>
```

Figure 15.3: Launching JShell

To start JShell in verbose mode, use the `-v` option:

```
jshell -v
```

To exit JShell, enter `/exit`

Key Features of JShell

→ Re-declaration of variables

In Java, it is not possible to re-declare a variable. With the help of JShell, however, you can always re-declare the variable based on requirements. Note that this is applicable for both primitive as well as for reference type variables. In fact, users can re-declare any construct as many times as they want.

Example:

```
jshell> String msg="Morning"
msg ==> "Morning"
jshell> Integer msg=90
msg ==> 90
```

→ Scratch variables

Any expression evaluation from JShell command line is assigned to some variables if not explicitly assigned by user. Such variables are called scratch variables.

Example:

```
jshell> "Northern"+ "Lights"
$1 ==> "Northern Lights"
```

→ Exception Handling in JShell

In following example, we are not catching any exceptions thrown by `divide()` method. JShell takes care of it. We are not importing `IOException` class either, yet the code is compiled and executed. This is because for any JShell session, some packages are imported by default.

Example:

```
jshell> int divide(int num1,int num2) throws IOException{
...> if(num2==0) {
...>     throw new IOException();
...> }
...> return num1/num2;
...> }
| created method divide(int,int)

jshell> divide(34,0)
| java.io.IOException thrown:
| at divide (#2:3)
| at (#3:1)
```

→ Tab completion in JShell

JShell allows developers to complete their code constructs automatically using the Tab key.

15.7 Hidden Classes and Sealed Classes

Hidden classes are classes that cannot be used directly by bytecode of other classes. Hidden classes are mainly intended for use by frameworks that generate classes at runtime and use them indirectly, via reflection. Hidden classes is a feature that will be useful primarily for framework developers and not general application developers.

The `sealed` scope modifier in Java 15 provides fine grained inheritance control for classes and interfaces. This modifier when applied to a class or interface restricts other classes or interfaces may extend it. Sealed classes are also useful for creating secure hierarchies by decoupling accessibility from extensibility.

As an example, imagine a business domain that only works with cars and trucks, not motorcycles.

When creating the `Automobile` abstract class, you should be able to allow only `Car` and `Truck` classes to extend it. In that way, you can ensure that there will be no misuse of the `Automobile` abstract class within your domain.

The sealed feature introduces a couple of new modifiers and clauses in Java: `sealed`, `non-sealed`, and `permits`. The `permits` clause then specifies the classes that are permitted to implement the sealed interface or class. This clause should be defined after any `extends` or `implements` clauses.

Code Snippet 20 shows an example.

Code Snippet 20:

```
public abstract sealed class Automobile permits Car, Truck {
    protected final String regID;
    public Automobile(String regID) {
        this.regID = regID;
    }
    public String getregID() {
        return regID;
    }
}
```

A permitted subclass must define a modifier. It may be declared final to prevent any further extensions.

Code Snippet 21 shows an example.

Code Snippet 21:

```
public final class Truck extends Automobile implements Service {
    private final int loadCapacity;
    public Truck(int loadCapacity, String regID) {
        super(regID);
        this.loadCapacity = loadCapacity;
    }
    public int getLoadCapacity() {
        return loadCapacity;
    }
    @Override
    public int getMaxServiceIntervalInMonths() {
        return 24;
    }
}
```

A permitted subclass may also be declared sealed. If on the other hand, you declare it as non-sealed, then it will be open for extension.

15.8 Check Your Progress

1. Which of these are valid directives that can be used in Java modules?

(A)	exports	(C)	requires
(B)	uses	(D)	implements

2. If you know the name of the class and it has a _____, we can create an object using `Class.forName()` and `newInstance()` methods.

(A)	public parameterized constructor	(C)	public default constructor
(B)	public static method	(D)	private method

3. Which among the following methods is applied to display a value higher than given value?

(A)	multiplyExact	(C)	nextUp
(B)	addExact	(D)	nextDown

4. What is the name of the module descriptor file to be used when defining modules?

(A)	module-info.xml	(C)	module.xml
(B)	moduledescriptor.java	(D)	module-info.java

5. _____ are mainly intended for use by frameworks that generate classes at runtime and use them indirectly, via reflection.

(A)	Private classes	(C)	Sealed classes
(B)	Hidden classes	(D)	Modules

15.8.1 Answers

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

Summary

- Between Java 9 to 15, several features that existed in earlier versions have been deprecated or removed now and no longer recommended for use.
- Nashorn is a JavaScript engine that was used in Java versions 8 to 10 to compile JavaScript into Java bytecode. It was deprecated in Java 11.
- Advanced mathematical operations can be performed with new methods in Math class.
- A module can be defined as a group of closely related packages and resources along with a new module descriptor file.
- Recent Java versions have enhanced the switch statement and also provided a switch expression.
- JShell is an interactive tool that helps you try out Java code and easily explore options as you develop your program.
- In Java 15 onwards, you can define sealed classes and hidden classes.

Try it Yourself

1. NewAge Fintech is a financial startup and has been performing data analysis for various clients spanning worldwide. The startup wants to develop software with advanced data modeling tool for its customers.

As a software engineer, using the new features of Java, you must perform following tasks:

- i. Create a class DemoMath.java and include methods for calculating the exponential, logarithmic, floor, ceil, and absolute value for a given integer.
- ii. Create a class DemoTrigMath.java and include methods for calculating the arcsine, arccosine, and arctangent value of a given integer.
- iii. In the DemoTrigMath.java class, include the method that returns the equivalent value in degrees for the value specified in radians as method argument and vice versa.

Use JShell to try out the code before adding it into NetBeans IDE.

Onlinevarsity

**LEARN VIRTUAL
LEARN ANYTHING**



Session - 16

Swing and JavaFX

Welcome to the Session, **Swing and JavaFX**.

This session explains extensible GUI components in Java. The session begins with explaining Swing and then, describes layout managers, and introduces JavaFX. It covers various aspects of JavaFX in detail.

In this Session, you will learn to:

- Explain Swing Components
- Describe Layout Managers
- Describe JavaFX in detail



16.1 Introduction to Swing

Java Swing is a part of Java Foundation Classes (JFC) used for creating desktop applications with GUI features. It is built on the top of Abstract Windowing Toolkit (AWT) API that was the first windowing and graphics toolkit created for Java. With time, a more sophisticated form of the toolkit called Swing was developed.

Swing has been included as part of the Java SE since release 1.2. It is completely written in Java. It is platform-independent, lightweight, and follows MVC architecture. The `javax.swing` package provides classes for Swing API such as `JButton`, `JTextField`, `JTextArea`, `JRadioButton`, `JCheckbox`, `JMenu`, `JColorChooser`, and so on.

Swing Features

Few key features of Swing include:

- **Lightweight**

Swing components are independent of native Operating System's API and its controls are mostly rendered using pure Java code.

- **Rich Controls**

It provides a rich set of advanced controls such as `Tree`, `TabbedPane`, `Slider`, `ColorPicker`, and table controls.

- **Highly Customizable**

Its controls are customized in an easy manner, as visual appearance is independent of internal representation.

- **Pluggable look-and-feel**

Its application look and feel can be changed at run-time, based on available instance values.

16.1.1 Swing Controls

Swing controls can be categorized as follows:

- **UI Elements**

These are core visual components with which user interacts and undergoes an experience. AWT provides widely used and common elements, from basic to complex.

- **Layouts**

They take care of how UI elements should be organized on the screen and provide a required look and feel.

- **Behavior**

They depend on events that occur when user interacts with UI elements.

16.1.2 Hierarchy of Swing Classes

Figure 16.1 depicts hierarchy of Swing classes.

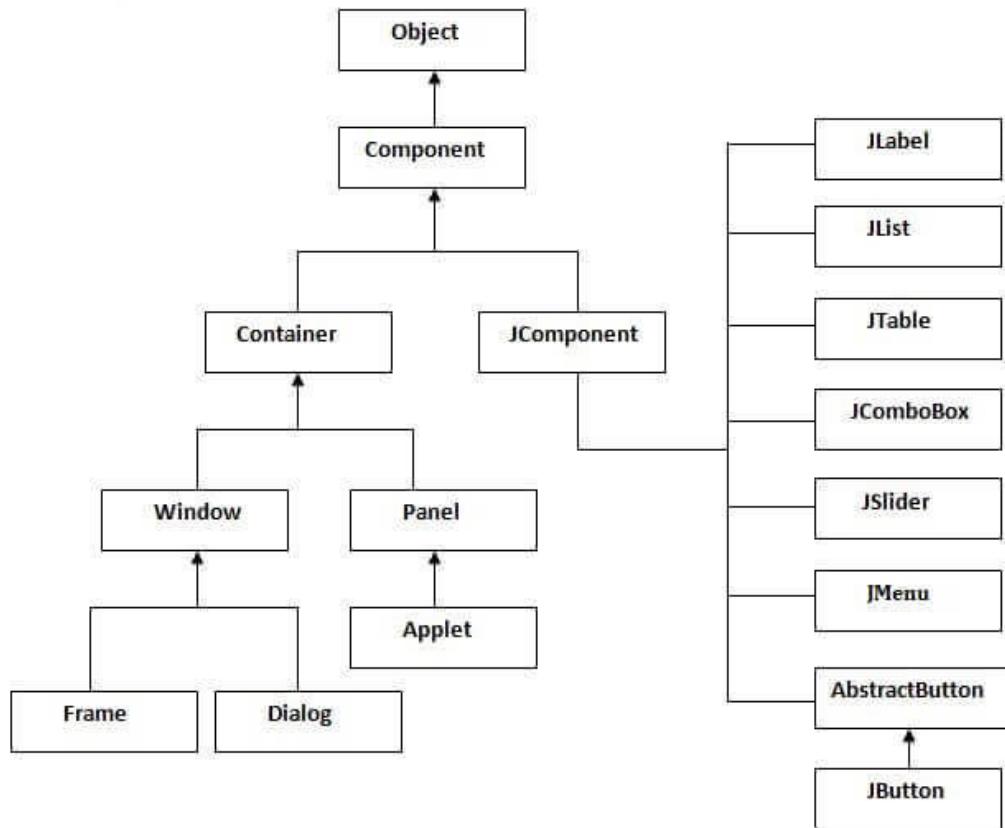


Figure 16.1: Swing Hierarchy

Each Swing control inherits properties from their respective component classes as shown in table 16.1.

Class	Description
Component	It is an abstract base class for the non-menu user-interface controls of Swing. It can also represent an object with graphical representation.
Container	It is the component that can contain other Swing components.
JComponent	JComponent is a base class for all Swing UI components. In order to use a Swing component that inherits from JComponent, it must be a part of containment hierarchy and its root should be top-level Swing container.

Table 16.1: Swing Classes and Description

16.1.3 Swing UI Elements

You can create Swing projects in Apache NetBeans 12 by selecting **Java with Ant → JavaFX** under **Categories** and then, selecting **JavaFX in Swing Application** in the **Projects** pane on the right in the **New Project** dialog box. Figure 16.2 depicts this.

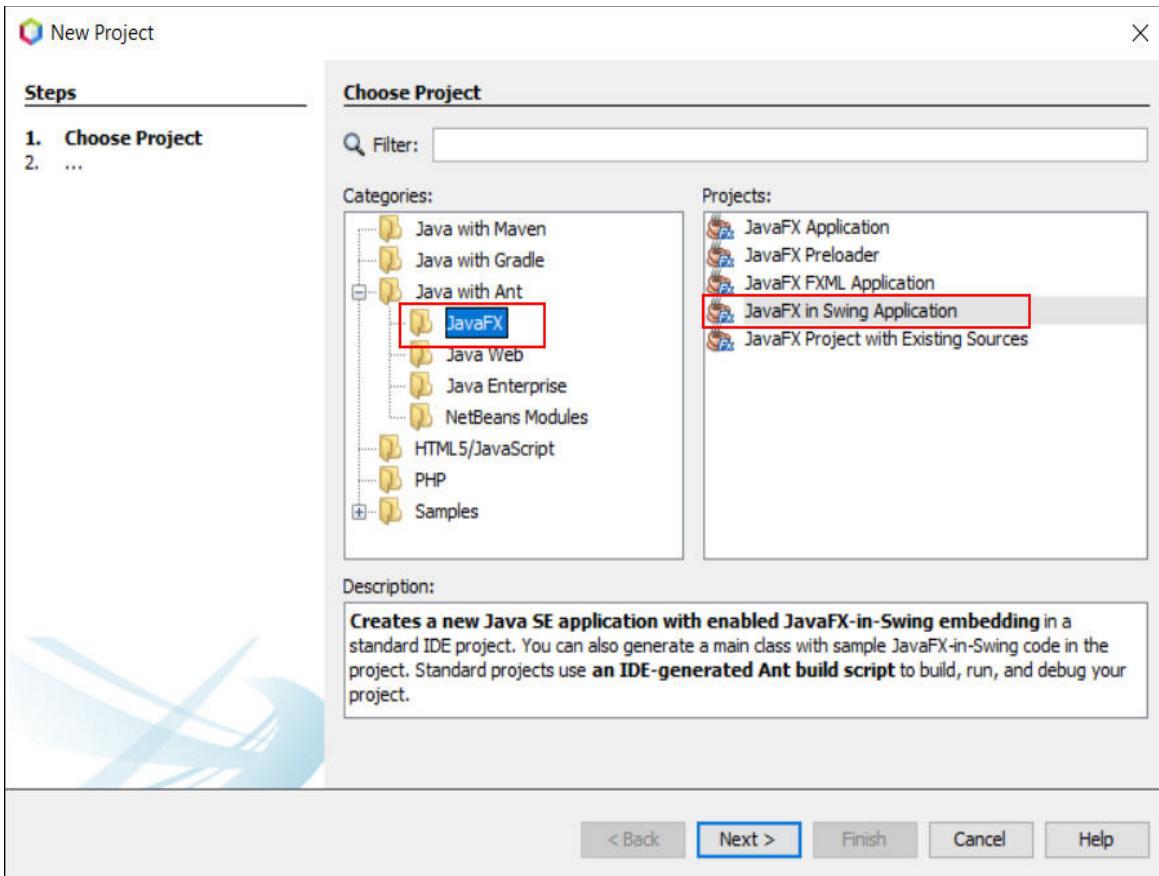


Figure 16.2: Creating Swing Projects

Following UI components are commonly used controls for designing GUI using Swing:

→ **JLabel:**

`JLabel` can display text, image, or both. Label contents can be aligned by setting the vertical and horizontal alignment in its display area. By default, labels are aligned vertically centered in their display area. Text-only labels are leading edge aligned, whereas image-only labels are horizontally center aligned. Code Snippet 1 displays `JLabel` example with aligns options and colors for `JLabel` text. It inherits `JComponent` class, which implements `SwingConstants` and `Accessible` interfaces.

Code Snippet 1:

```
package com.aptech.gui;
import javax.swing.*;
class JLabelDemo {
    public static void main(String args[]) {
        JFrame frame= new JFrame("Swing JLabel Example");
        JLabel label1,label2;
        label1=new JLabel("JLabel Example",JLabel.CENTER);
        label1.setBounds(50,50, 250,30);
        label2=new JLabel("Welcome to Aptech JLabel Example",JLabel.CENTER);
```

```

label2.setBounds(50,100, 250,30);
frame.add(label1);
frame.add(label2);
frame.setSize(400,400);
frame.setLayout(null);
frame.setVisible(true);
}
}

```

The code in Code Snippet 1 creates a `JFrame` with name **Swing JLabel Example**. Then, it displays two labels, namely `label1` and `label2`, which are created using `JLabel` component. `JLabel` accepts two arguments: label text and alignment position. The `setBounds()` method is useful to set x and y coordinate positions from top-left along with width and height for components.

Output of this Code Snippet is displayed in figure 16.3.

```

1 package com.aptech.gui;
2 import javax.swing.*;
3 class JLabelDemo
4 {
5     public static void main(String args[])
6     {
7         JFrame frame= new JFrame("Swing JLabel Example");
8         JLabel label1,label2;
9         label1=new JLabel("JLabel Example",JLabel.CENTER);
10        label1.setBounds(50,50, 250,30);
11        label2=new JLabel("Welcome to Aptech JLabel Example",JLabel.CENTER);
12        label2.setBounds(50,100, 250,30);
13        frame.add(label1);
14        frame.add(label2);
15        frame.setSize(400,400);
16        frame.setLayout(null);
17        frame.setVisible(true);
18    }
19 }

```

Figure 16.3: JLabel Example

→ JButton:

`JButton` is an implementation of a push button. It has a label and generates an event when user clicks it. It can also have an image and inherits `AbstractButton` class, which implements the `Accessible` interface.

Code Snippet 2 explains how to create `JButton` with caption `Click Here`.

Code Snippet 2:

```

package com.aptech.gui;
import javax.swing.*;
public class JButtonDemo {
    public static void main(String[] args) {
        JFrame frame=new JFrame("JButton Example");
        JButton button=new JButton("Click Here");
        button.setBounds(100,100,150,30);
        frame.add(button);
        frame.setSize(400,400);
        frame.setLayout(null);
        frame.setVisible(true);
    }
}

```

The output is shown in figure 16.4.

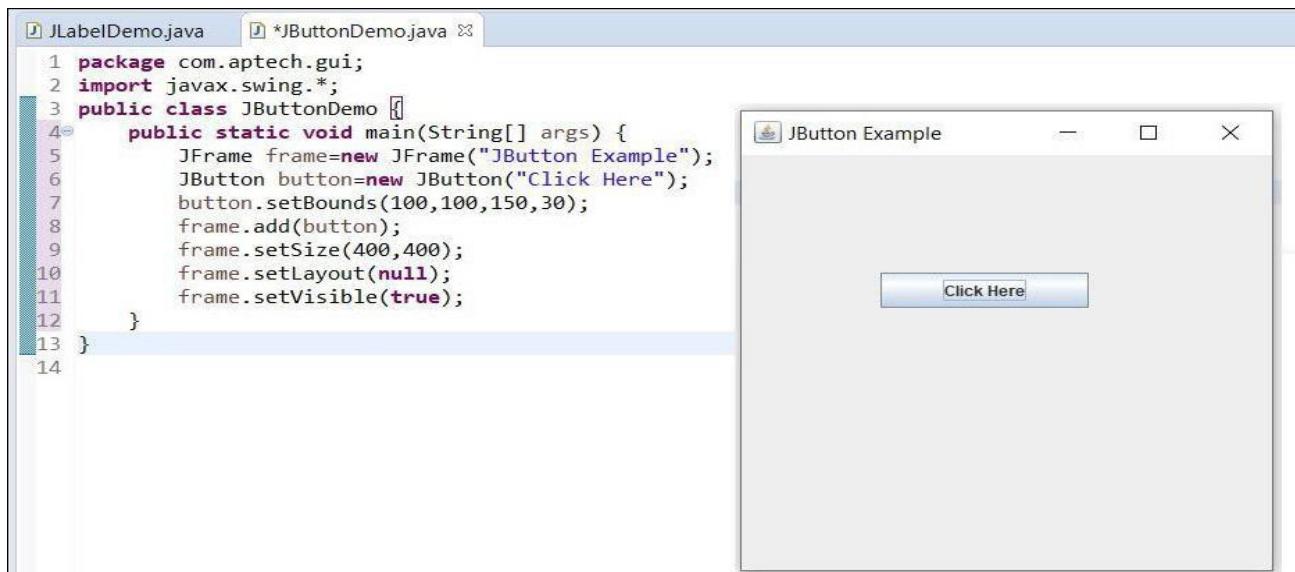


Figure 16.4: JButton Example

If the button is clicked, nothing happens. This is because the code for handling the button click is not yet added as of now. When a user clicks a button, an ‘action’ is performed. To listen to such actions, ActionListeners are used. This can be observed in Code Snippet 3. Using ActionListener, the system understands that a particular action has been performed by user. Then, accordingly, it carries out further processing such as setting some notification, performing some task, and so on.

Code Snippet 3:

```

package com.aptech.gui;
import java.awt.event.*;
import javax.swing.*;
public class JButtonALDemo {
    public static void main(String[] args) {
        JFrame frame=new JFrame("JButton ActionListener Example");
        final JTextFieldtextfield=new JTextField();textfield.setBounds(100,100,250,20);
        JButton button=new JButton("Click ME");
        button.setBounds(150,150,95,30);
        button.addActionListener(new ActionListener(){
            textfield.setText("Welcome to ActionListener Example.");
        });
        frame.add(button);
        frame.add(textfield);
        frame.setSize(400,400);
        frame.setLayout(null);
        public void actionPerformed(ActionEvent e){
            frame.setVisible(true);
        }
    }
}

```

Figures 16.5 and 16.6 display the output of Code Snippet 3.

```

package com.aptech.gui;
import java.awt.event.*;
import javax.swing.*;
public class JButtonALDemo {
    public static void main(String[] args)
    {
        JFrame frame=new JFrame("JButton ActionListener Example");
        final JTextField textfield=new JTextField();
        textfield.setBounds(100,100, 250,20);
        JButton button=new JButton("Click ME");
        button.setBounds(150,150,95,30);
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e){
                textfield.setText("Welcome to ActionListener Example.");
            }
        });
        frame.add(button);
        frame.add(textfield);
        frame.setSize(400,400);
        frame.setLayout(null);
        frame.setVisible(true);
    }
}

```

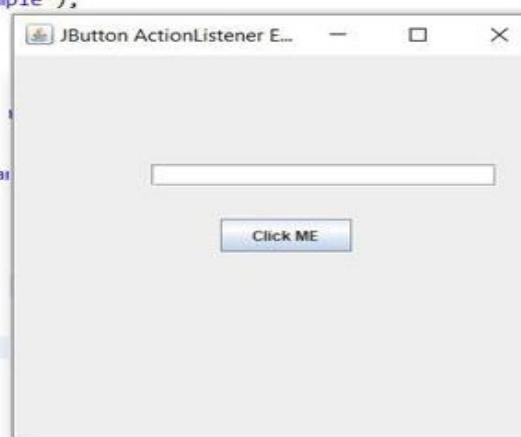


Figure 16.5: Output – Before Clicking the Button

```

package com.aptech.gui;

import java.awt.event.*;
import javax.swing.*;
public class JButtonALDemo {
    public static void main(String[] args)
    {
        JFrame frame=new JFrame("JButton ActionListener Example");
        final JTextField textfield=new JTextField();
        textfield.setBounds(100,100, 250,20);
        JButton button=new JButton("Click ME");
        button.setBounds(150,150,95,30);
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e){
                textfield.setText("Welcome to ActionListener Ex.");
            }
        });
        frame.add(button);
        frame.add(textfield);
        frame.setSize(400,400);
        frame.setLayout(null);
        frame.setVisible(true);
    }
}

```

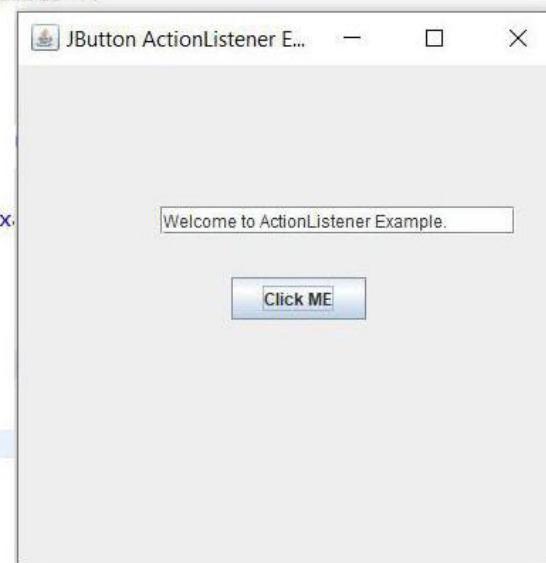


Figure 16.6: Output- After Button is Clicked

→ JColorChooser:

It provides a pane of controls designed to allow a user to manipulate and select a color. `JColorChooser()` creates a color chooser pane with an initial color of white. Whereas, `JColorChooser(color initialcolor)` is used to create a color chooser panel with the specified color initially as shown in Code Snippet 4.

Code Snippet 4:

```

package com.aptech.gui;
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
public class JColorChooserDemo extends JFrame implements ActionListener {
    JButton button;
    Container container;
    JColorChooserDemo() {
        container=getContentPane();
        container.setLayout(new FlowLayout());
        button=new JButton("color");
        button.addActionListener(this);
        container.add(button);
    }
}

```

```

}
public void actionPerformed(ActionEvent e) {
    Color initialcolor=Color.RED;
    Color color=JColorChooser.showDialog(this,"Select a
        color",initialcolor);
    container.setBackground(color);
}
public static void main(String[] args) {
    JColorChooserDemo colorchooser=new JColorChooserDemo();
    colorchooser.setSize(400,400);
    colorchooser.setVisible(true);
    colorchooser.setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
}

```

Code Snippet 4 enables user to create a container with a button and then, based on click action performed by user, it allows selecting a color with the help of `JColorChooser`.

Figure 16.7 displays the output for Code Snippet 4, after the click action is performed.

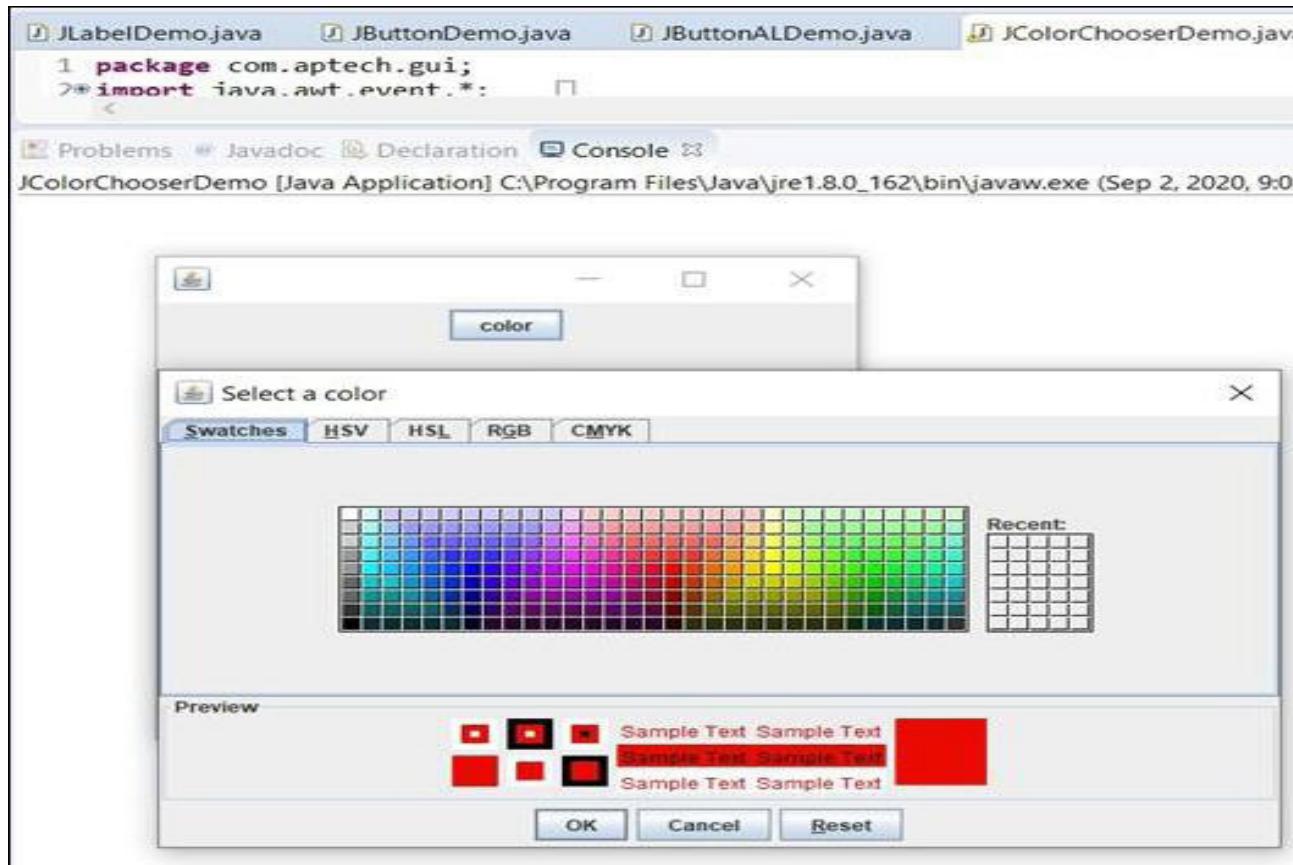


Figure 16.7: Color Chooser Application

Similarly, there are many other Swing UI classes that help to create GUI for desktop applications. Some of

these are listed in table 16.2 with their description.

Classes	Description
JCheckBox	It is a graphical component that can be in either an on (true) or off (false) state.
JRadioButton	It is a graphical component that can be in either an on (true) or off (false) state. in a group.
JList	JList component presents the user with a scrolling list of text items.
JComboBox	JComboBox component presents the user with a drop-down to show a menu of choices.
JTextField	JTextField object is a text component that allows editing of a single line of text.
JPasswordField	A JPasswordField object is a text component specialized for password entry.
JTextArea	A JTextArea object is a text component that allows editing of multiple lines of text.
ImageIcon	An ImageIcon control is an implementation of the Icon interface that paints icons from Images.
JScrollbar	Scrollbar control represents a scroll bar component in order to enable the user to select from range of values.
JOptionPane	JOptionPane provides set of standard dialog boxes that prompt users for a value or informs them of something.
JFileChooser	JFileChooser control represents a dialog window from which the user can select a file.
JProgressBar	JProgressBar represents a the progress bar displays the tasks percentage of completion, as the task progresses towards completion.
JSlider	JSlider lets the user graphically select a value by sliding a knob within a bounded interval.
JSpinner	JSpinner is a single line input field that lets the user select a number or an object value from an ordered sequence.

Table 16.2: Swing Classes

16.2 Layout Managers

A Layout manager helps to arrange position for all components within a container. If you do not specify a layout manager, components will be positioned by default layout manager named `FlowLayout`. Java provides various layout managers to position controls. Properties of layout managers such as size, shape, and arrangement vary from one layout manager to another.

A layout manager adapt to the dimensions of application window. It is associated with each Container object. Each layout manager is an object of the class that implements the `LayoutManager` interface.

Table 16.3 outlines Layout Managers that are commonly used while designing GUI with AWT.

Layout Manager	Description
BorderLayout	It arranges components to fit in the five regions such as east, west, north, south, and center.
CardLayout	It treats each component in container as a card and only one card is visible at a time.
FlowLayout	It is the default layout and it arranges components in a directional flow.
GridLayout	It manages the components in the form of a rectangular grid.
GridBagLayout	This is the most flexible layout manager class, it aligns the component vertically, horizontally, or along their baseline having different sizes.
GroupLayout	It hierarchically groups the components in order to position them in a container.
SpringLayout	It positions children of its associated container according to a set of constraints.

Table 16.3: Layout Manager Classes

Code Snippet 5 shows how to use BorderLayout, which arranges components in five regions namely, north, south, east, west, and center. Each region may contain one component only. It is the default layout of frame or window.

Code Snippet 5:

```
package com.aptech.gui;
import java.awt.*;
import javax.swing.*;
public class BorderLayoutDemo {
    JFrame frame;
    BorderLayoutDemo () {
        frame=new JFrame();
        JButton button1=new JButton("NORTH");
        JButton button2=new JButton("SOUTH");
        JButton button3=new JButton("EAST");
        JButton button4=new JButton("WEST");
        JButton button5=new JButton("CENTER");
        frame.add(button1, BorderLayout.NORTH);
        frame.add(button2, BorderLayout.SOUTH);
        frame.add(button3, BorderLayout.EAST);
        frame.add(button4, BorderLayout.WEST);
        frame.add(button5, BorderLayout.CENTER);
        frame.setSize(300,300);
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        new BorderLayoutDemo();
    }
}
```

```

    }
}
}
```

In Code Snippet 5, JButton is used to create five different buttons for various positions and JFrame is used to add buttons along with layout and directions as shown in figure 16.7.

```

1 package com.aptech.gui;
2 import java.awt.*;
3 import javax.swing.*;
4
5 public class BorderLayoutDemo {
6
7     JFrame frame;
8     BorderLayoutDemo(){
9         frame=new JFrame();
10
11         JButton button1=new JButton("NORTH");
12         JButton button2=new JButton("SOUTH");
13         JButton button3=new JButton("EAST");
14         JButton button4=new JButton("WEST");
15         JButton button5=new JButton("CENTER");
16
17         frame.add(button1,BorderLayout.NORTH);
18         frame.add(button2,BorderLayout.SOUTH);
19         frame.add(button3,BorderLayout.EAST);
20         frame.add(button4,BorderLayout.WEST);
21         frame.add(button5,BorderLayout.CENTER);
22
23         frame.setSize(300,300);
24         frame.setVisible(true);
25     }
26     public static void main(String[] args) {
27         new BorderLayoutDemo();
28     }
29
30 }
```



Figure 16.8: BorderLayout

In a similar way, other layout managers are used to arrange elements at various positions within window as described in table 16.3.

16.3 JavaFX

JavaFX is a Java library used to create Rich Internet Applications (RIA). Applications written using JavaFX can run consistently across multiple platforms and devices such as Desktop Computers, Mobile Phones, TVs, Tablets, and so on.

For developing GUI applications using Java programming language, programmers often relied on libraries such as Advanced Windowing Tool kit and Swing. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content. JavaFX contains a rich set of APIs that provide necessary classes that are responsible for executing a full featured JavaFX application. Table 16.4 lists some of the JavaFX APIs with their description.

Package Name	Description
javafx.animation	Provides set of classes that are responsible for transitions based animations

Package Name	Description
javafx.application	Provides application life-cycle methods
javafx.collections	Provides classes that can handle collections and related utilities
javafx.concurrent	Provides classes that are responsible for multitasking
javafx.embed.swing	Provides set of classes that can be used inside Swing code
javafx.embed.swt	Provides set of classes that can be used inside the Standard Widget Toolkit (SWT) code
javafx.event	Provides classes that deal with events and their handling
javafx.fxml	Contains set of classes that are responsible of loading hierarchy from markup
javafx.geometry	Provides 2D classes that contains methods to operate 2D geometry on the object
javafx.scene	Provides classes to deal with scene graph API
javafx.scene.canvas	Provides set of classes that deal with canvas
javafx.scene.control	Contains classes for all JavaFX components
javafx.scene.effect	Contains set of classes that apply the graphic effects to scene graph nodes
javafx.scene.image	Provides set of classes for loading and displaying images
javafx.scene.input	Provides set of classes for the mouse and keyboard events
javafx.scene.layout	Provides set of classes to support user interface layout
javafx.scene.shape	Provides set of 2D classes that performs the operations on objects related to 2D geometry
javafx.scene.text	Provides set of classes for fonts and rendering text nodes
javafx.scene.transform	Provides set of classes that are used to perform rotating, scaling, shearing operations on objects
javafx.scene.web	Provides means for loading and displaying Web content
javafx.stage	Provides top level container classes for JavaFX content
javafx.util	Provides utilities classes
javafx.util.converter	Provides standard string converters for JavaFX

Table 16.4: JavaFX APIs

From Java 11 onwards, JavaFX is not included as part of the SDK, but must be downloaded separately.

Following link can be used to download JavaFX 15.0:

<https://gluonhq.com/products/javafx/>

Once you have downloaded JavaFX, you can configure NetBeans for JavaFX using the steps given here:

<https://openjfx.io/openjfx-docs/#install-javafx>

16.3.1 Components of JavaFX Architecture

JavaFX architecture has several components as shown in figure 16.9.

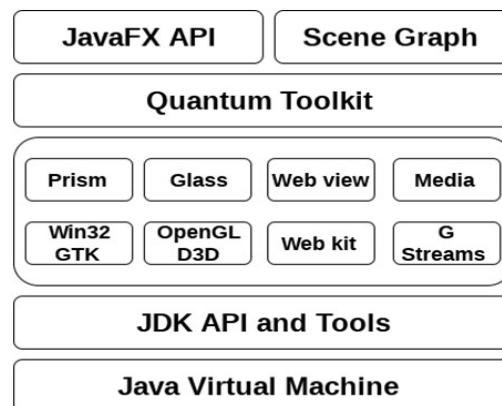


Figure 16.9: JavaFX Architecture

→ Scene Graph

JavaFX applications were coded using a Scene Graph; it is a starting point for construction of JavaFX application. Scene graph is a hierarchical tree of nodes that represent all visual elements of user interface and have capability of handling events. Each node has its separate id, style, and volume. There are various classes present in `javafx.scene` package that are used for creating, modifying, and applying some transformations on the node.

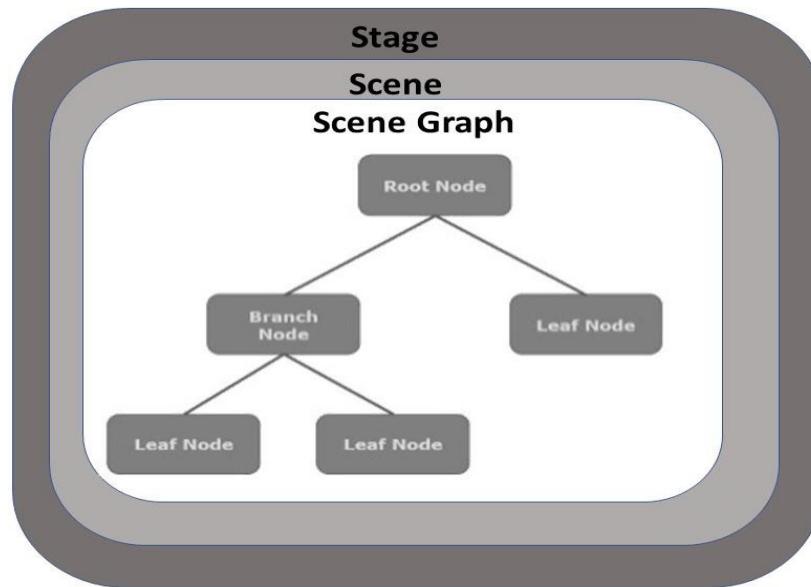


Figure 16.10: Scene Graph

Figure 16.10 shows a scene graph, where a node which does not contain any parents is known as the root node. In the same way, each node can have one or more children and a node without children is termed as leaf node; a node with children is termed as branch node.

→ Graphics Engine

JavaFX graphics engine provides the graphics support to the scene graph. It supports 2D as well as 3D graphics. It provides software rendering when graphics hardware present on the system is not able to support accelerated rendering. Prism and Quantum ToolKit are two graphics accelerated pipelines in the JavaFX.

→ Prism

It is a high performance hardware-accelerated graphics pipeline. It can render both 2D and 3D graphics. It is also able to render graphics on different platforms.

- a. DirectX 9 on Windows XP or vista
- b. DirectX 11 on Windows 7
- c. OpenGL on Mac, Linux, and Embedded

→ Quantum Tool Kit

It is used to bind Prism and Glass Windowing Toolkit (GWT) together and makes them available for layers in stack.

→ Glass Windowing Tool Kit

It is part of lowest level in JavaFX graphics stack and it is platform-dependent, which works as an interface between JavaFX and native operating system. Also, it is responsible for providing operating system services such as managing windows, timers, event queues, and surfaces.

→ Web View

It is used for embedding the HTML content to a JavaFX scene graph. Web View uses Web kit is an internal open source browser and used to render HTML5, Document Object Model (DOM), Cascading Style Sheets (CSS), and JavaScript. It renders HTML content from JavaFX application and applies CSS styles to user interfaces.

→ Media Engine

JavaFX application can support audio and video using Media Engine. It depends upon an open source engine called G Streamer. Package `javafx.scene.media` contains all classes and interfaces that support media functionalities to JavaFX applications.

16.3.2 JavaFX Application Structure

JavaFX application has majorly three components Stage, Scene, and Nodes as shown in figure 16.11.

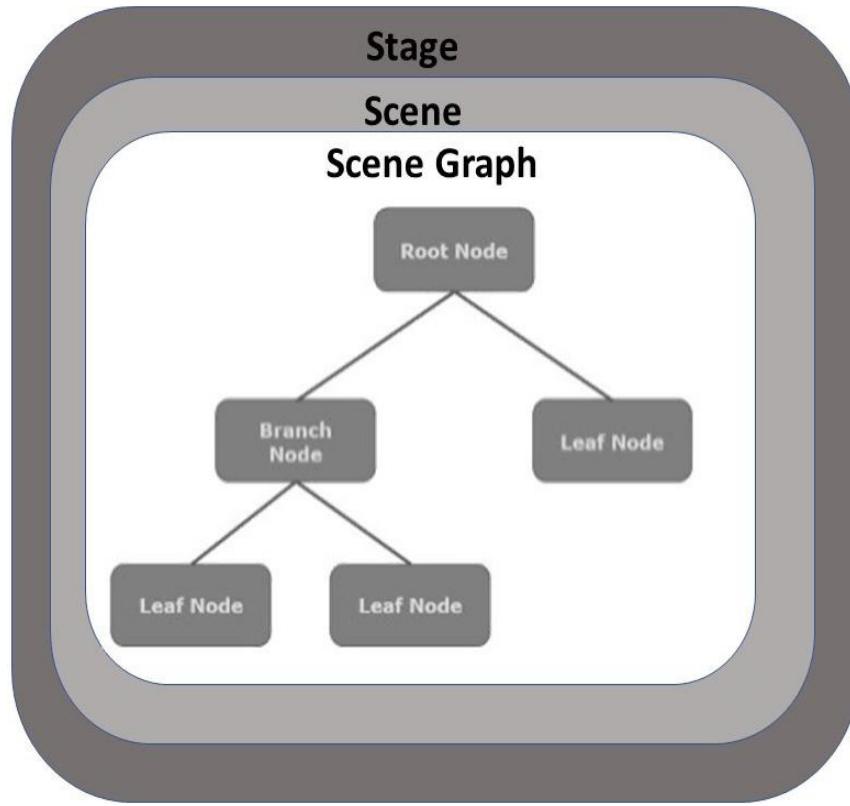


Figure 16.11: JavaFX Application Structure

→ Stage

It is a Window that contains all objects of a JavaFX application and it is represented by Stage class of package javafx.stage. Primary stage is created by platform itself and then, created stage object is passed as an argument to the start() method of Application class. Stage has two parameters determining its position namely, Width and Height. It is divided as Content Area and Decorations.

There are total five types of stages available as follows:

- Decorated
- Undecorated
- Transparent
- Unified
- Utility

The show() method is used to display contents of a stage. Following are steps to create your first JavaFX application:

- **Step 1:** Launch the NetBeans IDE.
- **Step 2:** Next, go to File and select New Project to open New Project wizard as shown in figure 16.12.

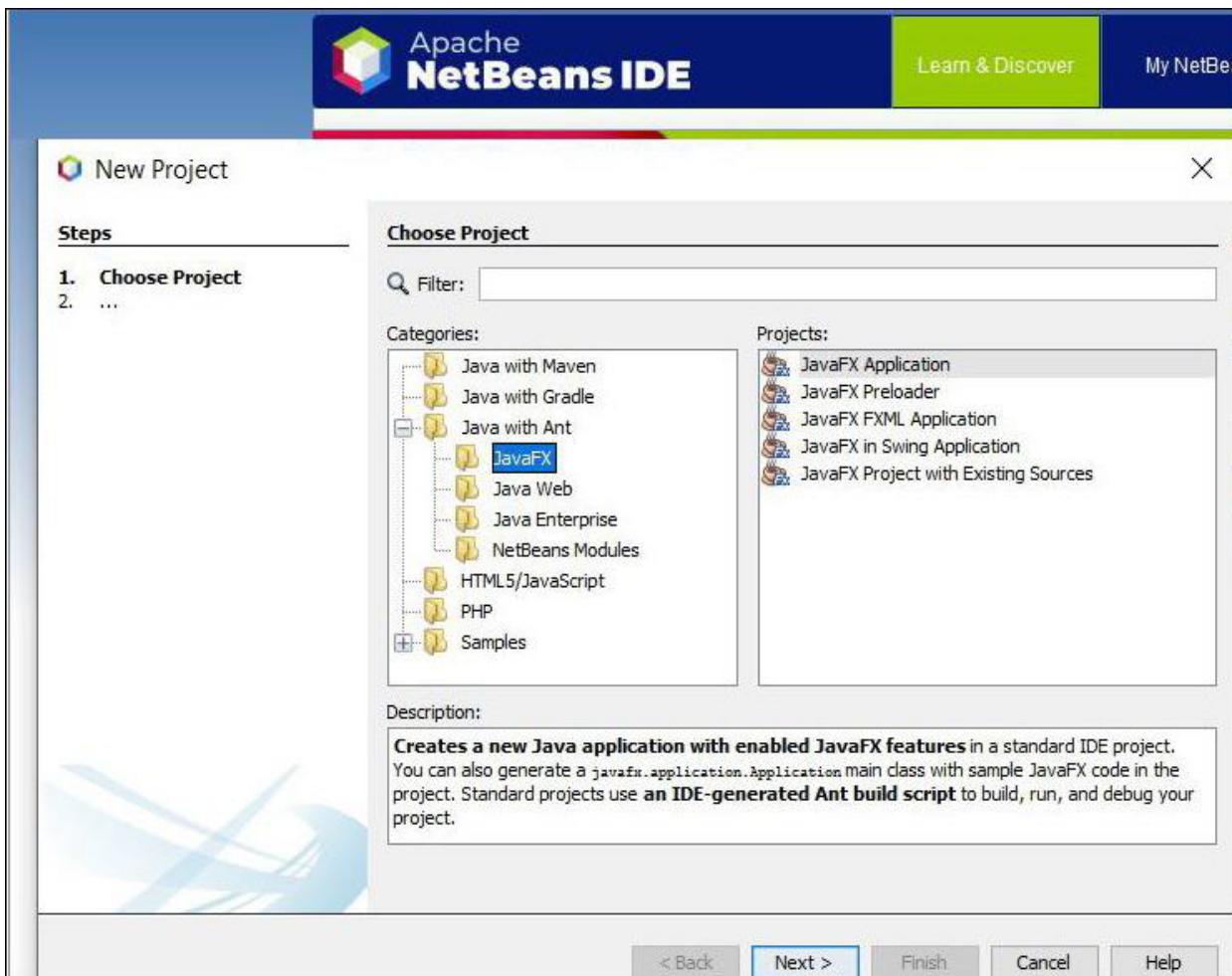
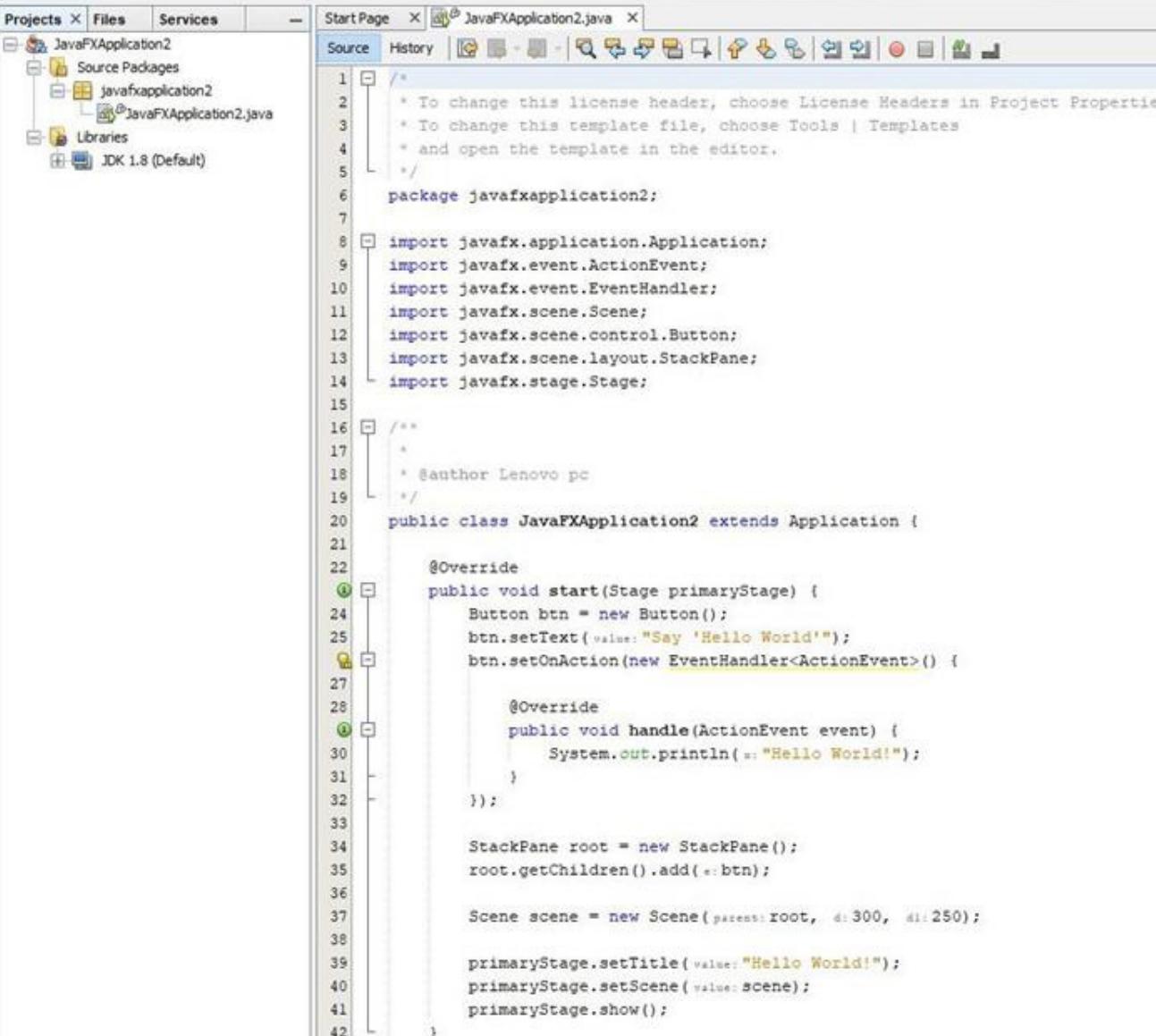


Figure 16.12: New Project Wizard

- **Step 3:** Select JavaFX Application and then, click **Next** and **Finish** on next screen. JavaFX application is created with default details as shown in figure 16.13.



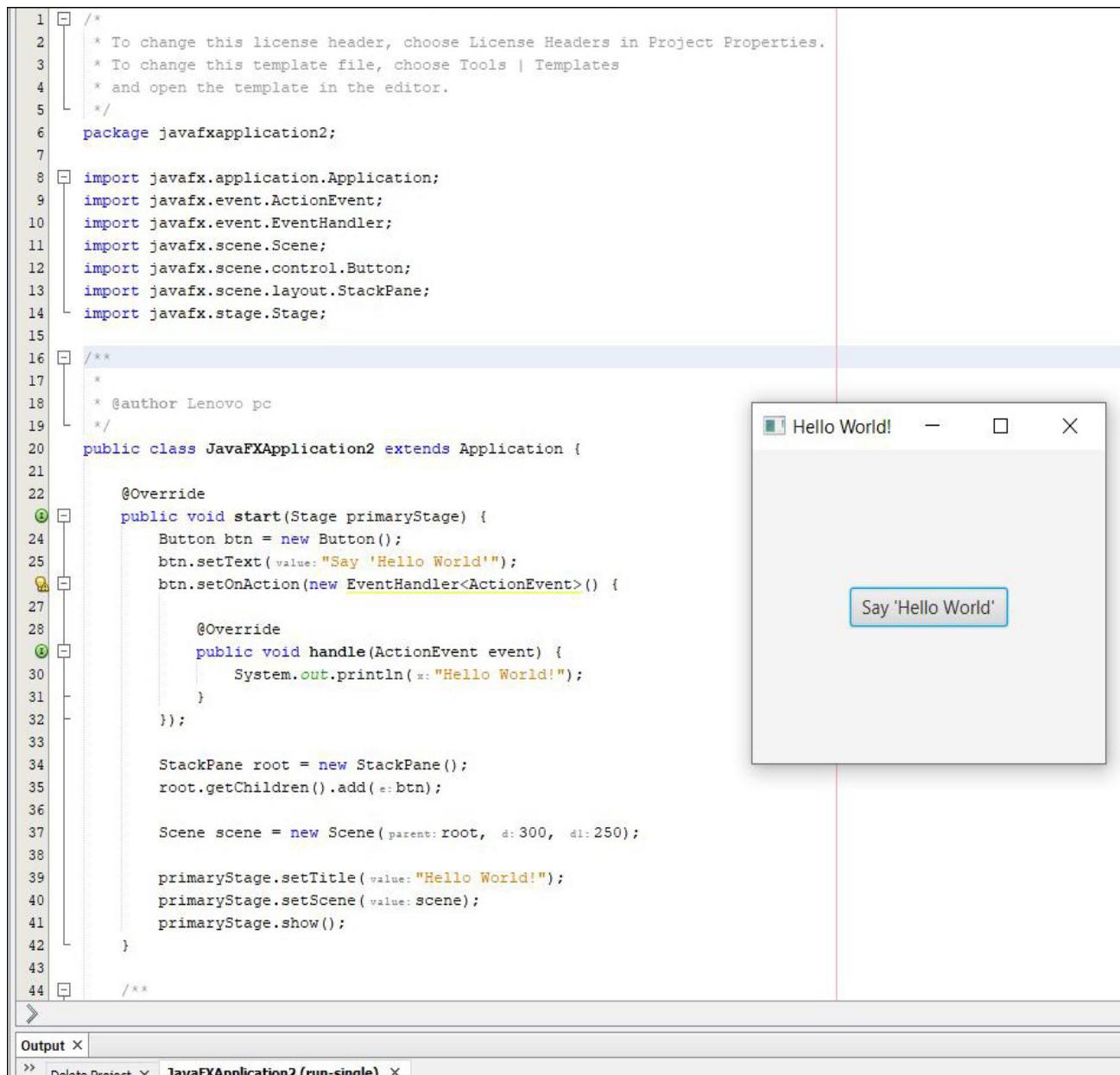
```

1  /*
2   * To change this license header, choose License Headers in Project Properties
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package javafxapplication2;
7
8  import javafx.application.Application;
9  import javafx.event.ActionEvent;
10 import javafx.event.EventHandler;
11 import javafx.scene.Scene;
12 import javafx.scene.control.Button;
13 import javafx.scene.layout.StackPane;
14 import javafx.stage.Stage;
15
16 /**
17 *
18 * @author Lenovo pc
19 */
20 public class JavaFXApplication2 extends Application {
21
22     @Override
23     public void start(Stage primaryStage) {
24         Button btn = new Button();
25         btn.setText("Say 'Hello World!'");
26         btn.setOnAction(new EventHandler<ActionEvent>() {
27
28             @Override
29             public void handle(ActionEvent event) {
30                 System.out.println("Hello World!");
31             }
32         });
33
34         StackPane root = new StackPane();
35         root.getChildren().add(btn);
36
37         Scene scene = new Scene(root, 300, 250);
38
39         primaryStage.setTitle("Hello World!");
40         primaryStage.setScene(scene);
41         primaryStage.show();
42     }
}

```

Figure 16.13: JavaFX Application in NetBeans

After execution of JavaFX application with default code in NetBeans IDE, the output is displayed as shown in figure 16.14.



The screenshot shows a JavaFX application window titled "Hello World!" with a single button labeled "Say 'Hello World'". The application's source code is displayed in the main pane, showing how a button is created, its text is set to "Say 'Hello World'", and an event handler is attached to handle its action.

```

1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6 package javafxapplication2;
7
8 import javafx.application.Application;
9 import javafx.event.ActionEvent;
10 import javafx.event.EventHandler;
11 import javafx.scene.Scene;
12 import javafx.scene.control.Button;
13 import javafx.scene.layout.StackPane;
14 import javafx.stage.Stage;
15
16 /**
17 *
18 * @author Lenovo pc
19 */
20 public class JavaFXApplication2 extends Application {
21
22     @Override
23     public void start(Stage primaryStage) {
24         Button btn = new Button();
25         btn.setText("Say 'Hello World'");
26         btn.setOnAction(new EventHandler<ActionEvent>() {
27
28             @Override
29             public void handle(ActionEvent event) {
30                 System.out.println("Hello World!");
31             }
32         });
33
34         StackPane root = new StackPane();
35         root.getChildren().add(btn);
36
37         Scene scene = new Scene(root, 300, 250);
38
39         primaryStage.setTitle("Hello World!");
40         primaryStage.setScene(scene);
41         primaryStage.show();
42     }
43
44 /**
45 */

```

Output

>> Delete Project X JavaFXApplication2 (run-single) X

Figure 16.14: Output - JavaFX Application

16.3.3 JavaFX 2D Shapes

In some specific applications, there is a requirement for two-dimensional shapes. JavaFX provides the flexibility to create 2D shapes with customized specifications. All classes that implement 2D shapes are part of `javafx.scene.shape` package. Methods inside these classes deal with coordinates of 2D shape creation. Table 16.5 list the commonly used 2D shapes with their description.

Shape	Description
Line	It is a geometrical shape, which connects two points on 2D coordinate system. To create a line in JavaFX application, <code>javafx.scene.shape.Line</code> class requires to be instantiated.
Rectangle	It is a geometrical shape with two pairs of two equal sides and four right angles at their joint. To create a rectangle in JavaFX application, <code>javafx.scene.shape.Rectangle</code> class requires to be instantiated.
Ellipse	Ellipse can be defined as a curve with two focal points. To create it in JavaFX application, <code>javafx.scene.shape.Ellipse</code> class requires to be instantiated.
Arc	Arc can be defined as part of circumference of the circle of ellipse. In JavaFX application, <code>javafx.scene.shape.Arc</code> class requires to be instantiated to create Arcs.
Circle	Circle is the special type of Ellipse having both the focal points at the same location. In JavaFX application, a circle can be created by instantiating <code>javafx.scene.shape.Circle</code> class.
Polygon	It is a geometrical shape that can be created by joining the multiple Co-planner line segments. In JavaFX application, <code>javafx.scene.shape.Polygon</code> class requires to be instantiated in order to create a polygon.
Cubic Curve	It is a curve of degree three in the XY plane. In JavaFX application, <code>javafx.scene.shape.CubicCurve</code> class requires to be instantiated in order to create Cubic Curves.
Quad Curve	It is a curve of degree two in the XY plane. In JavaFX application, <code>javafx.scene.shape.QuadCurve</code> class requires to be instantiated in order to create a QuadCurve.

Table 16.5: JavaFX 2D Shapes

Code Snippet 6 demonstrates how to create polygon using JavaFX.

Code Snippet 6:

```
package javafxapplication2;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.scene.shape.Polygon;
public class JavaFXApplication2 extends Application {
    @Override
    public void start(Stage primaryStage) {
```

```
Group root = new Group();
primarystage.setTitle("Polygon Example");
Polygon polygon = new Polygon();
polygon.getPoints().addAll(new Double[] {
    0.0, 0.0,
    100.0, 200.0,
    200.0, 100.0 });
root.getChildren().add(polygon);
Scene scene = new Scene(root,300,400);
primarystage.setScene(scene);
primarystage.show();
}
public static void main(String[] args) {
    launch(args);
}
}
```

Figure 16.15 displays the output of Code Snippet 6.

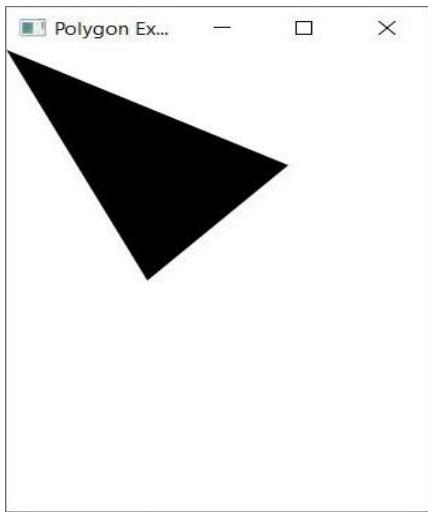


Figure 16.15: Polygon Generated Through JavaFX

Similarly, JavaFX applications can use Text, Effects, Transformation, Animation, 3D Shapes, Layouts, and so on to create interactive GUI applications.

16.4 Check Your Progress

1. Which of the following Swing component is used to create a label button?

(A)	JTextPane	(C)	JDialog
(B)	JTextArea	(D)	JButton

2. Which of the following layout is used as default layout by Swing applications?

(A)	BorderLayout	(C)	CardLayout
(B)	FlowLayout	(D)	GridLayout

3. Which of the following component positions children of its associated container according to a set of constraints?

(A)	SpringLayout	(C)	CardLayout
(B)	GroupLayout	(D)	GridLayout

4. A node that does not have any parents is known as the _____ node.

(A)	Parent	(C)	Branch
(B)	Leaf	(D)	Root

5. Which of the following is useful to embed HTML content in to a scene graph?

(A)	Prism	(C)	Quantum Toolkit
(B)	GWT	(D)	Web View

16.4.1 Answers

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

Summary

- Swing is a part of Java Foundation Classes (JFC) that is used to create desktop applications.
- JButton class is used to create a labeled button.
- JColorChooser class is used to create a color chooser dialog box.
- ActionListeners perform event handling based on actions done on GUI.
- Layout Managers are useful to arrange elements on GUI at specific position or flow.
- JavaFX is a library designed to help developers to create Rich Internet and Desktop GUI applications.
- Stage is a window that contains all objects of a JavaFX application.

Try it Yourself

1. Create a JavaFX GUI application (called GUICounter) having a button with caption **Count**. Each time the 'Count' button is clicked, the counter value should increase by 1.
2. Write an Swing/JavaFXGUI application called SumAccumulator, which has four components:
 - ➔ a Label saying Enter an integer and press ENTER
 - ➔ an input TextField
 - ➔ a Label saying The accumulated sum is
 - ➔ a protected (read-only) TextField for displaying the accumulated sum

The four GUI components are placed inside a container Frame, arranged in FlowLayout. The program must accumulate the numbers entered into the input TextField and display the accumulated sum on the display TextField.