SPPU New Syllabus

A Book Of

Java Programming

For M.C.A. (Management) : Semester - I

[Course Code IT-11 : Credit - 03]

CBCS Pattern

As Per New Syllabus, Effective from June 2020

Mr. Satyavan M. Kunjir
M.C.S. NET
Assistant Professor, Computer Science Dept.
Dr. D. Y. Patil ACS College, Pimpri Pune

Mrs. Madhuri Khadtare
M. Sc. (Computer Science), NET
Assistant Professor
MES Abasaheb Garware College, Pune - 4

Mrs. Malati V. Tribhuwan(Nikam)
M.SC. (Information Technology)
Assistant Professor
Dr. D.Y. Patil ACS College, Pimpri Pune

Price 470.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N5314

JAVA PROGRAMMING **ISBN 978-93-90437-82-5**

Second Edition : November 2022

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041

DISTRIBUTION CENTRES

PUNE

Nirali Prakashan
(For orders outside Pune)
S. No. 28/27, Dhayari Narhe Road, Near Asian College
Pune 411041, Maharashtra
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

Nirali Prakashan
(For orders within Pune)
119, Budhwar Peth, Jogeshwari Mandir Lane
Pune 411002, Maharashtra
Tel : (020) 2445 2044; Mobile : 9657703145
Email : niralilocal@pragationline.com

MUMBAI
Nirali Prakashan
Rasdhara Co-op. Hsg. Society Ltd., 'D' Wing Ground Floor, 385 S.V.P. Road
Girgaum, Mumbai 400004, Maharashtra
Mobile : 7045821020, Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES

DELHI
Nirali Prakashan
Room No. 2 Ground Floor
4575/15 Omkar Tower, Agarwal Road
Darya Ganj, New Delhi 110002
Mobile : 9555778814/9818561840
Email : delhi@niralibooks.com

BENGALURU
Nirali Prakashan
Maitri Ground Floor, Jaya Apartments,
No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003
Karnataka; Mob : 9686821074
Email : bengaluru@niralibooks.com

NAGPUR
Nirali Prakashan
Above Maratha Mandir, Shop No. 3,
First Floor, Rani Jhanshi Square,
Sitabuldi Nagpur 440012 (MAH)
Tel : (0712) 254 7129
Email : nagpur@niralibooks.com

KOLHAPUR
Nirali Prakashan
438/2, Bhosale Plaza, Ground Floor
Khasbag, Opp. Balgopal Talim
Kolhapur 416 012, Maharashtra
Mob : 9850046155
Email : kolhapur@niralibooks.com

JALGAON
Nirali Prakashan
34, V. V. Golani Market, Navi Peth,
Jalgaon 425001, Maharashtra
Tel : (0257) 222 0395
Mob : 94234 91860
Email : jalgaon@niralibooks.com

SOLAPUR
Nirali Prakashan
R-158/2, Avanti Nagar, Near Golden
Gate, Pune Naka Chowk
Solapur 413001, Maharashtra
Mobile 9890918687
Email : solapur@niralibooks.com

marketing@pragationline.com | www.pragationline.com

Also find us on www.facebook.com/niralibooks

Preface ...

We take this opportunity to present this book entitled as **"Java Programming"** to the students of First Semester - MCA (Management). The object of this book is to present the subject matter in a most concise and simple manner. The book is written strictly according to the New Syllabus (CBSC Pattern).

The book has its own unique features. It brings out the subject in a very simple and lucid manner for easy and comprehensive understanding of the basic concepts, its intricacies, procedures and practices. This book will help the readers to have a broader view on Java Programming. The language used in this book is easy and will help students to improve their vocabulary of Technical terms and understand the matter in a better and happier way.

We sincerely thank Shri. Dineshbhai Furia and Shri. Jignesh Furia of Nirali Prakashan, for the confidence reposed in me and giving me this opportunity to reach out to the students of management studies.

We have given our best inputs for this book. Any suggestions towards the improvement of this book and sincere comments are most welcome on niralipune@pragationline.com.

Authors

■■■

Syllabus ...

1. Introduction **[Weightage 3]**

1.1 About Java

1.2 Flavours of Java

1.3 Java Installation

1.4 Java Program Development Environment

Extra reading: Docs Oracle Docs

2. Object Oriented Programming **[Weightage 8]**

2.1 Class Fundamentals

2.2 Object and Object Reference

2.3 Object Life time and Garbage Collection

2.4 Creating and Operating Objects

2.5 Constructor and Initialization Code Block

2.6 Access Control, Modifiers, Use of Modifiers with Classes and Methods

2.7 Nested, Inner Class and Anonymous Classes, Abstract Class and Interfaces

2.8 Methods, Defining Methods, Argument Passing Mechanism, Method Overloading, Recursion, Dealing with Static Members, Finalize() Method, Native Method

2.9 Use of "this" reference

2.10 Design of Accessors and Mutator Methods

2.11 Cloning Objects, Shallow and Deep Cloning

2.12 Generic Class Types

Extra Reading: OCA Java Programmer - I Exam Kathy Sierra

3. Extending Classes and Inheritance **[Weightage 6]**

3.1 Use and Benefits of Inheritance in OOP

3.2 Types of Inheritance in Java

3.3 Inheriting Data Members and Methods

3.4 Role of Constructors in Inheritance

3.5 Overriding Super Class Methods, Use of "super"

3.6 Polymorphism in Inheritance

3.7 Type Compatibility and Conversion

3.8 Implementing Interfaces

Extra Reading: Understanding and practicing above concept in Depth - OCA Java Programmer - I Exam Kathy Sierra

4. Package **[Weightage 3]**

4.1 Organizing Classes and Interfaces in Packages
4.2 Package as Access Protection
4.3 Defining Package
4.4 Classpath Setting for Packages
4.5 Making JAR Files for Library Packages
4.6 Import and Static Import
4.7 Naming Convention for Packages.

Extra Reading: Oracle Javase Tutorial

5. Exception Handling **[Weightage 6]**

5.1 The Idea behind Exception
5.2 Exceptions and Errors
5.3 Types of Exception
5.4 Control Flow in Exceptions
5.5 JVM reaction to Exceptions
5.6 Use of try, catch, finally, throw, throws in Exception Handling
5.7 In-built and User Defined Exceptions Checked and Un-Checked Exceptions

Extra Reading: Oracle Javase Tutorial

6. Array and String **[Weightage 4]**

6.1 Defining an Array
6.2 Initializing and Accessing Array
6.3 Multi-Dimensional Array
6.4 Operation on String, Mutable and Immutable String
6.5 Using Collection Bases Loop for String, Tokenizing a String
6.6 Creating Strings using StringBuffer, String Builder

Extra Reading: Java arrays, tokenizer applications–Jenkov Tutorials

7. Thread **[Weightage 6]**

7.1 Understanding Threads
7.2 Needs of Multi-Threaded Programming
7.3 Thread Life-Cycle
7.4 Thread Priorities
7.5 Synchronizing Threads
7.6 Inter Communication of Threads
7.7 Critical Factor in Thread – Deadlock

Extra Reading: Animation Using Thread

8. A Collection of Useful Classes **[Weightage 6]**

8.1 Utility Methods for Arrays
8.2 Observable and Observer Objects
8.3 Date and Times
8.4 Using Scanner
8.5 Regular Expression
8.6 Input/Output Operation in Java (java.io Package)
8.7 Streams and the New I/O Capabilities
  8.7.1 Understanding Streams
  8.7.2 The Classes for Input and Output
  8.7.3 The Standard Streams
8.8 Working with File Object
  8.8.1 File I/O Basics
  8.8.2 Reading and Writing to Files
  8.8.3 Buffer and Buffer Management
  8.8.4 Read/Write Operations with File Channel
8.9 Serializing Objects

Extra Reading: regex – Pattern matching, split examples, reading and writing Character Stream, Bytestream and Objects in java files.

9. UI Programming **[Weightage 12]**

9.1 Designing Graphical User Interfaces in Java
9.2 Components and Containers
9.3 Basics of Components
9.4 Using Containers
9.5 Layout Managers
9.6 AWT Components
9.7 Adding a Menu to Window
9.8 Extending GUI Features Using Swing Components

Extra Reading: Using Swing toolkit GUI –oracle java tutorial

10. Event Handling **[Weightage 10]**

10.1 Event-Driven Programming in Java
10.2 Event- Handling Process
10.3 Event Handling Mechanism
10.4 The Delegation Model of Event Handling
10.5 Event Classes, Event Sources, Event Listeners
10.6 Adapter Classes as Helper Classes in Event Handling

Extra Reading: Hierarchy of Event Classes, Event Sources, Event Listeners- Oracle java docs

11. The Collection Framework **[Weightage 10]**

11.1 Introduction to Java Frameworks

11.2 Collections of Objects

11.3 Collection Types, Sets, Sequence, Map

11.4 Understanding Hashing

11.5 Use of ArrayList and Vector

11.6 Java Utilities (java.util Package)

Extra Reading: Searching, Sorting, Insertion, Manipulation, Deletion of Data using Java Collections

12. Database Programming using JDBC **[Weightage 10]**

12.1 Introduction to JDBC

12.2 JDBC Drivers and Architecture

12.3 CURD Operation Using JDBC

12.4 Connecting to Non-Conventional Databases

Extra Reading: List of JDBC Drivers and Jars, Statement, Prepared Statement and Callable Statement.

13. Java Server Technologies **[Weightage 16]**

13.1 Servlet Web Application Basics

13.2 Architecture and Challenges of Web Application

13.3 Introduction to Servlet

13.4 Introduction to JSP

13.5 Servlet Life Cycle

13.6 Developing and Deploying Servlets, Exploring Deployment Descriptor(web.xml)

13.7 Handling Request and Response

Java Extra Reading: Session Handling 4 Methods, RequestDispatcher, JSP Tags, JSP Implicit Objects, Generic Servlet

■■■

Contents ...

■■■

1... Introduction

Objectives...

- To understand Concept of Java.
- To learn Features of Java.
- To learn how the Java Language is Structured.
- To study how to install the JDK.
- To learn how to Write, Compile and Execute Java Programs.

1.1 INTRODUCTION TO JAVA

- Like any programming language, the Java language has its own structure, syntax rules, and programming paradigm.
- Java is a programming language and computing platform first released by Sun Microsystems in 1995 later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton.
- It is an object-oriented language similar to C++, but with advanced and simplified features.
- Java is independent programming language that follows the logic of "Write once, Run anywhere" i.e. the compiled code can run on all platforms which supports java.

1.2 FLAVORS OF JAVA

- The following are some important features of Java programming language:
  1. **Simple:** Java programming language is very simple and easy to learn, understand. It contains many features of other programming languages like C and C++. In java, many complicated features like pointers, operator overloading, structures, unions, etc. have been removed. Java is considered as one of simple language because it does not have complex features like Operator overloading, multiple inheritance, and pointers.
  2. **Object-Oriented:** In Java, object-oriented programming concepts are similar to C++. Java is said to be a pure object-oriented programming language. In java, everything is an object. Java provides a clean and efficient object-based development platform. It has all object oriented features such as abstraction, encapsulation, inheritance and polymorphism.

3. **Platform Independent:** Platform independent language means once compiled you can execute the program on any platform (OS). Java is platform independent.
4. **Portable:** Java is called portable because you can compile a Java code which will send out a byte-code, and then you run that code with Java Virtual Machine.
5. **Robust:** Java is robust as it is capable of handling run-time errors, supports automatic garbage collection and exception handling, and avoids explicit pointer concept.
6. **Secure:** Java is a more secure language as compared to C/C++, as it does not allow a programmer to explicitly create pointers. Thus in Java, we can not gain access to a particular variable if we do not initialize it properly.
7. **Multithreaded:** The Java platform is designed with multithreading capabilities built into the language. That means you can build applications with many concurrent threads of activity, resulting in highly interactive and responsive applications.
8. **High Performance:** Java code is compiled into bytecode which is highly optimized by the Java compiler, so that the Java Virtual Machine (JVM) can execute Java applications at full speed. In addition, compute-intensive code can be rewritten in native code and interfaced with Java platform via Java Native Interface (JNI) thus improve the performance.
9. **Distributed:** Java is distributed because it encourages users to create distributed applications.

   In Java, we can split a program into many parts and store these parts on different computers. A Java programmer sitting on a machine can access another program running on the other machine.

   This feature in Java gives the advantage of distributed programming, which is very helpful when we develop large projects. Java helps us to achieve this by providing the concept of RMI (Remote Method Invocation) and EJB (Enterprise JavaBeans).
10. **Dynamic and Extensible:** Java is dynamic and extensible. Means with the help of OOPs, we can add classes and add new methods to classes, creating new classes through subclasses. This makes it easier for us to expand our own classes and even modify them.

1.3 JAVA INSTALLATION

- Java Development Kit (JDK) allows you to code and run Java programs. It's possible that you install multiple JDK versions on the same PC. But it's recommended that you install only latest version.

- Following are steps to install Java in Windows,

  Step 1 : Download latest version of JDK for java.

  Step 2 : Once the download is complete, run the exe to install JDK. Click Next.

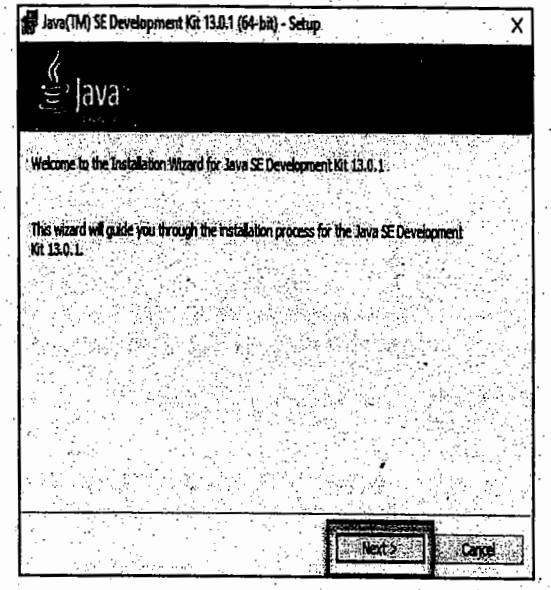


Fig. 1.1

Step 3 : Select the PATH for Java installation and click Next.

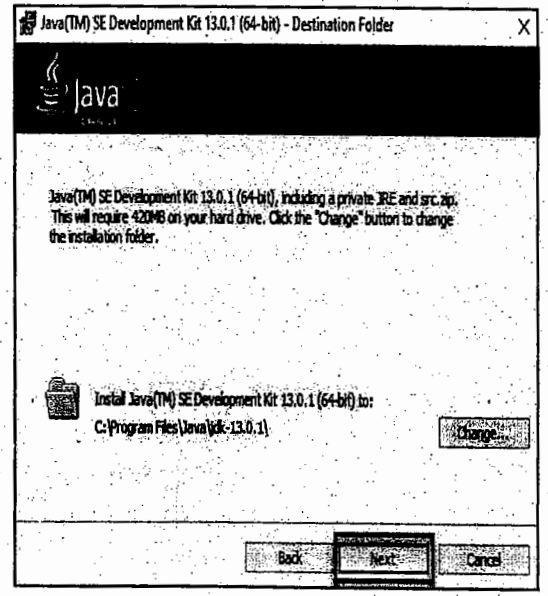


Fig. 1.2

Step 4 : Once installation is complete click Close.

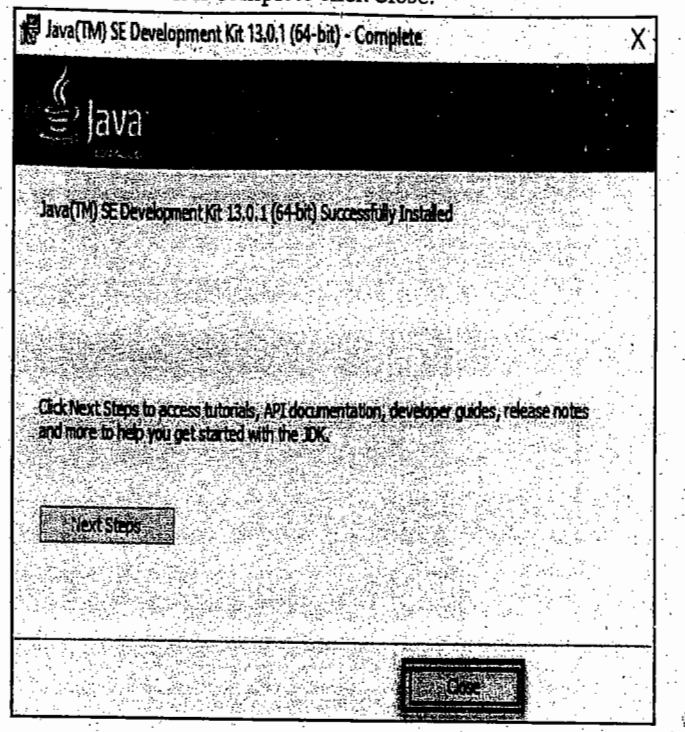


Fig. 1.3

1.3.1 How to set Environment Variables in Java: Path and Classpath

- The PATH variable gives the location of executables like javac, java etc. It is possible to run a program without specifying the PATH but you will need to give full path of executable like **C:\Program Files\Java\jdk-13.0.1\bin\javac A.java** instead of simple **javac A.java**
- The CLASSPATH variable gives location of the Library Files.
- Let's look into the steps to set the PATH and CLASSPATH

Step 1 : Right Click on the My Computer and Select the Properties → Advanced system settings → Environment Variables

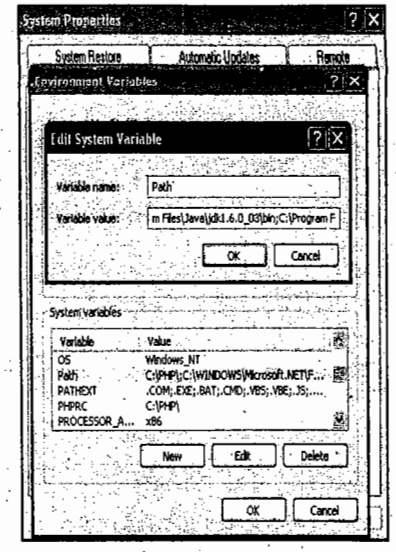


Fig. 1.4

Step 2 : Copy the path of bin folder which is installed in JDK folder and paste Path of bin folder in Variable value and click on OK Button.

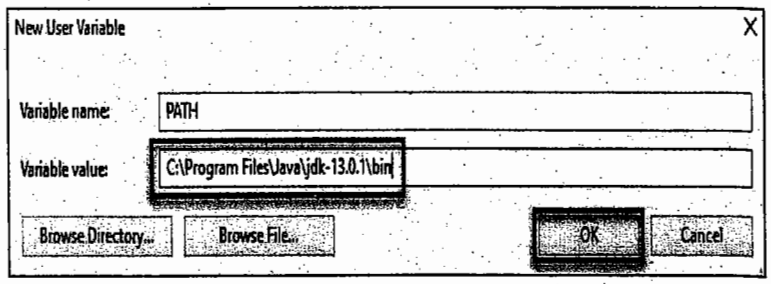


Fig. 1.5

Note : In case you already have a PATH variable created in your PC, edit the PATH variable to PATH = <JDK installation directory>\bin;%PATH%;
Here, %PATH% appends the existing path variable to our new value

Step 3 : You can follow a similar process to set CLASSPATH.

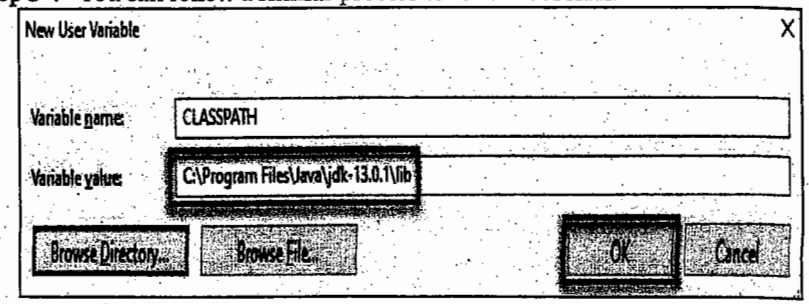


Fig. 1.6

Step 4 : Click on OK button.

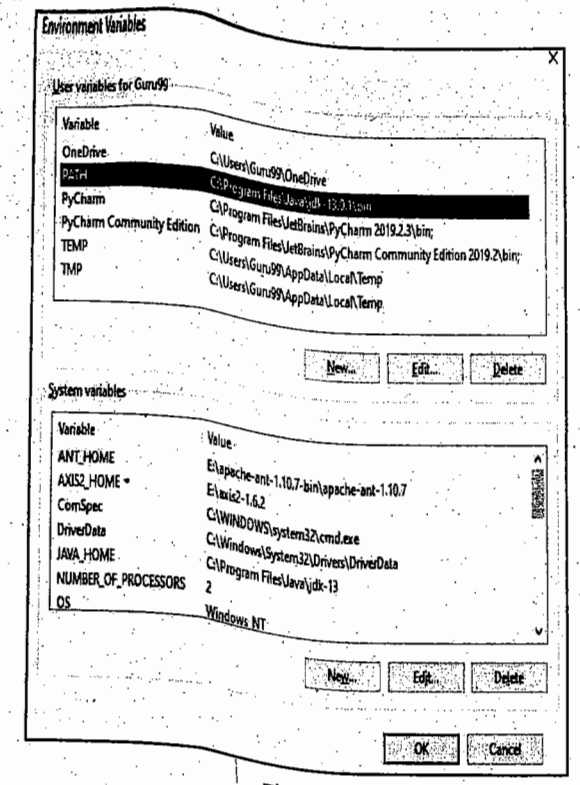


Fig. 1.7

Step 5 : Go to command prompt and type javac command.

1.4 JAVA PROGRAM DEVELOPMENT ENVIRONMENT

- Java files are stored with .java extension. These files are then compiled into Java byte code using the Java compiler, and the byte code is then executed using the Java Virtual Machine (JVM). The Java compiler and the JVM are part of the Java Development Kit.

Java Bytecode:

- Java programs written in the Java language are compiled into Java bytecode which can be executed by the Java Virtual Machine. The Java bytecode is stored in binary .class files.

1.4.1 Java Virtual Machine

- The JVM is a program that provides the runtime environment necessary for Java programs to execute.
- JVM is a part of Java Run Environment (JRE).
- Java programs cannot run unless there is a JVM available for the appropriate hardware and OS platform we wish to execute on.

- You start up the JVM and tell it what Java code to execute. The Java compiler doesn't create an object file, but instead it creates a bytecode file which is, essentially, an object file for a virtual machine. In fact, the Java compiler is often called the JVM compiler (for Java Virtual Machine).
- Consequently, you can write a Java program (on any platform) and use the JVM compiler (called javac) to generate a bytecode file (bytecode files use the extension .class).
- This bytecode file can be used on any platform (that has installed Java). However, bytecode is not an executable file. To execute a bytecode file, you actually need to invoke a Java interpreter (called java). Every platform has its own Java interpreter which will automatically address the platform-specific issues that can no longer be put off. When platform-specific operations are required by the bytecode, the Java interpreter links in appropriate code specific to the platform.
- The JVM performs following operation:
  1. Loads code
  2. Verifies code
  3. Executes code
  4. Provides runtime environment

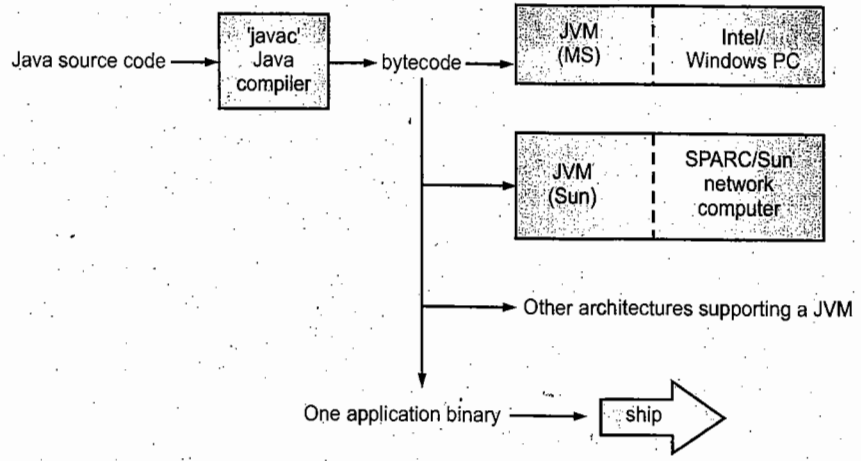


Fig. 1.8: Java Virtual Machine Environment

1.4.2 Just-In-Time (JIT) Compiler

- The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time.
- It is mainly responsible for performance optimization of Java-based applications at run time or execution time.
- In order to speed up the performance, JIT compiler communicates with JVM at the execution time in order to compile byte code sequences into native machine code. Basically, when using JIT Compiler, the native code is easily executed by the hardware as compared to JVM Interpreter.

- When the JIT Compiler compiles the series of bytecode, it also performs certain optimizations such as data analysis, translation from stack operations to register operations, eliminating subexpressions, etc. This makes Java very efficient when it comes to execution and performance.

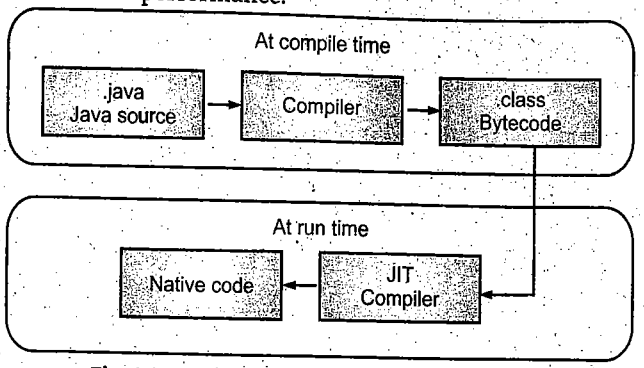


Fig. 1.9: Working of Just In Time (JIT) Compiler

1.4.3 How to Write Java Program and How to Compile and Run

- The basic steps to create the Hello World program are:
  1. Open a simple text editor program such as Notepad write the program in Java.
  2. Compile the source code.
  3. Run the program.

Program 1.1: Write a sample Java code for display Hello World.

```
// Hello World Program
class HelloWorld
{
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}
```

Output:

```
Hello World
```

- Note the lines with "//" above. These are comments in Java, and the compiler ignores them.
  1. Line 1 is a comment, introducing this program.
  2. Line 2 creates a class HelloWorld. All code needs to be in a class in order for the Java runtime engine to run it. Note that the entire class is defined within enclosing curly braces (on line 3 and line 8).
  3. Line 4 is the **main()** method, which is always the entry point into a Java program. It also is defined within curly braces (on line 5 and line 7). Let's break it down:
     - **public**: This method is public and therefore available to anyone.
     - **static**: This method can be run without having to create an instance of the class HelloWorld.
     - **void**: This method does not return anything.
     - **(String[] args)**: This method takes a String argument.

  4. Line 6 writes "Hello World" to the console.
  5. Save your program file as "HelloWorld.java". The filename must match this class name, hence the name "HelloWorld.java". The extension ".java" tells the computer that it is a Java code file.
  6. When you code the Java Program, JRE uses javac compiler to compile the high-level Source code to byte code. That means use the JDK compiler to turn your text file into an executable program. To do this use the javac command, which stands for Java compiler.

     ```
     javac HelloWorld.java
     ```

  7. To execute the code, enter the command java followed by the class name, as expected output Hello World is displayed now.

     ```
     java HelloWorld
     ```

     That invokes the Java Virtual Machine to run the program called HelloWorld (note that there is no .java or .class extension).
  8. Java is case sensitive programming language. All code, commands, and file names should be used in consistent casing. HelloWorld is not same as helloworld.

Summary

- Java is a programming language and computing platform first released by Sun Microsystems in 1995 later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton.
- Compiled code in Java is known as bytecode and carries the .class extension.
- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- JVM is a platform independent execution environment that converts bytecode into machine codes and executes it.
- In a standard form, JVM is an interpreter for bytecode. Therefore, java programs are executed by JVM.
- Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments.
- The execution of every program is under the control of JVM, making Java secure.
- JVM enabling bytecode to be executed on any processor regardless of operating system. Hence making Java Platform Independent.
- JVM is responsible for allocating memory space.
- JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time.
- NetBeans, Eclipse are the examples of java IDE.

Check Your Understanding

1. Which of the following option leads to the portability and security of Java?
   (a) Bytecode is executed by JVM
   (b) Dynamic binding between objects
   (c) The applet makes the Java code secure and portable
   (d) Use of exception handling

2. Which of the following is not a Java features?
   (a) Dynamic (b) Architecture Neutral
   (c) Use of pointers (d) Object-oriented
3. Compiler converts ____.
   (a) Java program to byte code (b) Java program to machine code
   (c) Java program to octal code (d) Java program to hexadecimal code
4. Platform independent code file created from Source file is understandable by ____.
   (a) Applet (b) Compiler
   (c) JRE (d) JVM
5. JVM Stands for ____.
   (a) Java Virtual Mechanism (b) Java Virtual Memory
   (c) Java Virtual Machine (d) Java Virtual Management

Answers

1. (a)	2. (c)	3. (a)	4. (d)	5. (c)

Practice Questions

Q.I Answer the following Questions in short:

1. Define the bytecode.
2. What is JVM?
3. Define JIT?
4. List out features of java.
5. How to run and compile java program?

Q.II Answer the following Questions:

1. List and explain the features of java.
2. Write a note on working of java program?
3. Write short note on Java virtual machine?
4. Write a note on JIT.
5. What is difference between JVM and JIT?

■■■

2... Object Oriented Programming

Objectives...

- To study the Fundamentals of Class.
- To study the difference between Object and Reference Variable.
- To study how Garbage Collection Works?
- To learn how to Create an Object of a Class?
- To understand the Concept of Constructor.
- To study different types of Access Controls.
- To study the concept of Nested, Inner, Anonymous, Abstract Class and Interfaces.
- To study the concept of Method Overloading, Recursion.
- To understand Static Members, Finalize() Method, Native Method.
- To learn Use of "this "reference.
- To understand the Design of Accessors and Mutator Methods.
- To study the Cloning Objects, Shallow and Deep Cloning.
- To study Generic Class Types.

2.1 INTRODUCTION

- Java is pure Object-Oriented programming Language. It supports to the following object oriented concepts: Class, Objects, Polymorphism, Inheritance, Encapsulation, Abstraction

OOPs (Object-Oriented Programming System):

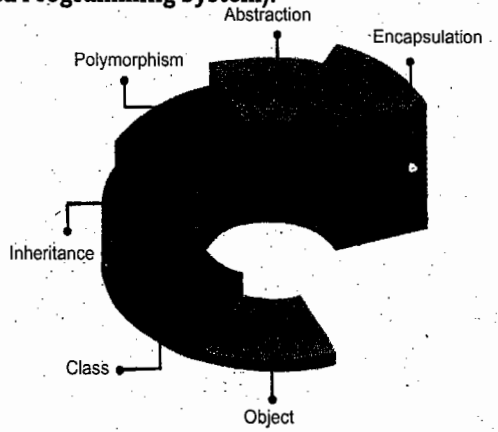


Fig. 2.1: OOPs Concept

1. **Class:**

- *Collection of objects is called class.* It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

2. **Object:**

- *Any entity that has state and behavior is known as an object.* For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.
- **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

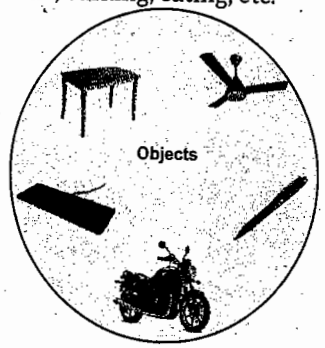


Fig. 2.2: Runtime example of an object

3. **Inheritance:**

- *When one object acquires all the properties and behaviors of a parent object, it is known as inheritance.* It provides code reusability. It is used to achieve runtime polymorphism.

Fig. 2.3: Real Time implementation of inheritance

4. **Polymorphism:**

- *If one task is performed in different ways, it is known as polymorphism.* For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
- In Java, we use method overloading and method overriding to achieve polymorphism.
- Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

5. **Abstraction:**

- *Hiding internal details and showing functionality is known as abstraction.* For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

Capsule

Fig. 2.4: Implementation of abstraction

6. **Encapsulation:**

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation.* For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

2.2 CLASS FUNDAMENTALS

- Class is most important part of object oriented programming. It is user define data type. The Collection of data members and member functions is called class. OR The binding of data members and member functions in a unit that unit is called class. Example: Class is Person.
- **Syntax:**

```
class ClassName [ extends SuperClass]
{
   Variable Declarations
   Method Definition /Declaration
 }
```

- **Example:**

```
class Add
{
   int a, b ,c;
   void disp()
   {
      c=a+b;
      System.out.println("Addition Is " + c);
   }
}
```

- The **ClassName** is nothing but user defined name given to the class, while giving name to the class **naming convention** has to follow. The naming convention is nothing but set of rules: Initial character of class name must be capital and name of class must be meaningful.

- The extends is a keyword used to inherit super class into the sub class.
- The variables and methods declared in a class without using any keyword are called instance variables and instance methods respectively because for accessing them object of class has to use.

2.3 OBJECT AND OBJECT REFERENCE

2.3.1 Creating and Operating Object

- Any real or runtime entity is called object. Objects have states and behaviors.
- An object of a class can be created by using new operator as follow:
  Syntax: `ClassName obj=new ClassName ();`
- By using an object of a class we can access data member and member functions of class.

```
obj.data_member
obj.functionName()
```

- **Example:**

```
class Add
{
   int a, b ,c;
   void disp()
   {
      c=a+b;
      System.out.println("Addition Is" + c);
   }
}
```

- We can create an object of Add class as follow:

```
Add obj=new Add();
obj.a=10;
obj.b=20;
obj.disp();
```

2.3.2 Object Reference

- In java there is no pointer because there is reference variable, it works as pointer. It can be declared as follows:

```
ClassName Var_Name;
```

- **Example:**

```
class Emp
{
   int eno,sal;
   String en;
   void disp()
   {
      System.out.println( eno + " " + en + " " + sal);
   }
}
Emp ob;
```


- ob is reference variable of class Emp, it always points to the null value, we cannot access data member and member functions of a class by using reference variable, if we try to access then compiler throws **NullPointerException.**
- If we store reference of an object into the reference variable of same class then by using reference variable we can access data member and member functions of that class.

```
Emp obj=new Emp();
ob=obj;
ob.eno=10;
ob.en="DPU";
ob.sal=12345;
ob.disp();
```

- If we call method of a class by using reference variable of that class at run time then that concept is called dynamic method dispatching.

2.4 OBJECT LIFE TIME AND GARBAGE COLLECTION

2.4.1 Object Life Time

In Java, it has seven states in Object lifecycle. They are:

1. Created
2. In use
3. Invisible
4. Unreachable
5. Collected
6. Finalized
7. De-allocated

1. Created:

- The following are the some actions performed when an object is created; new memory is allocated for an object.
- Once the object has been created, assuming that it is assigned to some variable and then it directly moves to the In Use state.

2. In use:

- Objects that are held by at least one strong reference are considered to be "In Use".

3. Invisible:

- An object is in the "Invisible" state when there are no longer any strong references that are accessible to the program, even though there might still be references.

4. Unreachable:

- An object enters an "unreachable" state when no more strong references to it exist. When an object is unreachable then it is a state for collection.
- It is important to note that not just any strong reference will hold an object in memory. These must be references that chain from a garbage collection root.
- Garbage collection roots are a special class of variable that includes, Temporary variables on the stack.

5. Collected:

- An object is in the "collected" state when the garbage collector has recognized an object as unreachable and readies it for final processing as a precursor to de-allocation. If the object has a finalize method, then it is marked for finalization.

6. **Finalized:**

- An object is in the "finalized" state if it is still unreachable after it's finalize method, if any, has been run. A finalized object is awaiting de-allocation. If you are considering using a finalized to ensure that important resources are freed in a timely manner, you might want to reconsider. To lengthening object lifetimes, finalize methods can increase object size.

7. **De-allocated:**

- The de-allocated state is the final step in garbage collection. If an object is still unreachable after all the above work has done, then this is the state for de-allocation.

2.4.2 Garbage Collection

- Garbage means unreferenced objects.
- Java garbage collection is the process by which Java programs perform automatic memory management. Garbage Collection is a technique used to delete all objects(Java & NonJava) from heap which are not used by any Java program from long time. It runs continuously at back end.
- Object can be unreferenced in following ways:
  1. By nulling the reference
  2. By assigning a reference to another
  3. By anonymous object etc.

  1. By nulling a reference:
  ```
  Employee e=new Employee();
  e=null;
  ```
  2. By assigning a reference to another:
  ```
  Employee e1=new Employee();
  Employee e2=new Employee();
  e1=e2; //now the first object referred by e1 is available for garbage collection
  ```
  3. By anonymous object:
  ```
  new Employee();
  ```
- Garbage collection can be implemented by calling either finalize() or gc() method explicitly:
  - **finalize () Method:**
    - The finalize () method is invoked each time before the object is garbage collected.
    - This method is defined in Object class as:
      ```
      protected void finalize(){}
      ```
  - **gc() Method:**
    - The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.
      ```
      public static void gc(){}
      ```

Program 2.1: Java Program for the implementation of garbage collector.

```
public class TestGarbage1
{
   public void finalize()
   {
      System.out.println("object is garbage collected");
   }
```



```
   public static void main(String args[])
   {
      TestGarbage1 s1=new TestGarbage1();
      TestGarbage1 s2=new TestGarbage1();
      s1=null;
      s2=null;
      System.gc();
   }
}
```

Output:

```
object is garbage collected
object is garbage collected
```

2.5 METHODS, DEFINING METHODS, ARGUMENT PASSING MECHANISM, METHOD OVERLOADING, RECURSION

2.5.1 Method, Defining Methods

- A self contained block of statements are used to perform a specific task is called method.
- A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.
- It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.
- The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.

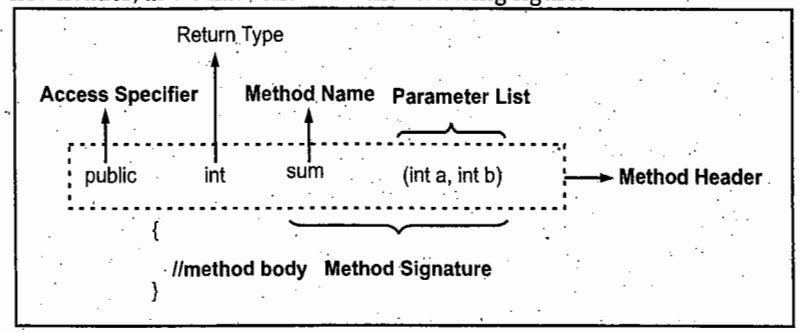


Fig. 2.5: Prototype of a Function

- **Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.
- **Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier: default, public, private, and protected. We learn these access specifiers in details in point 2.6.

- **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
- **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction()**. A method is invoked by its name.
- Naming convention:
  1. While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:
     - **Single-word method name:** `sum()`, `area()`
     - **Multi-word method name:** `areaOfCircle()`, `stringComparision()`
  2. There are different ways in which parameter data can be passed into and out of **methods and functions**. Let us assume that a function *B()* is called from another function *A()*. In this case *A* is called the ***"caller function"*** and *B* is called the ***"called function or callee function"***. Also, the arguments which *A* sends to *B* are called *actual arguments* and the parameters of *B* are called *formal arguments*.
- **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

  Types of Parameters:
  - **Formal Parameters:** The variables and their types as they appear in the prototype of the function or method called Formal Parameters

    Syntax: `function_name (datatype variable_name)`
  - **Actual Parameters:** The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

2.5.2 Arguments Passing Mechanism

Parameters can be passed to the function in two ways:

1. **Pass By Value:**

   Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as ***call by value***.

Program 2.2: Write a java program to calculate factorial of a number.

```
class Fact
{
   int i ,f=1;
   void cal(int n)
   {
      for(i=1;i<=n;i++)
      {
         f=f*i;
      }
      System.out.println ("Factorial of a number " + n + " is " + f);
   }
}
public class FunDemo
{
   public static void main(String args[])
   {
      int x;
      x=Integer.parseInt(args[0]);
      Fact obj=new Fact();
      obj.cal(x);
   }
}
```

Output:

```
5
Factorial of a number 5 is 120
```

2. **Call by Reference (aliasing):**

   Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as **call by reference**. This method is efficient in both time and space.

Program 2.3: Write a java program for the implementation of call by reference function.

```
class CallByReference
{
   int a, b;
   CallByReference(int x, int y)
   {
      a = x;
      b = y;
   }
```

```
    void ChangeValue(CallByReference obj)
    {
        obj.a += 10;
        obj.b += 20;
    }
}
public class Main
{
    public static void main(String[] args)
    {
        CallByReference object = new CallByReference(10, 20);
        System.out.println("Value of a: " +object.a+ " & b: "  + object.b);
        object.ChangeValue(object);
        System.out.println("Value of a: " +object.a + " & b: " + object.b);
    }
}
```

Output:

```
Value of a: 10 & b: 20
Value of a: 20 & b: 40
```

2.5.3 Method Overloading

- We can implement polymorphism at compile time by using methods overloading.
- In a same class, same function name with different parameter list such concept is called method overloading.
- In method overloading a method can be overloaded in three ways:
  1. Number of parameters.
     For example: This is a valid case of overloading
     `add(int, int)`
     `add(int, int, int)`
  2. Data type of parameters.
     For example:
     `add(int, int)`
     `add(int, float)`
  3. Sequence of Data type of parameters.
     For example:
     `add(int, float)`
     `add(float, int)`

Invalid case of method overloading:

- When we say argument list, we not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```


- **Method overloading** is an example of **Static Polymorphism.**
- **Points to Note:**
  1. Static Polymorphism is also known as compile time binding or early binding.
  2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Program 2.4: Write a java program to calculate area of circle and Triangle.

```
import java.util.*;
class A
{
    float a;
    void area (float r)
    {
        a=3.14f*r*r;
        System.out.println ("Area of Circle is " + a);
    }
    void area(float b, float h)
    {
        a=0.5f*b*h;
        System.out.println ("Area of Triangle is " + a);
    }
}
class Demo
{
    public static void main (String args[]) throws Exception
    {
        float r,b,h;
        Scanner ob=new Scanner (System.in);
        System.out.println ("Values for R B H");
        r=ob.nextFloat();
        b=ob.nextFloat();
        h=ob.nextFloat();
        A obj=new A();
        obj.area(r);
        obj.area (b,h);
    }
}
```

Output:

```
Values for R B H
5
6
7
Area of Circle is 78.5
Area of Triangle is 21.0
```

2.5.4 Recursion

- A method that calls itself is known as a recursive method. And, this process is known as recursion.

How Recursion works?

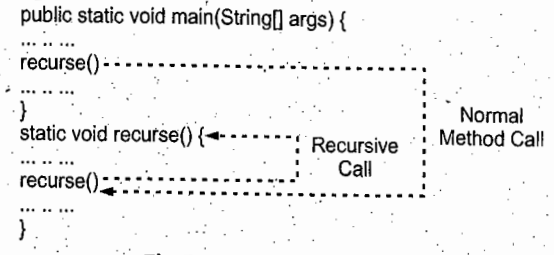


Fig. 2.6: Recursion Working

- In the above example, we have called the recurse () method from inside the main method (normal method call) and, inside the recurse () method, we are again calling the same recurse method. This is a recursive call.
- In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely.
- Hence, we use the **if...else statement** to terminate the recursive call inside the method.

Program 2.5: Factorial of a Number Using Recursion.

```
import java.util.*;
class Factorial
{
   static int factorial( int n )
   {
      if (n != 0)
         return n * factorial(n-1);
      else
         return 1;
   }
   public static void main(String[] args)
   {
      int number, result;
      Scanner ob=new Scanner(System.in);
      System.out.println("Enter the Number");
      number=ob.nextInt();
      result = factorial (number);
      System.out.println ("Factorial Is = " + result);
   }
}
```

Output:

```
Enter the Number
5
Factorial Is = 120
```


2.6 ACCESS CONTROL, MODIFIERS, USE OF MODIFIERS WITH CLASSES & METHODS

2.6.1 Modifiers

- Modifiers are the keywords, if we add them into the class then they affect to the runtime behavior of a class:
  - Access modifiers: public, protected, and private
  - Modifier requiring override: abstract
  - Modifier restricting to one instance: static
  - Modifier prohibiting value modification: final
  - Modifier forcing strict floating point behavior: strictfp
  - Annotations
- Not all modifiers are allowed on all classes, for example an interface cannot be final and an enum cannot be abstract.
- There are two types of modifiers:
  1. Access Control
  2. Non-Access control

2.6.2 Access Control and Use of Modifiers with Classes and Methods

- An access modifier defines the scope or accessibility of a class, constructor, data member and method in another class.
- In java there are four access modifiers:
  1. default
  2. private
  3. protected
  4. public

1. **Default**: The data members, class or methods which are declared without any access modifiers is considered as Default access modifier. The scope of a default modifier is only within the package means it cannot be accessed from outside the package.
2. **Private:** The keyword private is used to specify private access modifiers. The methods or data members declared as private are accessible only within the class in which they are declared. Any other class of same package will not be able to access these members.

Program 2.6: To demonstrate the scope of private variable.

```
class Number1
{
   private int x=10;
   private void display()
   {
      System.out.println("Display Function");
   }
}
```

```
class Number2
{
public static void main(String args[])
{
    Number1 N1=new Number1();
    System.out.println(N1.x);        //Compile Time Error
    N1.display();    //Compile Time Error
    }
}
```

Output:

```
Error: x has private access in Number1
Error: display() has private access in Number1
```

3. **public:** The keyword public is used to specify public access modifiers. Data members, class or methods declared as a public access modifier are accessible from any anywhere. It has the widest scope among all other modifiers.

Program 2.7: To demonstrate the scope of public variable.

```
package p1;
public class Number1
{
    public int x=10;
    public void display ()
    {
        System.out.println ("Display Function ");
    }
}
import p1.*;
classNumber2
{
    public static void main(String args[])
    {
        Number1 N1=new Number1();
        System.out.println(N1.x);
        N1.display();
    }
}
```

Output:

```
10
Display Function
```

4. **Protected:** The keyword protected is used to specify protected access modifiers. The methods or data members declared as protected in super class are accessible by any class within same package or sub classes in different package. The protected access modifier cannot be applied to class and interfaces

Program 2.8: To demonstrate the scope of protected variable.

```
package p1;
public class Number1
{
    protected void display()
    {
        System.out.println("Display Function ");
    }
}
import p1.*;
class Number2 extends class Number1
{
    public static void main(String args[])
    {
        Number1 N1=new Number1();
        N1.display();
    }
}
```

Output:

```
Display function
```

Table 2.1: Scope of Access Specifier

		Default	Private	Protected	Public
1.	Accessible inside the class	Yes	Yes	Yes	Yes
2.	Accessible within the subclass inside same package	Yes	No	Yes	Yes
3.	Accessible outside the package	No	No	No	Yes
4.	Accessible within the subclass outside the package	No	No	Yes	Yes

2.7 CONSTRUCTOR AND INITIALIZATION CODE BLOCK

- A constructor is a special method because its name and class name is same. It is used to initialize instance variables of a class. The constructor gets called automatically when an object of a class is created. Constructors have no explicit return type. Java does not support to the copy constructor directly but it can be implemented by using cloning.

Types of Constructor:

1. **Default /No-argument constructor:** A constructor that has no parameter is known as default constructor.
2. **Parameterized constructor:** A constructor that takes parameter is known as parameterized constructor.

Program 2.9: Write a Java program to demonstrate implementation of constructor.

```
class Number
{
    int num;
    Number ()        //Default Constructor
    {
        num=10;
    }
    Number(int a)    //Parameterized Constructor
    {
        num=a;
    }
}
class CDemo          // Main Class
{
    public static void main(String args[])
    {
        Number obj1=new Number ();
        Number obj2=new Number (20);
        System.out.println(obj1.num+ " " + obj2.num);
    }
}
```

Output:

```
10 20
```

Program 2.10: Write a java program to accept the details of employee(eno,ename,sal) through the command line and display it.

```
class Emp
{
    int eno,sal;
    String en;
    Emp(int e, String ename, int s)
    {
        eno=e;
        en=ename;
        sal=s;
    }
    void disp()
    {
        System.out.println(eno + " " + en + " " +sal);
    }
}
```



```
public class Demo
{
    public static void main(String args[])
    {
        int e,s;
        String en;
        e=Integer.parseInt(args[0]);
        en=args[1];
        s=Integer.parseInt(args[2]);
        Emp obj=new Emp(e,en,s);
        obj.disp();
    }
}
```

Input: 1 abc 8989

Output: 1 abc 8989

2.8 NESTED, INNER CLASS & ANONYMOUS CLASSES, ABSTRACT CLASS & INTERFACES

2.8.1 Nested Class

- A class defined within another class is called as nested class. Nested class can access all the members of outer class including private data members and methods. It logically group classes that are only used in one place, hence are used to develop more readable and maintainable code.
- The scope of a nested class is bounded by the scope of its enclosing class.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e., the enclosing class does not have access to the members of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared *private*, *public*, *protected*, or *package private(default)*.
- Nested classes are divided into two categories:
  - **Static nested class:** Nested classes that are declared *static* are called static nested classes.
  - **Inner class:** An inner class is a non-static nested class.

Syntax:

```
class OuterClass
{
    class NetsedClass
    {
        ---
        ---
    }
}
```

Program 2.11: Write a java program for the implementation of static Nested Class.

```
class OuterClass
{
    static int outer_x = 10;
    int outer_y = 20;
    private static int outer_private = 30;
    static class StaticNestedClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);
            // can access display private static member of outer class
            System.out.println("outer_private = " + outer_private);
            // The following statement will give compilation error
            // as static nested class cannot directly access non-static member
            // System.out.println("outer_y = " + outer_y);
        }
    }
}
public class StaticNestedClassDemo
{
    public static void main(String[] args)
    {
        // accessing a static nested class
        OuterClass.StaticNestedClass nestedObject =
        new OuterClass.StaticNestedClass();
        nestedObject.display();
    }
}
```

Output:

```
outer_x = 10
outer_private = 30
```

2.8.2 Inner Class

- A non-static class that is created inside a class but outside a method is called member inner class. Inner class can access the private data members & methods of outer class directly. It is used to group classes and interfaces in one place.

```
class OuterClass
{
    ----
    class InnerClass
    {
        ---
        ---
    }
}
```


Program 2.12: Program to demonstrate the implementation of inner and outer class concept.

```
class OuterClass
{
    int x=10;
    class InnerClass
    {
        void disp()
        {
            System.out.println("Inner class " +x); //x is accessed directly
        }
    }
    void show()
    {
        InnerClass I=new InnerClass();      //method of inner class accessed
        in outer class using object of inner class.
        I.disp();
    }
}
class InnerDemo
{
    public static void main(String args[])
    {
        OuterClass obj=new OuterClass();
        obj.show();
    }
}
```

Output:

```
Inner class 10
```

2.8.3 Anonymous Classes

- A class that has no name is known as anonymous inner class. It can be used to override method of class or interface. It can be created in two ways:
  1. Class (may be abstract or concrete).
  2. Interface

Program 2.13: Program to demonstrate the implementation of Anonymous inner class using abstract class.

```
abstract class Person
{
    abstract void eat();
}
```

```
class TestAnonymousInner
{
   public static void main(String args[])
   {
      Person p=new Person()
      {
         void eat()
         {
            System.out.println ("nice fruits");
         }
      };
      p.eat();
   }
}
```

Output:

```
nice fruits
```

Internal working of given code:

```
Person p=new Person()
{
   void eat()
   {
      System.out.println("nice fruits");
   }
};
```

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type.

Internal class generated by the compiler:

```
import java.io.PrintStream;
static class TestAnonymousInner$1 extends Person
{
   TestAnonymousInner$1()
   {

   }
   void eat()
   {
      System.out.println("nice fruits");
   }
}
```


Program 2.14: Anonymous inner class example using interface.

```
interface Eatable
{
   void eat();
}
class TestAnnonymousInner1
{
   public static void main(String args[])
   {
      Eatable e=new Eatable()
      {
         public void eat()
         {
            System.out.println("nice fruits");
         }
      };
      e.eat();
   }
}
```

Output:

```
nice fruits
```

Internal working of given code:

- It performs two main tasks behind this code:

```
Eatable p=new Eatable()
{
   void eat(){System.out.println("nice fruits");}
};
```

1. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Eatable type.

Internal class generated by the compiler:

```
import java.io.PrintStream;
static class TestAnonymousInner1$1 implements Eatable
{
   TestAnonymousInner1$1()
   {

   }
   void eat()
   {
      System.out.println("nice fruits");
   }
}
```

2.8.4 Abstract Class

- A class which is declared using keyword abstract is known as an **abstract class**. It contains abstract and non-abstract methods. It cannot be instantiated. It can contain constructor and static methods. It can contain final methods which will force the subclass not to change the body of the method. It needs to be extended and its method implemented.

Program 2.15: Define an abstract class Shape with abstract method area(). Write a java program to calculate area of Circle and square.

```
abstract class Shape
{
    abstract void area();
}
class Rectangle extends Shape
{
    int l,b;
    Rectangle(int x, int y)
    {
        l=x;
        b=y;
    }
    void area()
    {
        System.out.println("Area of Rectangle is " + l*b);
    }
}
class Square extends Shape
{
    int s;
    Square(int x)
    {
        s=x;
    }
    void area()
    {
        System.out.println("Area of square is " + 4*s);
    }
}
public class Demo
{
    public static void main(String args[])
    {
```



```
        Rectangle R=new Rectangle(5,6);
        R.area();
        Square S=new Square(5);
        S.area();
    }
}
```

Output:

```
Area of Rectangle is 30
Area of square is 20
```

2.8.5 Interface

- An interface is a collection of abstract methods & final/static data members.
- Interface looks like a class but it is not actually a class. It is not allowed to create the object of interface. Variables declared in an interface are public, static or final.
- **Syntax:**

```
interface interfaceName
{
    //Data members & abstract methods;
}
```

- Interface is allowed to implement multiple inheritances. The "implements" keyword is used to inherit interface into class. More than one interfaces can also be implemented in one class.
- For inhering an interface into any other interface extends keyword is used.

Program 2.16: Define an interface Shape with abstract method area(). Create a class a class Triangle with appropriate variables inherits it into the square class, implements Shape interface into the both classes. Write a java program to calculate area of Triangle and square.

```
interface Shape
{
    void area();
    float pi=3.14f;
}
class Triangle implements Shape
{
    int h,b;
    Triangle (int b, int h)
    {
        this.b=b;
        this.h=h;
    }
```

```
    public void area()
    {
        float a =0.5f*b*h;
        System.out.println ("Area of Triangle is =  " + a);
    }
}
class Square extends Triangle
{
    int s;
    Square(int b, int h, int s)
    {
        super(b,h);
        this.s=s;
    }
    public void area()
    {
        super.area();
        float a=s*s;
        System.out.println("Area of Square is =  " + a);
    }
}
public class InterfaceSQR
{
    public static void main(String args[])
    {
        Square S=new Square(2,2,2);
        S.area();
    }
}
```

Output:

```
Area of Triangle is =  2.0
Area of Square is =  4.0
```

2.9 DEALING WITH STATIC MEMBERS, FINALIZE() METHOD, NATIVE METHOD

2.9.1 Dealing with Static Members

- The variables and methods which are declared without using any keyword in a class are called instance variables and instance methods of a class respectively.
- The limitation of instance variables and method is wastage of memory space. The compiler creates separate copy of instance variables and methods for each object.
- For static data, compiler creates only one copy and multiple objects share it.

- Initial value of static variable is zero.
- Static data always preserved.
- Static variables are also called class variables because for accessing them class name is used.
- Static method can access only static data.
- We cannot be declared static variables by using static keyword in a static method.

Program 2.17: Write a java program for the implementation of static keyword.

```
class A
{
    static int a,b;
    A(int x,int y)
    {
        a=x;
        b=y;
    }
    static void add()
    {
        int c;
        c=a+b;
        System.out.println("Addition is " + c);
    }
}
public class StatDemo
{
    public static void main(String args[])
    {
        A obj=new A(5,6);
        A.add();
    }
}
```

Output:

```
Addition is 11
```

2.9.2 finalize Method

- It is protected & non-static method of java.lang. It is used to delete java as well as non-java objects in program. It is also called as destructor of java. It is defined in Object class.

  Syntax:

```
protected void finalize()
{
    ----
    ----
}
```

- The finalize method is invoked each time before the object is garbage collected. It is used in cleanup processing.

Program 2.18: Java Program for to calculate addition of first and last digit of a number.

```
class Sum
{
    int a,b,s,n;
    Sum(int x)
    {
        n=x;
        s=0;
    }
    void cal()
    {
        b=n%10;
        while(n>0)
        {
            a=n%10;
            n=n/10;
        }
        s=a+b;
        System.out.println(s);
    }
    protected void finalize()
    {
    }
}
public class Demo
{
    public static void main(String args[])
    {
        Sum obj=new Sum(1234);
        obj.cal();
    }
}
```

Output:

```
5
```

2.9.3 Native Method

- A native method is a Java method (either an instance method or a class method) whose implementation is also written in another programming language such as C/C++. Moreover, a method marked as native cannot have a body and should end with a semicolon:

```
[public | protected | private] native [return_type] method ();
```


- The native keyword is applied to a method to indicate that the method is implemented in native code using JNI (Java Native Interface), native is a modifier applicable only for methods and we can't apply it anywhere else.
- The main objectives of native keyword are:
  - To improve performance of the system.
  - To achieve machine level/memory level communication.
  - To use already existing legacy non-java code.

Pseudo code to use native keyword in java:

```
class Native
{
    static
    {
        System.LoadLibrary("Native library path");
    }
    public native void m();
}
class Test
{
    public static void main(String[] args)
    {
        Native n = new Native();
        n.m();
    }
}
```

Important points about native keyword:

- For native methods implementation is already available in old languages like C, C++ and we are not responsible to provide implementation. Hence native method declaration should end with ; (semi-colon).
- We can't declare native method as **abstract.**
- The main advantage of native keyword is improvement in performance but the main disadvantage of native keyword is that it breaks platform independent nature of java.

Writing native methods for Java programs is a multi-step process.

1. Begin by writing the Java program. Create a Java class that declares the native method; this class contains the declaration or signature for the native method. It also includes a main method which calls the native method.
2. Compile the Java class that declares the native method and the main method.
3. Generate a header file for the native method using javah with the native interface flag -jni. Once you've generated the header file you have the formal signature for your native method.
4. Write the implementation of the native method in the programming language of your choice, such as C or C++.
5. Compile the header and implementation files into a shared library file.
6. Run the Java program.

2.10 USE OF "THIS" REFERENCE

- There can be a lot of usage of java **this** keyword. In java, **this** is a reference variable that refers to the current object.

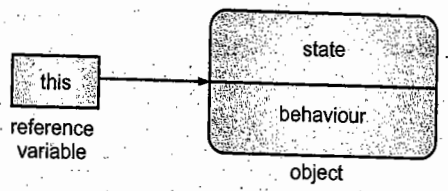


Fig. 2.7: Structure of this keyword

Usage of Java this keyword:

- Here is given the 6 usage of Java this keyword.
  1. this can be used to refer current class instance variable.
  2. this can be used to invoke current class method (implicitly).
  3. this() can be used to invoke current class constructor.
  4. this can be passed as an argument in the method call.
  5. this can be passed as argument in the constructor call.
  6. this can be used to return the current class instance from the method.

Program 2.19: Write a java program for the implementation of this keyword.

```
class Person
{
   String pname;
   int age;
   Person(String pname, int age)
   {
      this.pname=pname;
      this.age=age;
   }
   void disp()
   {
      System.out.println ("pname = " + pname + " " + "Age is = " + age );
   }
}
public class MPerson
{
   public static void main(String args[])
   {
      Person pobj=new Person("DPU",25);
      pobj.disp();
   }
}
```

Output:

```
pname = DPU  Age is = 25
```


2.11 DESIGN OF ACCESSORS AND MUTATOR METHODS

Mutator:

- This method allows the change in the value of class variable i.e. from outside the class data can be manipulated.

Accessors:

- This method allows just to access the data. It does not allow manipulating the data.
- Both methods are used to hide the data of object as much as , So these methods can prevent illegal access to these objects.

Example:

```
Class Number
{
   int x;
   int get()    //Acessor
   {
      Return x;
   }
   int set(int y)      //Mutator
   {
      x=y;
   }
}
```

2.12 CLONING OBJECTS, SHALLOW AND DEEP CLONING

2.12.1 Cloning Objects

- The object cloning is a way to create an **exact copy of an object.** For this purpose, the **clone ()** method of an object class is used to clone an object. The **Cloneable** interface must be implemented by a class whose object clone to create. If we do not implement Cloneable interface, clone() method generates **CloneNotSupportedException.**
- The clone() method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed, so we can use object cloning.

Syntax: `protected Object clone () throws CloneNotSupportedException`

Program 2.20: Write a Java program for the implantation of Cloneable interface.

```
public class EmployeeTest implements Cloneable
{
   int id;
   String name = "";
```

```
    EmployeeTest (int id, String name)
    {
        this.id = id;
        this.name = name;
    }
    public EmployeeTest clone() throws CloneNotSupportedException
    {
        return (EmployeeTest)super.clone();
    }
    public static void main(String[] args)
    {
        EmployeeTest emp = new EmployeeTest (115, "Raja");
        System.out.println (emp.name);
        try
        {
            EmployeeTest emp1 = emp.clone();
            System.out.println (emp1.name);
        }
        catch(CloneNotSupportedException cnse)
        {
            cnse.printStackTrace ();
        }
    }
}
```

Output:

```
Raja
Raja
```

Advantage:

- Java keeps references of objects. If two references are pointing to same object then modification to one reference will reflect in another reference as well. To prevent such situation, cloning is required. In cloning, a copy of object is to be created and used to that both objects can be modified independently. This is one of the major advantage of cloning.

2.12.2 Shallow and Deep Cloning

- The behavior of the Object.clone() method classifies cloning into two sections.

1. Shallow Cloning:

- This is the default cloning strategy provided by Object.clone(). The clone() method of the Object class creates a new instance and copies all fields of the Cloneable object to that new instance (either it is primitive or a reference). So in the case of reference types, only reference bits get copied to the new instance. Therefore, the reference variable of both objects will point to the same object.

Example: To demonstrate Shallow cloning see following Program 2.21, we will create two classes - Person and City. Both classes have:

1. toString() method is used to show the content of the person object.
2. equals() and hashCode() methods used to compare the objects.
3. clone() method is used to clone the object.

And as we can see, toString(), equals(), and hashCode() have the @Override annotation, while the clone() method doesn't have it because we are not overriding it from clone — we are creating a new method by the name of clone(), and we can name it anything else.

Program 2.21: Write a java program for the implementation of shallow interface.

```
class Person implements Cloneable
{
    private String name;
    private int income;
    private City city;
    public String getName()
    {
        return name;
    }
    public void setName(String firstName)
    {
        this.name = firstName;
    }
    public int getIncome()
    {
        return income;
    }
    public void setIncome(int income)
    {
        this.income = income;
    }
    public City getCity()
    {
        return city;
    }
    public void setCity(City city)
    {
        this.city = city;
    }
```

```
public Person(String firstName, int income, City city)
{
   super();
   this.name = firstName;
   this.income = income;
   this.city = city;
}
@Override
public Person clone() throws CloneNotSupportedException
{
   return (Person) super.clone();
}
@Override
public String toString()
{
   return "Person [name=" + name + ", income=" + income + ", city=" +
   city + "]";
}
@Override
public int hashCode()
{
   final int prime = 31;
   int result = 1;
   result = prime * result + ((city == null) ? 0: city.hashCode());
   result = prime * result + income;
   result = prime * result + ((name == null) ? 0: name.hashCode());
   return result;
}
@Override
public boolean equals(Object obj)
{
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   Person other = (Person) obj;
   if (city == null)
   {
      if (other.city != null)
      return false;
   }
```



```
      else if (!city.equals(other.city))
         return false;
      if (income != other.income)
         return false;
      if (name == null)
      {
         if (other.name != null)
         return false;
      }
      else if (!name.equals(other.name))
         return false;
         return true;
   }
}
class City implements Cloneable
{
   private String name;
   public String getName()
   {
      return name;
   }
   public void setName(String name)
   {
      this.name = name;
   }
   public City(String name)
   {
      super();
      this.name = name;
   }
   public City clone() throws CloneNotSupportedException
   {
      return (City) super.clone();
   }
   @Override
   public String toString()
   {
      return "City [name=" + name + "]";
   }
   @Override
```

```
    public int hashCode()
    {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0: name.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj)
    {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        City other = (City) obj;
        if (name == null)
        {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
                return false;
         return true;
    }
}
public class CloningExample
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        City city = new City("Dehradun");
        Person person1 = new Person("Naresh", 10000, city);
        System.out.println(person1);
        Person person2 = person1.clone();
        System.out.println(person2);
        if (person1 == person2)
        {
            System.out.println("Both person1 and person2 holds same object");
        }
        if (person1.equals(person2))
        {
```



```
            System.out.println("But both person1 and person2 are equal and
            have same content");
        }
        if (person1.getCity() == person2.getCity())
        {
            System.out.println("Both person1 and person2 have same city object");
        }
    }
}
```

Output:

```
Person [name=Naresh, income=10000, city=City [name=Dehradun]]
Person [name=Naresh, income=10000, city=City [name=Dehradun]]
But both person1 and person2 are equal and have same content
Both person1 and person2 have same city object
```

Explanation:

person1.clone () calls super.clone (), which means the Object.clone () method.

- The Object.clone() method copies the content of the object to another object bit-by-bit, meaning the values of all instance variables from one object will get copied to instance variables of other objects.
- So (person1 == person2) will evaluate false because person1 and person2 are copies of each other, but both are different objects and hold different spots in heap memory. Meanwhile, person1.equals (person2) evaluates true because both have the same content.
- But as we know, reference variables hold the address of the object instead of object itself, which can also be referred from other reference variables. And if we change one, the other will reflect that change.
- So Object.clone() will copy the address that person1.city is holding to person2.city, so now city, person1.city, and person2.city all are holding the same city object. That's why (person1.getCity() == person2.getCity()) evaluates true. This behavior of cloning is known as **Shallow Cloning.**

2. Deep Cloning:

- As the name suggests, deep cloning means copying everything from one object to another object. To achieve this, we will need to trick our clone() method provides our own cloning strategy. We can do it by implementing a Cloneable interface and overriding the clone() method in every reference type we have in our object hierarchy. Then, we call super.clone() and these clone() methods in our object's clone method.
- So we can change the clone method of the Person class in the following way:

```
public Person clone() throws CloneNotSupportedException
{
    Person clonedObj = (Person) super.clone();
    clonedObj.city = this.city.clone();
    return clonedObj;
}
```

- Now (person1.getCity() == person2.getCity()) will evaluate false because, in the clone() method of the Person class, we are cloning the city object and assigning it to the new cloned person object.
- In the example below, we have deep-copied the city object by implementing clone() in the City class and calling that clone() method of a person class, That's why person1.getCity() == person2.getCity() evaluates false – because both are separate objects. But we have not done the same with the Country class, person1.getCountry() == person2.getCountry() evaluates true.

Program 2.22: Write a java program for the implementation of deep cloning.

```
public class CloningExample
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        City city = new City("Dehradun");
        Country country = new Country("India");
        Person person1 = new Person("Naresh", 10000, city, country);
        System.out.println(person1);
        Person person2 = person1.clone();
        System.out.println(person2);
        if (person1 == person2)
        {
            System.out.println ("Both person1 and person2 holds same object");
        }
        if (person1.equals(person2))
        {
            System.out.println("But both person1 and person2 are equal and
            have same content");
        }
        if (person1.getCity() == person2.getCity())
        {
            System.out.println("Both person1 and person2 have same city
            object");
        }
        if (person1.getCountry() == person2.getCountry())
        {
            System.out.println("Both person1 and person2 have same country
            object");
        }
        city.setName("Pune");
        country.setName("IN");
        System.out.println(person1);
        System.out.println(person2);
    }
}
```



```
class Person implements Cloneable
{
    private String name;
    private int income;
    private City city;
    private Country country;
    public String getName() {
        return name;
    }
    public void setName(String firstName) {
        this.name = firstName;
    }
    public int getIncome() {
        return income;
    }
    public void setIncome(int income) {
        this.income = income;
    }
    public City getCity() {
        return city;
    }
    public void setCity(City city) {
        this.city = city;
    }
    public Country getCountry() {
        return country;
    }
    public void setCountry(Country country) {
        this.country = country;
    }
    public Person(String name, int income, City city, Country country) {
        super();
        this.name = name;
        this.income = income;
        this.city = city;
        this.country = country;
    }
    public Person clone() throws CloneNotSupportedException {
        Person clonedObj = (Person) super.clone();
        clonedObj.city = this.city.clone();
        return clonedObj;
    }
    @Override
```

```
public String toString()
{
   return "Person [name=" + name + ", income=" + income + ", city=" +
   city + ", country=" + country + "]";
}
@Override
public int hashCode()
{
   final int prime = 31;
   int result = 1;
   result = prime * result + ((city == null) ? 0: city.hashCode());
   result = prime * result + ((country == null) ? 0: country.hashCode());
   result = prime * result + income;
   result = prime * result + ((name == null) ? 0: name.hashCode());
   return result;
}
@Override
public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Person other = (Person) obj;
    if (city == null)
    {
        if (other.city != null)
            return false;
    }
    else if (!city.equals(other.city))
        return false;
    if (country == null)
    {
        if (other.country != null)
            return false;
    }
    else if (!country.equals(other.country))
        return false;
    if (income != other.income)
        return false;
```



```
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}
class City implements Cloneable {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public City(String name) {
        super();
        this.name = name;
    }
    public City clone() throws CloneNotSupportedException {
        return (City) super.clone();
    }
    @Override
    public String toString() {
        return "City [name=" + name + "]";
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0: name.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj)
    {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
```

```
        if (getClass() != obj.getClass())
            return false;
        City other = (City) obj;
        if (name == null)
        {
            if (other.name != null)
                return false;
        }
        else if (!name.equals(other.name))
            return false;
        return true;
    }
}
class Country
{
    private String name;
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public Country(String name)
    {
        super();
        this.name = name;
    }
    @Override
    public String toString()
    {
        return "Country [name=" + name + "]";
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0: name.hashCode());
        return result;
    }
    @Override
```



```
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Country other = (Country) obj;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}
```

Output:

```
Person [name=Naresh, income=10000, city=City [name=Dehradun], country=Country
[name=India]]
Person [name=Naresh, income=10000, city=City [name=Dehradun], country=Country
[name=India]]
But both person1 and person2 are equal and have same content
Both person1 and person2 have same country object
Person [name=Naresh, income=10000, city=City [name=Pune], country=Country
[name=IN]]
Person [name=Naresh, income=10000, city=City [name=Dehradun], country=Country
[name=IN]]
```

2.13 GENERIC CLASS TYPES

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- The type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Syntax:

```
public class Box<T>
{
   private T t;
}
```

Where,

- **Box** – Box is a generic class.
- **T** – The generic type parameter passed to generic class. It can take any Object.
- **t** – Instance of generic type T.

Description:

- The T is a type parameter passed to the generic class Box and should be passed when a Box object is created.

Program 2.23: Write a java program for the implementation Generic classes.

```
GenericsTester.java
public class GenericsTester
{
   public static void main(String[] args)
   {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add (new Integer (10));
      stringBox.add (new String("Hello World"));

      System.out.printf ("Integer Value:%d\n", integerBox.get());
      System.out.printf ("String Value:%s\n", stringBox.get());
   }
}
class Box<T>
{
   private T t;
   public void add(T t)
   {
      this.t = t;
   }
   public T get()
   {
      return t;
   }
}
```

Output:

```
Integer Value:10
String Value:Hello World
```


- By convention, type parameter names are named as single, uppercase letters so that a type parameter can be distinguished easily with an ordinary class or interface name.
- Following is the list of commonly used type parameter names:
  - **E** – Element, and is mainly used by Java Collections framework.
  - **K** – Key, and is mainly used to represent parameter type of key of a map.
  - **V** – Value, and is mainly used to represent parameter type of value of a map.
  - **N** – Number, and is mainly used to represent numbers.
  - **T** – Type, and is mainly used to represent first generic type parameter.
  - **S** – Type, and is mainly used to represent second generic type parameter.
  - **U** – Type, and is mainly used to represent third generic type parameter.
  - **V** – Type, and is mainly used to represent fourth generic type parameter.

Program 2.24: Write a java program to create generic classes for Hashtable, ArrayList and Map interface.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class GenericsTester
{
   public static void main(String[] args)
   {
      Box<Integer, String> box = new Box<Integer, String>();
      box.add(Integer.valueOf(10),"Hello World");
      System.out.printf("Integer Value:%d\n", box.getFirst());
      System.out.printf("String Value:%s\n", box.getSecond());

      Pair<String, Integer> pair = new Pair<String, Integer>();
      pair.addKeyValue("1", Integer.valueOf(10));
      System.out.printf("(Pair)Integer Value:%d\n", pair.getValue("1"));

      CustomList<Box> list = new CustomList<Box>();
      list.addItem(box);
      System.out.printf("(CustomList)Integer Value:%d\n",
       list.getItem(0).getFirst());
   }
}
class Box<T, S>
{
   private T t;
   private S s;
```

```
    public void add(T t, S s)
    {
       this.t = t;
       this.s = s;
    }
    public T getFirst()
    {
       return t;
    }
    public S getSecond()
    {
       return s;
    }
}
class Pair<K,V>
{
    private Map<K,V> map = new HashMap<K,V>();
    public void addKeyValue(K key, V value)
    {
       map.put(key, value);
    }
    public V getValue(K key)
    {
       return map.get(key);
    }
}
class CustomList<E>
{
    private List<E> list = new ArrayList<E>();
    public void addItem(E value)
    {
       list.add(value);
    }
    public E getItem(int index)
    {
       return list.get(index);
    }
}
```

Output:

```
Integer Value:10
String Value:Hello World
(Pair)Integer Value:10
(CustomList)Integer Value:10
```


Summary

- Java is object oriented programming language, it support to the basic features of oops such as class, object, abstraction, encapsulation, polymorphism and inheritance.
- Class is a collection of data member and member functions. The variables which are declared in a class are called instance variables because for accessing them object of class is created.
- In java, object can be created by using new operator and used to access data members and member functions of a class.
- In java there is no pointer because there is reference variable, it works as pointer used to store reference of an object of same type.
- In a class, we can defined a function, function is a self contained block of statements are used to perform a specific task.
- A function which calls to itself is called recursion.
- Constructor is a special member function because its name and class name is same.
- Java supports to the parameterized and default constructor, for the implementation of copy constructor cloning is used.
- The limitation of instance variable is wastage of memory space to overcome this static keyword is used.
- Polymorphism is implemented in two ways: Method overloading and method overriding.
- In a same class, same function name with different parameters such a concept is called method overloading.
- If classes are different and there is inheritance between them method names along with parameter list is same then such a concept is method overriding.
- Inheritance is a mechanism in which super class is inherited in sub class.
- The reusability is characteristic of inheritance.
- For accessing overridden data member and member functions of a class super keyword is used.
- We can call to a function in two ways: call by value and call by reference.
- Data member and member functions of outer class directly access in inner class but vice versa is not true.
- A function which does not have name and can be created by using lambda function. Such a function is called anonymous function.

Check Your Understanding

1. Which of the following option leads to the portability and security of Java?
   (a) Bytecode is executed by JVM
   (b) The applet makes the Java code secure and portable
   (c) Use of exception handling
   (d) Dynamic binding between objects
2. In which process, a local variable has the same name as one of the instance variables?
   (a) Serialization (b) Variable Shadowing
   (c) Abstraction (d) Multi-threading
3. Which of the following is true about the anonymous inner class?
   (a) It has only methods (b) Objects can't be created
   (c) It has a fixed class name (d) It has no class name
4. What do you mean by nameless objects?
   (a) An object created by using the new keyword.
   (b) An object of a superclass created in the subclass.
   (c) An object without having any name but having a reference.
   (d) An object that has no reference.
5. An interface with no fields or methods is known as a ____.
   (a) Runnable Interface
   (b) Marker Interface
   (c) Abstract Interface
   (d) CharSequence Interface
6. Which option is false about the *final* keyword?
   (a) A final method cannot be overridden in its subclasses.
   (b) A final class cannot be extended.
   (c) A final class cannot extend other classes.
   (d) A final method can be inherited.
7. In which memory a String is stored, when we create a string using new operator?
   (a) Stack (b) String memory
   (c) Heap memory (d) Random storage space
8. What is meant by the classes and objects that dependents on each other?
   (a) Tight Coupling (b) Cohesion
   (c) Loose Coupling (d) None of the above

9. Does constructor overloading include different return types for constructors to be overloaded?
   (a) Yes, if return types are different, signature becomes different
   (b) Yes, because return types can differentiate two functions
   (c) No, return type can't differentiate two functions
   (d) No, constructors doesn't have any return type
10. Which among the following function can be used to call default constructor implicitly in java?
    (a) this() (b) that()
    (c) super() (d) sub()

Answers

1. (a)	2. (b)	3. (d)	4. (d)	5. (b)	6. (a)	7. (c)	8. (a)	9. (d)	10. (a)

Practice Questions

Q.I Answer the following Questions in short:

1. What is Native method? Give its use.
2. What is used of Accessors method?
3. What do you mean by Object reference?
4. What is object? How it differ from reference variable?
5. What is Abstract class? How it is differ from concrete class.
6. What is Interface? How it is differ from abstract class?
7. What is Cloning? Give its types.
8. What is Inner class? How to access the data of inner class in outer class.
9. What is finalize () method? How to call it?
10. What is Recursion? How it is differ from method?

Q.II Answer the following Questions:

1. Explain different types of Generic class parameter in details.
2. Explain object cloning in details.
3. Differentiate between deep cloning and shallow cloning.
4. Explain different types of Accessors and Mutator methods.
5. What is used of this keyword? Explain with an example.
6. Differentiate between constructor and member function.
7. Explain method overloading with an example.
8. Write a java program to accept the details product (PCode, Pname, Rate) and display it.(Use constructor)
9. Can you overload to the constructor? Explain.
10. Differentiate between instance variable and class variable.

Q.III Write short note on:

1. Garbage collection
2. Access Control
3. Anonymous Class
4. Abstract Class
5. Method overloading
6. Generic class type
7. Cloning object

■■■

3... Extending Classes and Inheritance

Objectives...

- To understand uses and benefits of inheritance.
- To study the types of Inheritance.
- To study how to inheriting data members and methods.
- To study Role of Constructors in inheritance.
- To understand Polymorphism in inheritance.
- To study Type Compatibility and Conversion.
- To learn how multiple inheritance is implemented by using interface.

3.1 INTRODUCTION

- Inheritance is one of the most important feature of java.
- The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called **Inheritance.**
- **Child Class:** The class that extends the features of another class is known as Child class, Sub class or Derived class.
- **Parent Class:** The class whose properties and functionalities are used (inherited) by another class is known as Parent class, Super class or Base class.
- Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object. The idea behind inheritance is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Java does not support to the multiple inheritance.
- Reusability is a characteristic of inheritance.

3.2 USE AND BENEFITS OF INHERITANCE IN OOP

Use of Inheritance in OOP:

- The use of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.
- Primarily we use inheritance, so that, we can use specific portions of code and can modify certain features according to our need and this can be done without any complexity. Inheritance provides flexibility in our code to reuse it from base class to required derived class. A child class can override properties of base class without rewriting code in the same class again and again.
- To reuse code, write code and apply it further, wherever necessary.
- To avoid duplicity and data redundancy in the program.
- To reduce space and time complexity.
- Easier in hierarchal programming paradigm.
- Variables of the same name can be used multiple times in the scope of the code.
- To create dominant data objects and functions.

Benefits of Inheritance in OOP:

- A code can be used again and again.
- Inheritance in Java enhances the properties of the class, which means that property of the parent class will automatically be inherited by the base class.
- It can define more specialized classes by adding new details.
- It helps for better implementation of an application i.e. it increases execution speed of a program.
- It implements is-a relationship.
- One superclass can be used for the number of subclasses in a hierarchy.
- No changes to be done in all base classes, just do changes in parent class only.
- Inheritance is used to generate more dominant objects.
- Inheritance avoids duplicity and data redundancy.
- Inheritance is used to avoid space complexity and time complexity.
- Inheritance is implemented by using extends keyword in java.

Syntax:

```
class Child-class-name extends Parent-class-name
{
   //methods and fields
}
```


Program 3.1: Write a Java program for the implementation of Inheritance.

```
class Teacher
{
   String designation ="Teacher";
   String collegeName="DPU";
   void does()
   {
      System.out.println ("Teaching");
   }
}
class JavaTeacher extends Teacher
{
   StringmainSubject="Computer";
   public static void main(String args[])
   {
      JavaTeacher obj=new JavaTeacher ();
      System.out.println (obj.collegeName);
      System.out.println (obj.designation);
      System.out.println (obj.mainSubject);
      obj.does ();
   }
}
```

Output:

```
DPU
Teacher
Computer
Teaching
```

- In this example, class Teacher is inherited in class JavaTeacher so we can access all the data member and member functions of class Teacher in class JavaTeacher. For accessing data member and member functions of both the classes in main class need to create an object of derived class i.e. of class JavaTeacher.

3.3 TYPES OF INHERITANCE

- Java supports to the different types of inheritance:
  1. Single Inheritance
  2. Multiple Inheritance (Through Interface)
  3. Multilevel Inheritance
  4. Hierarchical Inheritance
  5. Hybrid Inheritance (Through Interface)

- Let's see about each one of them one by one.

1. Single Inheritance:

- Single inheritance is the simple inheritance, when a class extends another class (Only one class) then we call it as **Single inheritance.**
- The below diagram represents the single inheritance in java where *Class B* extends only one class *Class A*. Here *Class B* will be the Sub class and *Class A* will be one and only super class.

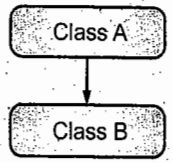


Fig. 3.1: Single Inheritance

Program 3.2: Java program to demonstrate Single level inheritance.

```
class ClassA
{
   public void dispA()
   {
      System.out.println ("disp() method of ClassA");
   }
}
class ClassB extends ClassA
{
   public void dispB()
   {
      System.out.println("disp() method of ClassB");
   }
   public static void main(String args[])
   {
      ClassB b = new ClassB ();
      b.dispA();
      b.dispB();
   }
}
```

Output:

```
disp() method of ClassA
disp() method of ClassB
```

2. Multiple Inheritance:

- Multiple inheritance is nothing but one class extending more than one class.
- Multiple inheritance is basically not supported by many Object Oriented Programming languages such as Java, Small Talk, C# etc. (C++ Supports Multiple Inheritance).

- As the Child class has to manage the dependency of more than one Parent class. But you can achieve multiple inheritance in Java using Interface.

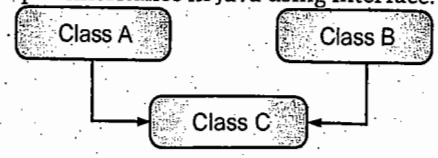


Fig. 3.2: Multiple Inheritance

3. Multilevel Inheritance:

- In Multilevel inheritance a derived class will be inheriting a parent class and as well as the derived class act as the parent class to other class.
- As seen in the below diagram. *ClassB* inherits the property of *ClassA* and again *ClassB* act as a parent for *ClassC*. In Short *ClassA* parent for *ClassB* and *ClassB* parent for *ClassC*.

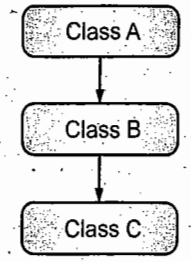


Fig. 3.3: Multilevel Inheritance

Program 3.3: Java program to demonstrate Multilevel Inheritance.

```
class ClassA
{
   public void dispA()
   {
      System.out.println("disp() method of ClassA");
   }
}
class ClassB extends ClassA
{
   public void dispB()
   {
      System.out.println("disp() method of ClassB");
   }
}
class ClassC extends ClassB
{
   public void dispC()
   {
      System.out.println("disp() method of ClassC");
   }
```

```
    public static void main(String args[])
    {
        classC c = new ClassC();
        c.dispA();   //call dispB() method of ClassB
        c.dispB();   //call dispC() method of ClassC
        c.dispC();
    }
}
```

Output:

```
disp() method of ClassA
disp() method of ClassB
disp() method of ClassC
```

4. Hierarchical Inheritance:

- In Hierarchical inheritance, one parent class will be inherited by many sub classes.
- As per the below example, *ClassA* will be inherited by *ClassB*, *ClassC* and *ClassD*. *ClassA* will be acting as a parent class for *ClassB* ,*ClassC* and *ClassD*.

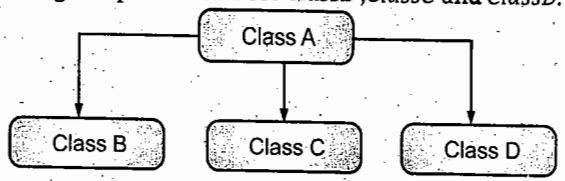


Fig. 3.4: Hierarchical Inheritance

Program 3.4: Java program to demonstrate Hierarchical Inheritance

```
class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}
class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
}
class ClassC extends ClassA
{
    public void dispC()
    {
        System.out.println("disp() method of ClassC");
    }
}
```



```
class ClassD extends ClassA
{
    public void dispD()
    {
        System.out.println("disp() method of ClassD");
    }
}
public class InheritanceTest
{
    public static void main(String args[])
    {
        //Assigning ClassB object to ClassB reference
        ClassB b = new ClassB();
              //call dispB() method of ClassB
        b.dispB();
              //call dispA() method of ClassA
        b.dispA();
              //Assigning ClassC object to ClassC reference
        ClassC c = new ClassC();
              //call dispC() method of ClassC
        c.dispC();
              //call dispA() method of ClassA
        c.dispA();
              //Assigning ClassD object to ClassD reference
        ClassD d = new ClassD();
              //call dispD() method of ClassD
        d.dispD();
              //call dispA() method of ClassA
        d.dispA();
        }
}
```

Output :

```
disp() method of ClassB
disp() method of ClassA
disp() method of ClassC
disp() method of ClassA
disp() method of ClassD
disp() method of ClassA
```

5. Hybrid Inheritance:

- Hybrid Inheritance is the combination of both Single and Multiple Inheritance.
- Hybrid inheritance is also not directly supported in Java only through interface we can achieve this.

- Flow diagram of the Hybrid inheritance will look like below. As you can see ClassA will be acting as the Parent class for ClassB & ClassC and ClassB & ClassC will be acting as Parent for ClassD.

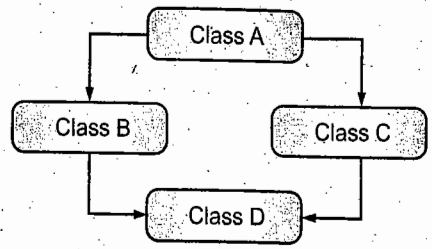


Fig. 3.5: Hybrid Inheritance

3.4 INHERITING DATA MEMBERS AND METHODS

- The derived class inherits all the members and methods that are declared as public or protected.
- If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

Program 3.5: Write a java program to demonstrate the scope of variables with their access specifiers.

```
class Teacher
{
   private String designation ="Teacher";
   private String collegeName="DYPACS";
   public String getDesignation ()
   {
      return designation;
   }
   protected void setDesignation(String designation)
   {
      this.designation= designation;
   }
   protected String getCollegeName()
   {
      return collegeName;
   }
   protected void setCollegeName (String collegeName)
   {
      this.collegeName=collegeName;
   }
```



```
   void does()
   {
      System.out.println ("Teaching");
   }
}
public class JavaExample extendsTeacher
{
   StringmainSubject="Java";
   public static void main(String args[])
   {
      JavaExample obj=new JavaExample();
      System.out.println (obj.getCollegeName ());
      System.out.println (obj.getDesignation ());
      System.out.println (obj.mainSubject);
      obj.does();
   }
}
```

Output:

```
DYPACS
Teacher
Java
Teaching
```

- The important point to note in the above example is that the child class is able to access the private members of parent class through **protected methods** of parent class.
- When we make an instance variable (data member) or method **protected**, this means that they are accessible only in the class itself and in child class.

3.5 ROLE OF CONSTRUCTORS IN INHERITANCE

- Constructor is a block of code that allows you to create an object of class and has same name as class with no explicit return type.
- Whenever a class (child class) extends another class (parent class), the sub class inherits state and behavior in the form of variables and methods from its super class but it does not inherit constructor of super class because of following reason:
  - Constructors are special and have same name as class name. So if constructors were inherited in child class then child class would contain a parent class constructor which is against the constraint that constructor should have same name as class name.

For example:

```
class Parent
{
    Parent()
    {
    }
    public void print()
    {
    }
}
class Child extends Parent
{
    Parent()
    {
    }
    void print()
    {
    }
    public static void main(String[] args)
    {
    Child c1 = new Child(); // allowed
    Child c2 = new Parent(); // not allowed
    }
}
```

- If we define Parent class constructor inside Child class it will give compile time error for return type and consider it a method. But for print method it does not give any compile time error and consider it an overriding method.
- Now suppose if constructors can be inherited then it will be impossible to achieving encapsulation. Because by using a super class's constructor we can access/initialize private members of a class.
- A constructor cannot be called as a Method. It is called when object of the class is created so it does not make sense of creating child class object using parent class constructor notation. i.e. Child c = new Parent();
- A parent class constructor is not inherited in child class and this is why super() is added automatically in child class constructor, if there is no explicit call to super or this.

- Subclasses inherit all the private instance variables in a superclass that they extend, but they cannot directly access them since they are private. And constructors are not inherited.

- How do you initialize inherited private variables if you don't have direct access to them in the subclass? For accessing them you can put a call to the parent constructor using as the first line in a subclass constructor. The superclass constructor can be called from the first line of a subclass constructor by using the keyword super and passing appropriate parameters.
- For example **super(); or super(theName);** as in the code below. The actual parameters given to super() are used to initialize the inherited instance variables, for example the name instance variable in the Person superclass.

Example:

```
class Employee extends Person
{
    Employee()
    {
        super();        // calls the Person() constructor
    }
    Employee (StringtheName)
    {
        super (theName); // calls Person(theName) constructor
    }
}
```

- The super (theName) in the Employee constructor will call the constructor that takes a String object in the Person class to set the name.
- Try creating another Employee object in the main method that passes in your name and then use the get methods to print it out.

Program 3.6: Write a Java program to implement how to call constructor of super class into the constructor of subclass.

```
class Person
{
    private String name;
    public Person(String theName)
    {
        this.name = theName;
    }
    public String getName()
    {
        return name;
    }
}
public class Employee extends Person
{
    private static int nextId = 1;
    private int id;
```

```
    public Employee(String theName)
    {
        super(theName);
        id = nextId;
        nextId++;
    }
    public int getId()
    {
        return id;
    }
    public static void main(String[] args)
    {
        Employee emp = new Employee ("DPU");
        System.out.println ("Emp Name is" + emp.getName ());
        System.out.println ("Emp ID is" + emp.getId ());
    }
}
```

Output:

```
Emp Name is DPU
Emp ID is 1
```

- If a class has no constructor in Java, the compiler will add a no-argument constructor. A no-argument constructor is one that doesn't have any parameters, for example public Person().
- If a subclass has no call to a superclass constructor using super as the first line in a subclass constructor then the compiler will automatically add a super() call as the first line in a constructor. So, be sure to provide no-argument constructors in parent classes or be sure to use an explicit call to super() as the first line in the constructors of subclasses.
- Regardless of whether the superclass constructor is called implicitly or explicitly, the process of calling superclass constructors continues until the Object constructor is called since every class inherits from the Object class.

3.6 OVERRIDING SUPER CLASS METHODS, USE OF SUPER

3.6.1 Overriding super class methods

- We talked about super classes and sub classes. If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.
- The benefit of overriding is: ability to define a behavior that's specific to the sub class type, which means a sub class, can implement a parent class method based on its requirement.

- In object-oriented terms, overriding means to override the functionality of an existing method.
- In method overriding, method of derived class hides to the method of its base class so we cannot access overridden method of base class by using an object of derived class.

Program 3.7: Java program to demonstrate method overriding.

```
class A
{
    void disp()
    {
        System.out.println("We are DYPIAN");
    }
}
class B extends A
{
    void disp()
    {
        System.out.println("Basically we are from DYPACS");
    }
}
Class MethOverDemo
{
    public static void main (String args[])
    {
        B obj=new B();
        obj.disp();  //It will call disp() method of class B
    }
}
```

Output:

```
Basically we are from DYPACS
```

- The disp() method of class B hides to the disp() method of class A, so we cannot access disp() method of class A by using an object of class B.

3.6.2 Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the super class method is declared public then the overriding method in the subclass cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.

- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

3.6.3 Use of Super Keyword

- For accessing overridden method of class A in class B, we have to use *super* keyword.

Program 3.8: Java program to demonstrate method overriding with super keyword.

```
class A
{
    void disp()
    {
        System.out.println ("We are DYPIAN");
    }
}
class B extends A
{
    void disp()
    {
        super.disp();  // Calling to the disp() method of class A
        System.out.println ("Basically we are from DYPACS");
    }
}
public class MethOverDemo
{
    public static void main (String args[])
    {
        B obj=new B();
        obj.disp();  //It will call disp() method of class B
    }
}
```

Output:

```
We are DYPIAN
Basically we are from DYPACS
```


3.7 POLYMORPHISM IN INHERITANCE

- Polymorphism means one object many forms. We can implement it in two ways:
  (a) Compile Time (Method overloading)
  (b) Run Time (Method overriding).
- Polymorphism in Java occurs when there are one or more classes or objects related to each other by inheritance.
- In other words, it is the ability of an object to take many forms. Inheritance lets users inherit attributes and methods, and polymorphism uses these methods for performing different tasks. So, the goal is communication but the approach is different.
- For example, you have a Smartphone for communication. The communication mode you choose could be anything. It can be a call, a text message, a picture message, mail, etc. So, the goal is common that is communication, but their approach is different. This is called **Polymorphism.**
- For implementing polymorphism in inheritance method overriding is used.

Program 3.9: Write a java program to calculate area of circle and triangle.(Use Method Overriding).

```
import java.util.*;
class Triangle
{
    float a,b,h;
    Triangle(float b,float h)
    {
        this.b=b;
        this.h=h;
    }
    void area()
    {
        a=0.5f*b*h;
        System.out.println ("Area Of Triangle Is " + a);
    }
}
class Circle extends Triangle
{
    float r;
    Circle(float b, float h, float x)
    {
        super(b,h);
        r=x;
    }
```

```
    void area()
    {
        super.area();
        a=3.14f*r*r;
        System.out.println("Area Of Circle Is " + a);
    }
}
class MethOver
{
    public static void main(String args[])
    {
        float b,h,x,r;
        Scanner ob=new Scanner (System.in);
        System.out.println ("Enter the values for B, H and R");
        b=ob.nextFloat ();
        h=ob.nextFloat ();
        r=ob.nextFloat ();
        Circle obj=new Circle (5,6,7);
        obj.area();
    }
}
```

Output:

```
Enter the values for B, H and R
    5
    6
    7
Area Of Triangle Is 15.0
Area Of Circle Is 153.86002
```

3.8 TYPE COMPATIBILITY AND CONVERSION

- When you assign value of one data type to another, the two types might not be compatible with each other.
- If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly.
- For example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

- Widening conversion takes place when two data types are automatically converted. This happens when:
  - The two data types are compatible.
  - When we assign value of a smaller data type to a bigger data type.

- For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.

```
byte->short->int->long->float->double
```

Program 3.10: Java Program to demonstrate automatic conversion.

```
public class Test
{
    public static void main(String[] args)
    {
        int i = 100;
        // automatic type conversion
        long l = i;
        // automatic type conversion
        float f = l;
        System.out.println ("Int value "+i);
        System.out.println ("Long value "+l);
        System.out.println ("Float value "+f);
    }
}
```

Output:

```
Int Value 100
Long Value 100
Float value 100.0
```

Narrowing or Explicit Conversion

- If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.
- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

```
double->float->long->int->short-> byte
```

Program 3.11: Java program to illustrate explicit type conversion.

```
public class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;
        //explicit type casting
        long l = (long)d;
        //explicit type casting
        int i = (int)l;
```

```
        System.out.println("Double value "+d);
        //fractional part lost
        System.out.println ("Long value "+l);
        //fractional part lost
        System.out.println ("Int value "+i);
    }
}
```

Output:

```
Double Value 100.04
Long Value 100
Int value 100
```

- While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:
  1. Java automatically promotes each byte, short or char operand to int when evaluating an expression.
  2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

3.9 IMPLEMENTING INTERFACES

- Java does not support to the multiple inheritance directly, for its implementation interface is used.
- Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
  - Interfaces specify what a class must do and not. It is the blueprint of the class.
  - An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move (). So it specifies a set of methods that the class has to implement.
  - If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
  - A Java library example is, **Comparator Interface**. If a class implements this interface, then it can be used to sort a collection.
  - **Syntax for interface:**

```
Interface <interface_name>
{
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```


- To declare an interface, use ***interface*** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface. To implement interface use ***implements*** keyword.

Why do we use interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction.

Program 3.12: Java program to calculate area of circle use interface.

```
interface Shape
{
   void area();
   float PI=3.14f;
}
class Circle implements Shape
{
   floata,r;
   Circle(float x)
   {
      r=x;
   }
   public void area()
   {
      a=PI*r*r;
      System.out.println ("Area Of Circle is: "  + a);
   }
}
public class  InterfaceDemo
{
   public static void main(String args[])
   {
      Circle obj=new Circle(5);
      obj.area();
   }
}
```

Output:

```
Area of Circle is: 78.5
```

Summary

- Inheritance is a mechanism in which one class is inherited from another class i.e. Derived class is inherited from the base class.
- The Reusability is a characteristic of inheritance.
- Inheritance can be implemented by using extends keyword.

Syntax:

```
class MyBase
{
}
class Derived extends MyBase
{
}
```

- Like C++, java supports to the all types of inheritance except the structure of multiple inheritance.
- Multiple inheritance is implemented in java by using Interface.
- Polymorphism means one object many forms. It can be defined in two ways:
  (a) Method Overloading
  (b) Method Overriding
- If classes are different and there is inheritance between them methods names along with parameter list is same then that concept is called Method Overriding.
- For accessing overridden method and constructor of super class in derived class super keyword is used.
- Type casting is process used to convert a value of a variable from one type into another type.
- In java data type get promoted from smaller to larger but vice versa it is not true, In such case type casting is used.
- Interface is called fully abstract class because in this we cannot write definition for any method, all the methods are considered as abstract and all the variables are considered as final or static.

Check Your Understanding

1. Which inheritance is not supported in Java programming?
   (a) Multiple inheritance using classes
   (b) Multiple inheritance using interfaces
   (c) Multilevel inheritance
   (d) Single inheritance.
2. Which class cannot be sub classed?
   (a) final class (b) object class
   (c) abstract class (d) child class

3. What do you mean by nameless objects?
   (a) An object created by using the new keyword.
   (b) An object of a superclass created in the subclass.
   (c) An object without having any name but having a reference.
   (d) An object that has no reference.
4. Java used inheritance ____.
   (a) for code re-usability (b) to achieve runtime polymorphism
   (c) Both of the above (d) None
5. Which of this keyword must be used to inherit a class?
   (a) super (b) this
   (c) extent (d) extends
6. A class member declared protected becomes a member of subclass of which type?
   (a) public member (b) private member
   (c) protected member (d) static member
7. Using which of the following, multiple inheritance in Java can be implemented?
   (a) Interfaces (b) Multithreading
   (c) Protected methods (d) Private methods
8. If super class and sub class have same variable name, which keyword should be used to use super class?
   (a) super (b) this
   (c) upper (d) classname
9. All classes in Java are inherited from which class?
   (a) java.lang.class (b) java.class.inherited
   (c) java.class.object (d) java.lang.Object
10. ____ helps to extend the functionality of an existing by adding more methods to the sub class.
    (a) Mutual Exclusion (b) Inheritance
    (c) Package (d) Interface

Answers

1. (a)	2. (a)	3. (d)	4. (c)	5. (b)	6. (b)	7. (a)	8. (a)	9. (d)	10. (b)

Practice Questions

Q.I Answer the following Questions in short:

1. What is use of extends keyword?
2. What is Polymorphism? How to implement it?
3. What do you mean by Method overriding?
4. How to call constructor of super class into the constructor of sub class?
5. What is use of implement keyword?
6. What is borrowing in Type casting?
7. What is Automatic conversion?

Q.II Answer the following Questions:

1. Explain different Types of inheritance in details.
2. How to implement multiple inheritance in java? Explain with an example.
3. Explain the uses of super keyword with an example.
4. Explain type casting with an example.
5. Explain the role of constructor in inheritance.
6. Write a java program to accept the details of employee (eno, ename, sal) and display it. (Use super keyword)
7. Explain the uses of inheritance.
8. Explain the benefits of inheritance.
9. Define an interface Shape with abstract method area. Write a java program to calculate area of sphere.
10. What are the advantages of interface over abstract class?

Q.III Write short note on:

1. Inheritance
2. Type casting
3. Method overriding
4. Data members

■■■

4... Package

Objectives...

- To understand the concept of package.
- To learn different types of access specifier.
- To understand CLASSPATH Setting for Packages.
- To learn Making JAR Files for Library Packages.
- To understand the concept of Import and Static Import.
- To learn Naming Convention for Packages.

4.1 INTRODUCTION

4.1.1 Defining Package

- Package is a directory or an area where classes, sub-packages and interfaces are encapsulated.
- Packages are used for:
  - **Providing controlled access:** Protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only. Packages can be considered as data encapsulation (or data-hiding).
  - **Reusability:** While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
  - **Better Organization:** In a large java project where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.
  - **Name Conflicts:** We can define two classes with the same name in different packages so to avoid name collision, we can use packages.

4.2 NAMING CONVENTION FOR PACKAGES

- It is an area or directory in which user defined as well as system defined classes are stored. It is used to avoid naming confliction of classes and interfaces.
- While choosing a package name you need to keep the following points in mind.
  - The name of the package should be in small letters.
  - If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
  - It is suggested to start the name of the package with the top level domain level followed by sub domains, Example: dpu.dypacs.cs
- If you are not followed above points which are related with naming convention then there might be problem of Name Ambiguities.
- If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, the graphics package defined a class named Rectangle. The java.awt package also contains a Rectangle class.
- If both graphics and java.awt have been imported, the following is ambiguous.

```
Rectangle rect;
```

- In such situation, you have to use the member's fully qualified name to indicate exactly which Rectangle class you want. For example,

```
Graphics.Rectangle rect;
```

4.3 ORGANIZING CLASSES AND INTERFACES IN PACKAGES

4.3.1 Organizing Classes in Package

- A Package is a directory or an area in which classes, interfaces and sub packages are stored. We use packages to avoid name conflicts, and to write a better maintainable code.
- Packages are divided into two categories:
  1. Built-in Packages (packages from the Java API)
  2. User defined Packages (create your own packages)

1. Built-in Packages:

- The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment.
- The library is divided into *packages* and *classes*, so you can either import a single class (along with its methods and attributes), or a package that contain all the classes in an application.
- To use a class or a package from the library, you need to use the ***import*** keyword.

Example: `import PackageName.ClassName;`

- In java we have several built-in packages, for example when we need user input, we import a package like this:

```
import java.util.Scanner
```

Here: *Java* is a top level package, *util* is a sub package. *Scanner* is a class which is present in the sub package util.

2. User Defined Packages:

- Like built-in packages, user defined packages can be developed as follows:
- Steps to create a package:
  1. Create a directory i.e. package.
  2. Include the package command as the first line of code in your Java Source File.
  3. The Source file contains the classes, interfaces, etc you want to include in the package.
  4. Everything in package must be public except variable declaration.
  5. Compile to create the Java packages.

Example:

```
package DPU;
public class MyCollege
{
   public void disp()
   {
      System.out.println ("Yes..!..I m DYPIAN");
   }
}
```

- Save this file as name MyCollege.java in a DPU directory and compile it as follow:

```
javac -d directory javafilename
```

Example: `javac -d. MyCollege.java`

- The *-d* switch specifies the destination where to put the generated class file. You can use any directory name like d:\DPU. If you want to keep the package within the same directory, you can use . ***(dot)***.
- We can also compiled java program from DPU directory through command prompt as follow:

```
javac MyCollege.java
```

How to Access Package?

- There are three ways to access the package from outside the package.
  1. import package.*;
  2. import package.classname;
  3. Fully qualified name.

1. Using Package.*:

- If you use package.* then all the classes and interfaces except sub packages from that package you can use in an application.
- The import keyword is used to make the classes and interface of another package accessible to the current program.

Program 4.1: Java Program on how to access user defined package in a program using *.

```
import DPU.*;
class Demo
{
    public static void main (String args[])
    {
        MyCollege obj=new MyCollege();
        obj.disp();
    }
}
```

Output:

```
- Yes..!..I m DYPIAN
```

2. Using PackageName.ClassName:

- If you import package.classname then only declared class of this package will be accessible.

Program 4.2: Java Program on how to access user defined package in a program with class name.

```
import DPU.MyCollege;
class Demo
{
    public static void main (String args[])
    {
        MyCollege obj=new MyCollege();
        obj.disp();
    }
}
```

Output:

```
- Yes..!..I m DYPIAN
```

3. Using fully qualified name:

- If you use fully qualified name then only declared class of this package will be accessible, there is no need to import such package in an application, but you need to use fully qualified name every time when you are accessing the class or interface.

Program 4.3: Java Program on how to access user defined package in a program with fully qualified name.

```
class Demo
{
    public static void main(String args[])
    {
        DPU.MyCollege obj=new DPU.MyCollege();
        //DPU is Package and MyCollege is a class
        obj.disp();
    }
}
```

Output:

```
- Yes..!..I m DYPIAN
```

4.3.2 Subpackage in java

- Package inside the package is called the Subpackage. It should be created to categorize the package further.

  Example:

```
package DPU.DYPACS;
public class MyCollege
{
    public void disp()
    {
        System.out.println("We are DYPIAN");
    }
}
```

  Save this file as name MyCollege.java in a DYPACS directory.
  Compiled file as: javac –d . MyCollege.java
- We can access that package in another java program as follow:

```
import DPU.DYPACS.MyCollege;
class Demo
{
    public static void main (String args[])
    {
        MyCollege obj=new MyCollege();
        obj.disp();
    }
}
```

4.3.3 Organizing Interfaces in Package

- An interface is defined just like a class except the keyword **interface** in place of class.
- Member elements are present in an interface are static or final.
- Methods are defined in an interface are abstract.
- Classes can inherit from only one super class but, interfaces can inherit from multiple super interfaces. For inheriting interface in a class ***implements*** keyword is used and for inheriting an interface into another interface ***extends*** keyword is used.
- We can organize interfaces in package as follow:

Program 4.4: Java Program for the implementation of Interface in Package.

```
package DPU;
public interface SharedConstants
{
   int NO = 0;
   int YES = 1;
   int MAYBE = 2;
   int LATER = 3;
   int SOON = 4;
   int NEVER = 5;
}
import DPU.*;
import java.util.*;
class Question implements SharedConstants
{
   Random rand = new Random();
   int ask()
   {
      int prob = (int) (100 * rand.nextDouble());
      if (prob < 30)
      {
         return NO;
      }
      else if (prob < 60)
      {
         return YES;
      }
      else if (prob < 75)
      {
         return LATER;
      }
```



```
      else if (prob < 98)
      {
         return SOON;
      }
      else
      {
         return NEVER;
      }
   }
   static void answer(int result)
   {
      switch (result)
      {
         case NO:
         System.out.println ("No");
         break;
         case YES:
         System.out.println ("Yes");
         break;
         case MAYBE:
         System.out.println ("Maybe");
         break;
         case LATER:
         System.out.println ("Later");
         break;
         case SOON:
         System.out.println ("Soon");
         break;
         case NEVER:
         System.out.println ("Never");
         break;
      }
   }
   public static void main(String args[])
   {
      Question q = new Question ();
      answer (q.ask());
      answer (q.ask());
      answer(q.ask());
      answer(q.ask());
   }
}
```

Output:

```
c:\jdk1.8.0_221\bin>java Question
Yes
Soon
Yes
Later

c:\jdk1.8.0_221\bin>java Question
Yes
No
No
Yes
```

- The output of this program is depends upon value of prob variable which will be generated by nextDouble() function of Random class at run time.

4.4 PACKAGE AS ACCESS PROTECTION

- Access modifiers define the scope of the class and its members (data and methods).
- For example, private members are accessible within the same class members (methods). Java provides many levels of security that provides the visibility of members (variables and methods) within the classes, subclasses, and packages.
- Packages are meant for encapsulating, it works as containers for classes and other sub packages. Class acts as containers for data and methods.
- There are four categories, provided by Java regarding the visibility of the class members between classes and packages:
  1. Subclasses in the same package
  2. Non-subclasses in the same package
  3. Subclasses in different packages
  4. Classes that are neither in the same package nor subclasses
- The three main access modifiers *private, public* and *protected* provides a range of ways to access required by these categories.

	Private	No Modifier	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- Simply remember, private cannot be seen outside of its class, public can be access from anywhere, and protected can be accessible in subclass only in the hierarchy.
- A class can have only two access modifier, one is default and another is public. If the class has default access then it can only be accessed within the same package by any other code. But if the class has public access then it can be access from anywhere by any other code.

Examples:

1. **Default:**
   - If you don't use any modifier, it is treated as default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.
   - In this example, we have created two packages pack and mypack. We are accessing the class A from outside its package, since class A is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
   void msg()
   {
      System.out.println("Hello");
   }
}
//save by B.java
package mypack;
import pack.*;
class B
{
   public static void main(String args[])
   {
      A obj = new A(); //Compile Time Error
      obj.msg(); //Compile Time Error
   }
}
```

   - In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.
2. **Protected:**
   - The protected access modifier is accessible within package and outside the package but through inheritance only. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class. It provides more accessibility than the default modifier.
   - In this example, we have created the two packages pack and mypack. The class A of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A
{
    protected void msg()
    {
        System.out.println ("Hello");
    }
}
//save by B.java
package mypack;
import pack.*;
class B extends A
{
    public static void main(String args[])
    {
        B obj = new B ();
        obj.msg ();
    }
}
```

3. **Public:**
   - The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

```
//save by A.java
package pack;
public class A
{
    public void msg()
    {
        System.out.println ("Hello");
    }
}
//save by B.java
package mypack;
import pack.*;
class B
{
    public static void main(String args[])
    {
        A obj = new A();
        obj.msg();
    }
}
```


4.5 CLASSPATH SETTING FOR PACKAGES

- **CLASSPATH:** CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files.
- The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.
- You need to set the CLASSPATH if:
  - You need to load a class that is not present in the current directory or any sub-directories.
  - You need to load a class that is not in a location specified by the extensions mechanism.
- The CLASSPATH must be end with either directory name or filename. The following points describe,
  What should be the end of the CLASSPATH?
  - If a JAR or zip file contains class files, then CLASSPATH end with the name of the zip or JAR file.
  - If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
  - If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name that is the first package in the full package name.
- The default value of CLASSPATH is a dot (.). It means the only current directory searched. The default value of CLASSPATH overrides when you set the CLASSPATH variable or using the -classpath command (for short -cp). Put a dot (.) in the new setting if you want to include the current directory in the search path.
- If CLASSPATH finds a class file which is present in the current directory, then it will load the class and use it, irrespective of the same name class presents in another directory which is also included in the CLASSPATH.
- If you want to set multiple classpath, then you need to separate each CLASSPATH by a semicolon (;).
- There are two ways to set CLASSPATH: through Command Prompt or by setting Environment Variable.
- **To set classpath through environment variable:**
  Step 1 : Click on the Windows button and choose Control Panel. Select System.
  Step 2 : Click on Advanced System Settings.
  Step 3 : A dialog box will open. Click on Environment Variables.
  Step 4 : If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the path.

- If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as *Path that you have to set and put* ;;; (Semicolon comma semicolon) at the end of the path for setting new path.

4.6 MAKING JAR FILES FOR LIBRARY PACKAGES

- Packages within JAR files can be optionally sealed, which means that all classes defined in that package must be archived in the same JAR file. You might want to seal a package, for example, to ensure version consistency among the classes in your software.
- You seal a package in a JAR file by adding the Sealed header in the manifest, which has the general form:

```
Name: DPU/DYPACS/
Sealed: true
```

- The value DPU/DYPACS/ is the name of the package to seal.
- Note that the package name must end with a "/".
- **Example:**
- We want to seal two packages firstPackage and secondPackage in the JAR file MyJar.jar.
- We first create a text file named Manifest.txt with the following contents:

```
Name: DPU/firstPackage/
Sealed: true
Name: DPU/secondPackage/
Sealed: true
```

- **Note:** The text file must end with a new line or carriage return. The last line will not be parsed properly if it does not end with a new line or carriage return.
- We then create a JAR file named MyJar.jar by entering the following command:

```
jar cfm MyJar.jar Manifest.txt DPU/*.class
```

- This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
Created-By: 1.7.0_06 (Oracle Corporation)
Name: DPU/firstPackage/
Sealed: true
Name: DPU/secondPackage/
Sealed: true
```

Enhancing Security with Manifest Attributes:

- The following JAR file manifest attributes are available to help ensure the security of your applet or Java Web Start application. Only the Permissions attribute is required.
  - The Permission attribute is used to ensure that the application requests only the level of permissions that is specified in the applet tag or JNLP file used to invoke the application. Use this attribute to help prevent someone from re-deploying an application that is signed with your certificate and running it at a different privilege level.
  - This attribute is required in the manifest for the main JAR file.
  - The Codebase attribute is used to ensure that the code base of the JAR file is restricted to specific domains. Use this attribute to prevent someone from re-deploying your application on another website for malicious purposes.
  - The Application-Name attribute is used to provide the title that is shown in the security prompts for signed applications.
  - The Application-Library-Allowable-Codebase attribute is used to identify the locations where your application is expected to be found. Use this attribute to reduce the number of locations shown in the security prompt when the JAR file is in a different location than the JNLP file or the HTML page.
  - The Caller-Allowable-Codebase attribute is used to identify the domains from which JavaScript code can make calls to your application. Use this attribute to prevent unknown JavaScript code from accessing your application.
  - The Entry-Point attribute is used to identify the classes that are allowed to be used as entry points to your RIA. Use this attribute to prevent unauthorized code from being run from other available entry points in the JAR file.
  - The Trusted-Only attribute is used to prevent untrusted components from being loaded. See **Trusted-Only Attribute** in the Java Platform, Standard Edition Deployment Guide for more information.
  - The Trusted-Library attribute is used to allow calls between privileged Java code and sandbox Java code without prompting the user for permission.

Adding Classes to the JAR File's Classpath:

- You may need to reference classes in other JAR files from within a JAR file.
- For example, in a typical situation an applet is bundled in a JAR file whose manifest references a different JAR file (or several different JAR files) that serves as utilities for the purposes of that applet.
- You specify classes to include in the Class-Path header field in the manifest file of an applet or application. The Class-Path header takes the following form:

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

- By using the Class-Path header in the manifest, you can avoid having to specify a long -classpath flag when invoking Java to run your application.
- **Note:** The Class-Path header points to classes or JAR files on the local network, not JAR files within the JAR file or classes accessible over Internet protocols. To load classes in JAR files within a JAR file into the class path, you must write custom code to load those classes. For example, if MyJar.jar contains another JAR file called MyUtils.jar, you cannot use the Class-Path header in MyJar.jar's manifest to load classes in MyUtils.jar into the class path.

Example:

- We want to load classes in MyUtils.jar into the class path for use in MyJar.jar. These two JAR files are in the same directory.
- We first create a text file named Manifest.txt with the following contents:

```
Class-Path: MyUtils.jar
```

- We then create a JAR file named MyJar.jar by entering the following command:

```
jar cfm MyJar.jar Manifest.txt MyPackage/*.class
```

- This creates the JAR file with a manifest with the following contents:

```
Manifest-Version: 1.0
Class-Path: MyUtils.jar
Created-By: 1.7.0_06 (Oracle Corporation)
```

- The classes in MyUtils.jar are now loaded into the class path when you run MyJar.jar.

4.7 IMPORT AND STATIC IMPORT

- We can use an import statement to import classes and interface of a particular package. Whenever we are using import statement it is not required to use the fully qualified name and we can use short name directly.
- We can use static import to import static member from a particular class and package. Whenever we are using static import it is not required to use the class name to access static member and we can use directly.

import statement:

- To access a class or method from another package we need to use the fully qualified name or we can use import statements.
- The class or method should also be accessible. Accessibility is based on the access modifiers.
- Private members are accessible only within the same class. So we won't be able to access a private member even with the fully qualified name or an import statement.
- The java.lang package is automatically imported into our code by Java.

Program 4.5: Write a Java Program for the implementation Vector class with import statement.

```
import java.util.Vector;
public class ImportDemo
{
    public ImportDemo()
    {
        Vector v = new Vector();
        v.add("A");
        v.add("B");
        v.add("C");
```



```
        System.out.println("Vector values are: "+ v);
        //Package not imported, hence referring to it using the complete
        package.
            java.util.ArrayList list = new java.util.ArrayList();
            list.add("A");
            list.add("B");
            System.out.println("Array List values are: "+ list);
    }
    public static void main(String arg[])
    {
        new ImportDemo();
    }
}
```

Output:

```
Array List values are [A, B]
```

Static Import Statement:

- Static imports will import all static data so that can use without a class name.
- A static import declaration has two forms, one that imports a particular static member which is known as single static import and one that imports all static members of a class which is known as a static import on demand.
- One of the advantages of using static imports is reducing keystrokes and re-usability.

Program 4.6: Write a java program for the implementation of static import statement.

```
import static java.lang.System.*; //Using Static Import
public class StaticImportDemo
{
    public static void main(String args[])
    {
        System.out.println("Welcome to DYP");
    }
}
```

Output:

```
Welcome to DYP
```

- Static import allows you to access the static member of a class directly without using the fully qualified name. To understand this topic, you should have the knowledge of packages in Java.
- Static imports are used for saving your time by making you type less. If you hate to type same thing again and again then you may find such imports interesting.

- Lets understand this with the help of examples:

Program 4.7: Write a java program for the implementation of without static imports.

```
class Demo1
{
   public static void main(String args[])
   {
      double var1= Math.sqrt(5.0);
      double var2= Math.tan(30);
      System.out.println("Square of 5 is:"+ var1);
      System.out.println("Tan of 30 is:"+ var2);
   }
}
```

Output:

```
Square of 5 is:2.23606797749979
Tan of 30 is:-6.405331196646276
```

Program 4.8: Write a java program for the implementation of using static imports.

```
import static java.lang.System.out;
import static java.lang.Math.*;
class Demo2
{
   public static void main(String args[])
   {
      //instead of Math.sqrt need to use only sqrt
      double var1= sqrt(5.0);
      //instead of Math.tan need to use only tan
      double var2= tan(30);
      //need not to use System in both the below statements
      out.println("Square of 5 is:"+var1);
      out.println("Tan of 30 is:"+var2);
   }
}
```

Output:

```
Square of 5 is: 2.23606797749979
Tan of 30 is: -6.405331196646276
```

When to use static imports?

- If you are going to use static variables and methods a lot then it's fine to use static imports. For example, if you want to write a code with lot of mathematical calculations then you may want to use static import.

- **Drawbacks:**

  It makes the code confusing and less readable so if you are going to use static members very few times in your code then probably you should avoid using it. You can also use wildcard(*) imports.

Difference between import and static import:

- The import allows the Java programmer to access classes of a package without package qualification whereas the static import feature allows accessing the static members of a class without the class qualification.
- The import provides accessibility to classes and interfaces whereas static import provides accessibility to static members of the class.

Summary

- Package is a directory or an area where classes, subpackages and interfaces are stored.
- Everything in package must be public except variable declaration.
- Package can be defined by using package command.
- Package is used to avoid naming confliction between the names.
- The main use of package is reusability.
- There are two types of package:
  1. Built-in
  2. User defined
- For importing the packages in a program, import statement is used:
- Package can be accessed in following ways:
  1. Using Package.*
  2. Using PackageName.ClassName
  3. Using fully qualified name
- In java there are different types of access specifiers : Private, Protected, Default, Public.
- Classes and interfaces can be organized in a package:

  In a class we can define methods as well as variables of different types.

  In an interface we can declare abstract methods and final or static variables.
- In java jar file can be created by using jar command. Jar stands for java achieve file. Jar is nothing but exe or setup file of an java application.
- Manifest file acts as supporting file for the creation of jar file. A file having extension .txt and which contains single line code "Main-Class: Class Name "is called manifest file.

- In java we can import the packages in two ways:
  - The import allows the Java programmer to access classes of a package without package qualification whereas the static import feature allows accessing the static members of a class without the class qualification.
  - The import provides accessibility to classes and interfaces whereas static import provides accessibility to static members of the class.
- Naming conventions are used to give appropriate name to the package, for that following rules has to follow:
  - The name of the package should be in small letters.
  - If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
  - It is suggested to start the name of the package with the top level domain level followed by sub domains, ex: dpu.dypacs.cs

Check Your Understanding

1. Package in Java is a mechanism to encapsulate a _____.
   (a) Classes (b) Sub Packages
   (c) Interfaces (d) All of the above
2. Which of these keywords is used to define packages in Java?
   (a) pkg (b) Pkg
   (c) package (d) Package
3. Which of the following packages is used to includes utility classes like Calendar, Collection, Date?
   (a) java.lang (b) java.net
   (c) java.awt (d) java.util
4. Packages that are inside another package are the _____.
   (a) packages (b) nested packages
   (c) util subpackages (d) subpackages
5. Which of these can be used to fully abstract a class from its implementation?
   (a) Objects (b) Packages
   (c) Interfaces (d) None of the Mentioned
6. Which of the following is the correct way of implementing an interface A by class B?
   (a) class B extends A{} (b) class B implements A{}
   (c) class B imports A{} (d) None of the mentioned

7. What happens when we access the same variable defined in two interfaces implemented by the same class?
   (a) Compilation failure
   (b) Runtime Exception
   (c) The JVM is not able to identify the correct variable
   (d) The interfaceName.variableName needs to be defined.
8. What does an interface contain?
   (a) Method definition
   (b) Method declaration
   (c) Method declaration and definition
   (d) Method name
9. Which of the following is/are advantages of packages?
   (a) Packages avoid name clashes
   (b) Classes, even though they are visible outside their package, can have fields visible to packages only
   (c) We can have hidden classes that are used by the packages, but not visible outside.
   (d) All of the above
10. If a variable is declared as private , then it can be used in _____.
    (a) Any class of any package (b) Any class of same package
    (c) Only in the same class (d) Only subclass in that package

Answers

1. (d)	2. (c)	3. (d)	4. (d)	5. (c)	6. (b)	7. (d)	8. (b)	9. (d)	10. (c)

Practice Questions

Q.I Answer the following Questions in short:

1. What is use of Classpath?
2. What is use of Manifest file for the creation of jar file?
3. What is an Interface?
4. What do you mean by Naming convention?
5. What is use of Jar command?
6. What is package? State the types Packages.
7. Package accessing using fully qualified name:
8. What is use of package command?
9. How to access user defined package in java application.
10. How to create Subpackage?

Q.II Answer the following Questions:

1. What is package? Write down all the steps for package creation.
2. What are the different types of visibility controls?
3. Differentiate between interface and abstract class.
4. Explain different types of built in packages with their uses.
5. What is jar file? How to create it?
6. Differentiate between static import and import.
7. Explain the naming conventions for package in details.
8. How to set classpath to the package? Explain with an example.
9. Create a package DPU with a class Admission. Write a java program to accept the details of student (Name, MobileNo, course) and display it.
10. Create a package MCA with the classes Teacher and Student with suitable attributes. Write a java program to accept the details of teacher and student and display it.

Q.III Write short note on:

1. CLASSPATH
2. Built in packages
3. Jar and Manifest File
4. Uses of Package
5. Compilation and Execution of package.
6. Static import.

■■■

5... Exception Handling

Objectives...

- To understand what exceptions and errors are.
- To study when and how exceptions happen.
- To learn concepts of In-built and User Defined Exceptions Checked and Un-Checked Exceptions.
- To study how to implement exception handling.
- To understand the use of try, catch, finally, throw, throws statement.
- To learn What's JVM reaction to exception.

5.1 INTRODUCTION

- An exception is a divergence from an application's normal behavior. Java, being the most prominent object-oriented language, provides a powerful mechanism to handle these errors/exceptions.
- Exception can occur for various reasons like user has entered an invalid data, file not found, network connection has been lost in the middle of communications, JVM has run out of a memory etc.
- We can handle exception by catching in the same method and try to recover from it, or catch it and rethrow it, or just leave the handling part to the caller of this method. If no one handles the exception, it is finally thrown to JVM and the program halts execution.
- In Java, exceptions broadly categories into checked exception, unchecked exception.
- We can create our own exception class called as user-defined or custom exceptions. In this chapter we will study a complete insight into the fundamentals, types, and methods of Exception Handling.

5.2 THE IDEA BEHIND EXCEPTION

- An exception is an abnormal situation that is occurred during the program execution. In simple words, an exception is a problem that arises at the runtime.

- When an exception occurs, the program execution gets terminated, and the system generates an error. We use the exception handling mechanism to avoid abnormal termination of program execution.
- An exception is an object that describes an erroneous or unusual situation.
- Exceptions in java can arise from different kind of situations such as wrong input entered by user, hardware failure, network connection failure, Database server down etc. In this section, we will learn how exceptions are handled in java.
- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.
- During the execution of a program, various conditions can occur that cause an unexpected and abnormal termination of the program.

Example: Consider the following program:

```
public class TestException
{
    public static void main (String[] args)
    {
        int a=10;
        int b = Integer.parseInt("Hi");
            int c=a+b;
            System.out.println(" c + "End of the program");
    }
}
```

- The following message is printed on the screen:

  Exception in thread "main" java.lang.NumberFormatException: For input string: "Hi"
- In this example, program has bunch of statements and an exception occurs mid-way after executing certain statements then the statements after the exception will not execute and the program will terminate suddenly.
- Hence, c= a+b is not evaluated and the string "End of the program" is not printed. Java offers a predefined set of exceptions that can be thrown during program execution. To avoid that a program terminates unexpectedly, Java allows us to handle exceptions by means of a suitable construct.
- The advantage of Exception handling is to maintain the normal flow of the program.

5.3 EXCEPTION AND ERRORS

Exception Class:

- Exception class is for exceptional conditions that program should catch. All exception classes are subtypes of the java.lang.Exception class. The Exception class is a subclass of the Throwable class. This class is extended to create user specific exception classes.

- A hierarchy of Java Exception classes are given below:

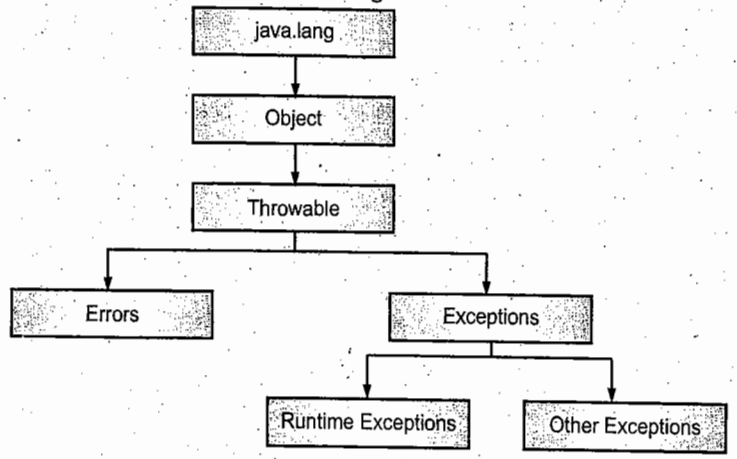


Fig. 5.1: Exception Hierarchy

- The java.lang.Throwable class is the root class of Java Exception hierarchy.
- Here, the class Throwable is used to represent all exceptional conditions. Two immediate subclasses of Throwable are Exception, and Error. The class Exception is used for exceptional conditions that user programs can catch.
- A further refinement is done by a sub class of Exception, which is for exceptional condition that created at run time called RuntimeException.

Errors:

- An Error "indicates serious problems that a reasonable application should not try to catch."
- Both Errors and Exceptions are the subclasses of java.lang.Throwable class.
- Errors are the conditions which cannot get recovered by any handling techniques. It surely causes termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime.
- Some of the examples of errors are Dynamic linking failure, Memory shortage, Instantiating abstract class, InternalError, IllegalAccessErrors, StackOverFlowError etc.
- The other branch of the Throwable tree is the class Error, which defines the conditions that should not be expected to be caught under normal circumstances. This class is responsible for giving errors in some catastrophic failures.

5.3.1 Errors vs. Exceptions

- Following table shows difference between Errors and Exceptions.

Table 5.1: Difference between Errors and Exceptions

Errors	Exceptions
1. Impossible to recover from error.	1. Possible to recover from Exception.
2. Errors are of type unchecked exception.	2. Exceptions can be checked type or unchecked type.
3. All errors are of type java.lang.Error	3. All exceptions are of type java.lang.Exception
4. They are not known to compiler. They happen at run time.	4. Checked exceptions are known to compiler whereas unchecked exceptions are not known to compiler.
5. Errors are caused by the environment in which the application is running.	5. Exceptions are caused by the application.
Examples : `java.lang.StackOverflowError`, `java.lang.OutOfMemoryError`	**Examples :** `Checked Exceptions : SQLException, IOException` `Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException`

5.4 IN-BUILT AND USER DEFINED EXCEPTIONS

In-built Exceptions:

- Java provides rich set of built-in Exception classes. All these classes are available in the java.lang package and used in exception handling.
- Let's see some built-in exception in the following table with their meaning.

Table 5.2: In-built Exceptions

Exception	Meaning
`ArithmeticException`	Arithmetic error, such as divide-by-zero.
`ArrayIndexOutOfBoundsException`	Array index is out-of-bounds.
`ArrayStoreException`	Assignment to an array element of an incompatible type.
`ClassClassException`	Invalid cast.
`EnumConstantNotPresentException`	An attempt is made to use an undefined enumeration value.
`IllegalArgumentException`	Illegal argument used to invoke a method.

contd. ...

`IllegalMonitorStateException`	Illegal monitor operation, such as waiting on an unlocked thread.
`IllegalStateException`	Environment or application is in incorrect state.
`IllegalThreadStateException`	Requested operation not compatible with current thread state.
`IndexOutOfBoundsException`	Some type of index is out-of-bounds.
`NegativeArraySizeException`	Array created with a negative size.
`NullPointerException`	Invalid use of null reference.
`NumberFormatException`	Invalid conversion of a string to a numeric format.
`SecurityException`	Attempt to violate security.
`StringIndexOutOfBounds`	Attempt to index outside the bounds of a string.
`TypeNotPresentException`	Type not found.
`UnsupportedOperationalException`	An unsupported operation was encountered.

User defined Exception:

- Apart from built-in exception classes, Java allows us to create our own exception class to provide own exception implementation.
- If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message that can be thrown using throw keyword.
- You can create your own exception simply by extending java Exception class. You can define a constructor for your Exception (not compulsory) and you can override the toString() function to display your customized message on catch.

Program 5.1: Program for user defined exception.

```
public class demo extends Exception
{
    public String toString()
    {
        return "Custom Exception";
    }
    public static void main(String[] args)
    {
        demo d= new demo();
```

```
        try
        {
            throw d;
        }
        catch(demo e)
        {
            System.out.println("Exception handled - "+ e);
        }
    }
}
```

Output:

```
Exception handled - Custom Exception
```

5.5 TYPES OF EXCEPTION

- In Java, exceptions are mainly categorized into two types, and they are as follows.
  1. Checked exceptions
  2. Unchecked exceptions

1. Checked Exceptions:

- Any exception that is checked by the compiler during compile time is called as Checked Exception.

  Checked exceptions = checked by the compiler

- Checked exceptions are also known as compile-time exceptions as these exceptions are checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If not, then the system displays a compilation error.
- Checked Exception are the classes which inherit Throwable class except RuntimeException and Error. If these exceptions are not handled/declared in the program, you will get compilation error.
- The following are a few built-in classes used to handle checked exceptions in Java.
  - IOException
  - FileNotFoundException
  - ClassNotFoundException
  - SQLException
  - DataAccessException
  - InstantiationException
  - UnknownHostException
- A checked exception can be thrown within a try block or by using throws keyword.

Program 5.2: Program for the checked exception method.

```
public class Main
{
    public static void main(String[] args)
    {
        File f= new File("D:\\Java\\sample.txt");
        try
        {
            FileReader fr=new FileReader(f);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output:

```
java.io.FileNotFoundException: D:\Java\sample.txt (No such file or directory)
```

- In above example assume that sample.txt file is not present in Java directory, thus at compile time you will get java.io.FileNotFoundException

2. Unchecked Exceptions:

- The unchecked exception is an exception that occurs at the time of program execution.

  Unchecked exceptions = Runtime exceptions

- Runtime Exceptions are also known as Unchecked Exceptions.
- Unchecked exceptions are the classes which inherit RuntimeException and Error.
- These exceptions are not checked at compile time, so compiler does not check whether the programmer has handled them or not but it is the responsibility of the programmer to handle these exceptions and provide a safe exit.
- The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.
- The following are a few built-in classes used to handle unchecked exceptions in java.
  - ArithmeticException
  - NullPointerException
  - NumberFormatException
  - ArrayIndexOutOfBoundsException
  - StringIndexOutOfBoundsException

Program 5.3: Program to demonstrate different unchecked Exceptions.

```
public class unchecked_demo
{
   public static void main(String[] args)
   {
      int arr[] = {10,20,30,40,50};
      System.out.println(arr[6]);
                                  // example of ArrayIndexOutOfBoundsException
      String str=null;
      System.out.println(str.length()); // example of NullPointerException
      String name="Hello";
      int i=Integer.parseInt(name);
                                     // example of NumberFormatException
   }
}
```

5.6 USE OF try, catch, finally, throw, throws IN EXCEPTION HANDLING

- Java has a built-in mechanism for handling runtime errors, referred to as *exception handling*. This is to ensure that you can write *robust* programs for mission-critical applications.
- When an exception occurs program execution gets terminated. In such cases we get a system generated error message.
- The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.
- Java's exception handling consists of three operations:
  1. Declaring exceptions.
  2. Throwing an exception.
  3. Catching an exception.
- Five keywords are used in exception handling: try, catch, finally, throw and throws.

1. try:

- The try statement allows you to define a block of statements to be tested for errors while it is being executed.
- A try block is always followed by a catch block, which handles the exception that occurs in associated try block.
- A try block must be followed by catch blocks or finally block or both.

- Its syntax is:

```
try
{
   //statements that may cause an exception
}
```

2. catch:

- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The catch block also known as handler which is used to handle the exception. It handles the exception thrown by the code enclosed into the try block. Try block must provide a catch handler or a finally block.
- A catch block is where you handle the exceptions. This block must follow the try block.
- For example, if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.
- Its syntax is:

```
try
{
   //statements that may cause an exception
}
catch (exception_type eob)
{
   //error handling code
}
```

Program 5.4: Program to demonstrate the use of try – catch statement.

```
public class catchDemo
{
   public static void main(String[] args)
   {
       int a = Integer.parseInt(args[0]);
       int b = Integer.parseInt(args[1]);
       try
       {
           int c = a / b;
           System.out.println("The quotient is "+ c);
       }
       catch (ArithmeticException e)
       {
           System.out.println("Divisor should not be 0");
       }
   }
}
```

Output:

```
if we assume a=10 and b=10 then output will be The quotient is 1.
if we assume a=10 and b=0 then output will be Divisor should not be 0.
```

Multiple catch statements:

- A single try block can have one or more catch blocks associated with it. Each catch block must contain a different exception handler. You can catch different exceptions in different catch blocks.
- When an exception occurs in try block, the corresponding catch block that handles that particular exception, executes. So, if you have to perform different tasks at the occurrence of different exceptions, use java multiple catch block.
- Its syntax is:

```
try
{
    //statements that may cause an exception
}
catch (ExceptionType1 eob)
{
    //error handling code
}
catch (ExceptionType2 eob)
{
    //error handling code
}
```

Points to Remember:

- Single try block can have any number of catch blocks.
- At a time only one exception occurs and its corresponding catch block is invoked.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.
- If no exception occurs in try block then the catch blocks are completely ignored.
- Corresponding catch blocks execute for that specific type of exception:

  `catch(ArithmeticException e)` is a catch block that can handle `ArithmeticException`

  `catch(NullPointerException e)` is a catch block that can handle `NullPointerException`
- The catch-block declared with a specific exception class must always be placed above the catch-block declared with a Exception class; otherwise, Java Compiler will throw a compile-time error.

Program 5.5: Following program demonstrate the use of multiple catch statements.

```
public class Main
{
    public static void main(String[] args)
    {
        try
        {
            int arr[] = {10,20,30};
            System.out.println("Value = "+ arr[5]);
        }
        catch (ArithmeticException e)
        {
            System.out.println("Exception caught" +e);
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Exception caught : " +e);
        }
        catch (Exception e)
        {
            System.out.println("Exception caught : " +e);
        }
    }
}
```


Output:

```
Exception caught : java.lang.ArrayIndexOutOfBoundsException: Index 5 out of
bounds for length 3
```

3. finally:

- A try statement can have an optional clause designated by the reserved word *finally*.
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete.
- Also, if an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete.
- There are two scenarios in which finally block can be defined:

Case 1: A finally block can be defined right after the try-catch block.

Program 5.6: Program to demonstrate finally block can be defined right after the try-catch block.

```
public class demo
{
    public static void main(String[] args)
    {
        try
        {
            Integer ob = new Integer("Hundred");
        }
        catch(Exception e)
        {
            System.out.println("Exception caught - "+e);
        }
```

```
        finally
        {
            System.out.println("Even after an exception is caught in the
            catch-block still finally block is executed ");
        }
    }
}
```

Output:

```
Exception caught-
                java.lang.NumberFormatException: For input string: "Hundred"
Even after an exception is caught in the catch -
                                        block still finally block is executed
```

Case 2: A finally block can be defined right after the try block(if catch block is not defined)

Program 5.7: Program to demonstrate finally block can be defined right after the try block.

```
public class demo
{
    public static void main(String[] args)
    {
        try
        {
            Integer ob = new Integer("5");
        }
        finally
        {
System.out.println("Even though catch block is not defined, finally block
is executed ");
        }
    }
}
```

Output:

```
Even though catch block is not defined,finally block is executed
```

When to use finally keyword?

- Objects may hold references to the system resources like memory, files, database which need to be released in a timely fashion when they are no longer needed.
- Hence, a finally block usually contains a clean-up code i.e. code to release any of such resources that were referenced or locked in the try block.
- In short, finally block is used to execute essential code, especially to release the occupied resources.

4. throw:

- In Java, you may use the throw keyword when you explicitly want to throw a single exception.
- When an exception is thrown, the flow of program execution transfers from the try block to the catch block. We use the throw keyword within a method.
- Its syntax is: `throw throwableObject;`
- A throwable object is an instance of class Throwable or subclass of the Throwable class.
- Using *throw* keyword, an exception can be explicitly thrown from either in try-block or catch-block.
- To throw an exception out of try block, keyword *throw* is followed by creating a new object of exception class that we wish to throw.

Program 5.8: Program to illustrate throw in try block.

```
class demo
{
    public static void main(String args[])
    {
        try
        {
            throw new ArithmeticException("test");
        }
        catch(ArithmeticException e)
        {
            System.out.println("Exception caught");
        }
    }
}
```

Output:

```
Exception caught
```

Program 5.9: Program to illustrate throw in catch block

```
public class demo
{
    public static void main(String[] args)
    {
        try
        {
            Integer ob = new Integer("hi");   //throws NumberFormatException
        }
```

```
        catch(Exception e)
        {
            System.out.println("Caught exception is thrown again");
            throw e; //throwning the caught exception
        }
    }
}
```

Output:

```
Caught exception is thrown again
Exception in thread "main" java.lang.NumberFormatException: For input string: "hi"
java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
at java.base/java.lang.Integer.<init>(Integer.java:1096)
at demo.main(demo.java:4)
```

- In this code, an exception of type NumberFormatException is thrown from try block, when an illegal string, which can't be converted to an Integer object, is passed to the constructor of wrapper class, Integer. Next to try block, a catch block is declared with Exception class, which is a superclass of every exception class in Java. Hence, the catch block catches the exception NumberFormatException and this exception is thrown again using throw keyword.

5. throws keyword:

- We use the throws keyword in the method declaration to declare the type of exceptions that might occur within it.
- Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the throws keyword.
- Its syntax is:

```
type method_name(parameter_list) throws exception_list
{
  // definition of method
}
```

Program 5.10: Program to illustrate the use of throws keyword.

```
class demo
{
    public static void search_file() throws IOException
    {
        throw new IOException("File is not present");
    }
```



```
    public static void main(String[] args)
    {
        try
        {
            search_file();
        }
        catch (IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
File is not present
```

Difference between throw and throws keywords:

Table 5.3: Difference between throw and throws

Throw	throws
1. Used to throw an exception explicitly.	1. Used to declare an exception possible during its execution.
2. throw keyword is declared inside a method body.	2. throws keyword is used with method signature (method declaration).
3. We cannot throw multiple exceptions using throw keyword.	3. We can declare multiple exceptions (separated by commas) using throws keyword.
4. Followed by an instance of Throwable class or one of its sub-classes.	4. Followed by one or more Exception class names separated by commas.

5.7 CONTROL FLOW IN EXCEPTIONS

Flow of control in try/catch/finally blocks:

- If exception occurs in try block's body then control immediately transferred (skipping rest of the statements in try block) to the catch block. Once catch block finished execution then finally block and after that rest of the program.
- If there is no exception occurred in the code which is present in try block then first, the try block gets executed completely and then control gets transferred to finally block (skipping catch blocks).
- If a return statement is encountered either in try or catch block. In this case finally block runs. Control first jumps to finally and then it returned back to return statement. Let's consider following example:

Program 5.11: Program for use of try, catch and finally keyword.

```
class demo
{
   public static int fun()
   {
      try
      {
         return 1;
      }
      finally
      {
         System.out.println("In finally block");
      }
   }
   public static void main(String[] args)
   {
      System.out.println(demo.fun());
   }
}
```

Output:

```
In finally block
1
```

5.8 JVM REACTION TO EXCEPTIONS

- When an exception occurs inside a Java method, the method creates an Exception object and passes the Exception object to the JVM (in Java term, the method "throw" an Exception). The Exception object contains the type of the exception, and the state of the program when the exception occurs.
- The JVM is responsible for finding an *exception handler* to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the Exception).
- If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.

Summary

- Exception handling maintains the normal flow of the application.
- The try keyword must be used to monitor for exceptions.
- The throw keyword is used to manually throw an exception.
- The throws keyword is used by calling function to handle exception thrown by called function.
- The finally part of code gets executed whether exception is caught or not.

- Use the finally blocks instead of catch blocks if you are not going to handle the exception.
- Error class can catch all exceptions which cannot be caught.
- Checked exceptions are the exceptions which are known to compiler. These exceptions are checked at compile time only. Hence the name checked exceptions.
- Unchecked exceptions are those exceptions which are not at all known to compiler. These exceptions occur only at run time. These exceptions are also called as run time exceptions.

Check Your Understanding

1. When does Exceptions in Java arises in code sequence?
   (a) Run Time (b) Compilation Time
   (c) Can Occur Any Time (d) None of the mentioned
2. The class at the top of exception class hierarchy is _____
   (a) ArithmeticException (b) Throwable
   (c) Object (d) Exception
3. In _____ package exception class exists.
   (a) java.util (b) java.lang
   (c) java.io (d) java.net
4. Exception generated in try block is caught in _____ block.
   (a) catch (b) finally
   (c) throw (d) throws
5. What keyword is used to explicitly raise an exception?
   (a) catch (b) throw
   (c) throws (d) raise
6. What will happen when catch and finally block both return value?
   (a) method will return value returned by finally block
   (b) method will return value returned by catch block
   (c) finally block won't execute
   (d) None
7. What type of Exceptions can NOT be ignored at compile time?
   (a) Checked Exceptions (b) Unchecked Exceptions
   (c) Uncompiled Exceptions (d) Error
8. What will be the output of the following Java program?

```
class exception_handling
{
   public static void main(String args[])
   {
      try
      {
         System.out.print("Hi" + " " + 10 / 0);
      }
```

```
        catch(ArithmeticException e)
        {
            System.out.print("Bye");
        }
    }
}
```

(a) Hi (b) Bye

(c) HiBye (d) Hi Bye

Answers

1. (a)	2. (b)	3. (b)	4. (a)	5. (b)	6. (a)	7. (a)	8. (b)

Practice Questions

Q.I Answer the following Questions in short:

1. Which class is the super class for all types of errors and exceptions in Java?
2. What are exception handling keywords in Java?
3. What is the difference between throw and throws in Java?
4. What is the difference between an exception and error?
5. What happens when an exception is thrown by the main method?
6. What are the advantages of Java exceptions?

Q.II Answer the following Questions:

1. How the exceptions are handled in java?
2. Explain the Java exception hierarchy.
3. What is the difference between checked and unchecked exceptions in Java?
4. List and explain any 5 in-built exceptions.
5. How Can We Write a Custom Exception in Java?
6. Write a Java program to implement the concept of exception handling
7. Write an java program that shows how to create a user-defined exception.

Q.III Write short note on:

1. Error
2. Exception
3. try statement
4. finally keyword
5. throw keyword
6. Checked exception
7. Unchecked exception

■■■

6... Array and String

Objectives...

- To understand how arrays are implemented in Java.
- To declare, instantiate, populate, and use arrays of primitives.
- To learn use of multi-dimensional arrays.
- To study use of the String class and some of its methods in Arrays.
- To learn use of the StringBuffer class and some of its methods.
- To learn how to use StringTokenizer Class.

6.1 INTRODUCTION

- A Java class library provides several classes to store collections of data that can be homogeneous (of same type) or heterogeneous (of different types). Heterogeneous collection of data is stored in the classes belonging to Java collection packaged whereas homogenous collections of data are stored in generic collections.
- Apart from classes, Java provides basic data types that can store a collection of homogeneous data. One such data type is the Java array data type. Arrays are used to store collections of data of the same type. Java also supports Multi-dimensional arrays.
- Strings are basically a sequence of characters that convey some meaningful interpretation to the user. String is immutable and it is final class in Java. StringBuffer and StringBuilder are classes used for String manipulation. These are mutable objects, which provide methods such as substring(), insert(), append(), delete() for String manipulation.

6.2 ARRAY DEFINITION

- Array is collection of similar type of elements.
- For Example: Let's consider the situation where we need to get 5 student's roll number. Since roll number is an integer type, we can store it something like below:

```
int r1, r2, r3, r4, r5;
```

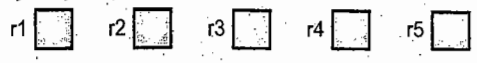


Fig. 6.1: Variable

- To store 5 student's roll number we have created 5 different variables. Every single variable can contain a single value, under that variable name.
- It will be very difficult for us to remember all identifiers to manipulate the data. To overcome this kind of situation, we should use **Array** data structure.
- An array can contain more than one value at time under same variable name.
- All array elements are stored in a contiguous memory location. An array is a special kind of object which is collection of variables of same type.

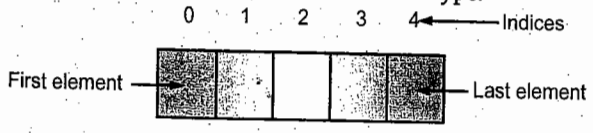


Fig. 6.2: Array

- For instance, an array of int is a collection of variables of the type int. The variables in the array are ordered and each have an index. Array is index-based, the 1st element of the array is stored at the 0th index, 2nd element is stored at 1st index and so on. The index value of an array starts with '0' and ends with 'size-1'. We use the index value to access individual elements of an array.

6.2.1 Declaring an Array

- Here is how we can declare an array.

Syntax:

```
data_Type array_Name[ ];
array_Name = new data_Type[size];
        or
data_Type array_Name [] = new data_Type[size];
```

where,

dataType: It can be primitive data types like int, char, double, etc. or Java objects.

arrayName: It is an identifier.

[]: A pair of square brackets [] indicate that we have an array.

Size: The size of an array is also known as the length of an array.

new: An array must be created using new operator and with a specific size. The size must be an integer value but not a byte, short, or long.

- **Note:** Once the size of the array is defined, it cannot be changed in the program.
- **Example:** `int[] arr = new int[5];`

Here, arr is an array. It can hold 5 values of int type.

- **Note:** The array type does not specify its size, but each object of that type has a specific size.
- If the length of an array is n, the first element of the array will be array_Name[0] and the last element will be array_Name[n-1].

- **Examples of array declarations:**

```
int[] A = new int[20];
boolean[] B;
B = new boolean[10];
char[] C = new char[30];
double[] D = new double[40];
```

Auto-Initialization:

- The initialization of variables to a default value, as in the initialization of array elements when an array is constructed.

Type	int	double	char	boolean	object
Value	0	0.0	'\0'	false	null

- When Java performs auto-initialization, it always initializes to the zero-equivalent for the type. Above table indicates the zero-equivalent values for various types.
- If we did not store any value to an array, then array will be auto-initialized.

6.3 INITIALIZING AND ACCESSING ARRAY

6.3.1 Initializing array

- There are several ways how we can initialize an array in Java. An array is created and initialized in two ways.

1. **An array can be initialized when it is declared.**

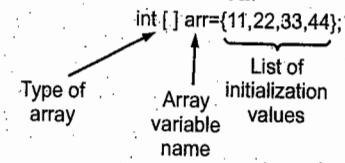


Fig. 6.3: Array Declaration, Instantiation and Initialization at same time

- **Declaration:** Assigns a name to the reference.
- **Instantiation:** Creates space for the object.
- **Initialization:** Gives the objects values for the first time.

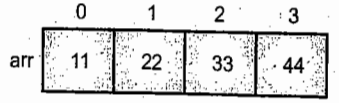


Fig. 6.4: Array of 4 integer element

- In this example, statement creates an array named arr and initialized it with the values provided in the curly braces. The length of the array is determined by the number of values provided inside the curly braces separated by commas. In our example, the length of arr is 4.

2. An array can be declared first and then initialize.

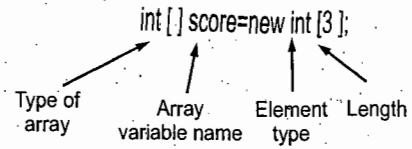


Fig. 6.5: Array Declaration

- score[0] = 100; // insert 100 to first element
- score[1] = 200; // insert 200 to second element
- score[2] = 300; // insert 300 to third element

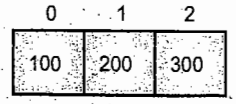


Fig. 6.6: Array with 3 integer elements

- In the above example, we have first created an array of named score and then initialized its elements one by one.

6.3.2 Accessing array

- We can easily access and alter elements of an array by using its numeric index. After the array is created, its elements can be accessed by their index. The index is a number placed inside square brackets which follow the array name.
- **Example:**

```
String[] states = {"Pune", "Goa", "Mumbai", "Nashik"};
System.out.println(states[0]);
System.out.println(states[1]);
System.out.println(states[2]);
System.out.println(states[3]);
```

- In the example, we created an array of string named states. We accessed each of the elements by its index and printed them to the terminal.
- It is possible to change the elements of an array. For example,

```
int[ ] nums = { 1, 2, 3 };
An array of three integers is created.
nums[0] *= 2;
nums[1] *= 2;
nums[2] *= 2;
```

- Using the element access, we multiplied each value in the array by two. All three integers have been multiplied by number 2.

6.3.3 Bounds Checking

- Once an array is created, it has a fixed size (e.g. N). Index value must be in range 0 to N–1.
- The Java interpreter throws an ArrayIndexOutOfBoundsException if an array index is out of bounds. This is called automatic bounds checking.

- For example

```
int x[]={ 99, 98, 97};
  for (int index=0; index <= 10; index++)
      System.out.println("Value =" +x[index]);
```

- In this example array x stored 3 values, so it can be indexed from 0 to 2. Here for loop is trying to fetch values from 0^{th} index to 10^{th} index. So interpreter throws an ArrayIndexOutOfBoundsException.

6.4 MULTI-DIMENSIONAL ARRAY

- Arrays are classified into two types. They are as follows:
  1. Single Dimensional Array / One Dimensional Array
  2. Multi-Dimensional Array
- So far we have been working with one dimensional arrays. In Java, we can create multi-dimensional arrays. A multi-dimensional array is an array of arrays. In such array, the elements are themselves arrays. In multi-dimensional arrays, we use two or more sets of brackets.
- The Java multi-dimensional arrays are arranged as an array of arrays i.e. each element of a multi-dimensional array is another array. The representation of the elements is in rows and columns. Thus, you can get a total number of elements in a multi-dimensional array by multiplying row size with column size.
- Multi-dimensional array can be of two dimensional array or three dimensional array or four dimensional array or more.
- Most popular and commonly used multi-dimensional array is two dimensional array. A two-dimensional array is actually an array of a one dimensional array. The 2D arrays are used to store data in the form of table. Popular application of multi-dimensional arrays is in matrix manipulation.

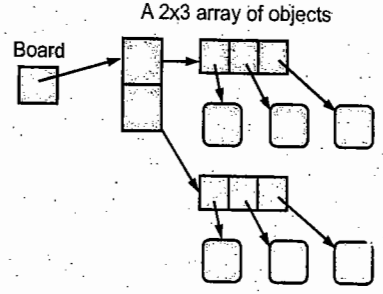


Fig. 6.7: Multi-dimensional array

How to Create Two dimensional Array?

- If you know how to create one dimensional array, then creating a two dimensional array is very easy. You can define a 2D array in Java as follows:
- The general declaration of a two dimensional array is,

```
data_type[ ] [ ] arr = new data_type [Row_Size][Column_Size];
```

Here:

Row_Size: Number of Row elements an array can store. For example, Row_Size = 5, then the array will have five rows.

Column_Size: Number of Column elements an array can store. For example, Column_Size = 6, then the array will have 6 Columns.

- Similar to the one dimensional array, the length of the two dimensional array is also fixed. You can not change the length of an array, i.e the number of rows and columns will remain fixed.
- A 2x2 array can hold total 4 elements and they can be accessed using row and column index like a[0][0] will give you elements in the first row and first column, similarly a[1][1] will give you elements from 2nd row and 2nd column. Just like a normal array, the index starts at 0 and finishes at length -1.
- In Java, a two dimensional array is stored in the form of rows and columns and is represented in the form of a matrix as follows:

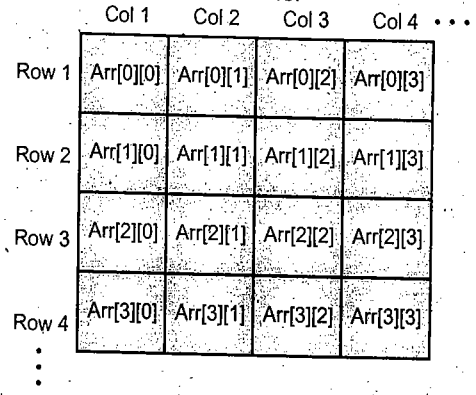


Fig. 6.8: Representation of 2D array

- **Some other examples of 2D array declarations:**

```
int [ ][ ] roll = new int[2][4];
                                  // 2D integer array with 2 rows and 4 columns
String [ ][ ] names = new String[3][3];
                                  // 2D String array with 3 rows and 3 columns
```

- **Note:** When you declare a two-dimensional array, you must remember to specify the first dimension, for example, following array declaration is illegal in Java.

```
int [ ][ ] wrong = new int[ ][ ];
                                  // not OK, you must specify 1st dimension
int [ ][ ] right = new int[2][ ]; // OK
```

- The first expression will throw "Variable must provide either dimension expressions or an array initializer" error at compile time. On the other hand, the second dimension is optional and even if you don't specify compiler will not complain, as shown below:

```
String [ ][ ] X = new String[5][];  // OK
String [ ][ ] Y = new String[5][4]; // OK
```


How to Initialize Two Dimensional Array?

- Once the array is declared and created, it is time to initialize it with values.
- There are various ways of initializing the 2D array.

Method 1: The first method is the traditional method of assigning values to each element.

The general syntax for initialization is:

```
array_name[row_index][column_index] = value;
```

Example:

```
Int [ ][ ] my_arr = new int[2][2];
my_arr [0][0] = 10;
my_arr [0][1] = 20;
my_arr [1][0] = 30;
my_arr [1][1] = 40;
```

The above statements initialize all the elements of the given 2D array.

This method may be useful when the dimensions involved are smaller. As the array dimension grows, it is difficult to use this method of individually initializing the elements.

Method 2: Another method of initializing the 2D array is by initializing the array at the time of declaration only. The general syntax for this initialization method is as given below:

```
data_type [ ][ ] array_name =
{
   {val_r1c1, val_r1c2,...val_r1cn},
   {val_r2c1, val_r2c2,...val_r2cn},
   …
   {val_rnc1, val_rnc2,… val_rncn}
};
```

For Example, if you have a 2x3 array of type int, then you can initialize it with the declaration as:

```
int [ ][ ] arr = { {1, 2, 3}, {4, 5, 6} };
```

Program 6.1: The following program shows the 2D array declaration with initialization.

```
public class demo
{
   public static void main(String[] args)
   {
      int [ ][ ] X = { { 1, 2 }, { 3, 4 }, {5,6} };
                                  //2D array initialized with values
      System.out.println("Initialized Two dimensional array:");
```

```
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 2; j++)
            System.out.print(X[i][j] + " ");
        }
    }
}
```

Output:

```
Initialized Two dimensional array:
1 2 3 4 5 6
```

- In the above program, the array is initialized at the time of the declaration itself and then the values are displayed.

Advantages of Arrays:

1. **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
2. **Random access:** We can access any element of array by its index.

Disadvantages of Arrays:

1. You can't change the size of an array in the middle of program execution.
2. We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.
3. You can't compare arrays for equality using a simple == test. Remember that arrays are objects, so if you ask whether one array is == to another array, you are asking whether they are the same object, not whether they store the same values.
4. You can't print an array using a simple print or println statement. You will get odd output when you do so.

6.5 OPERATION ON STRING, MUTABLE AND IMMUTABLE STRING

Definition of String:

- String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The **java.lang.String** class is used to create string object.
- The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.
- All string objects are stored in either String Constant Pool or in Heap memory.
- String constant pool is a special type of memory dedicated to store String objects only.
- There are two ways through which we can declare and define string objects.

1. By string literal: Java String literal is created by using double quotes.
   For Example: String str = "Best Luck";
   In this case, only one object will be created in SCP. Here 'str' will point to value in SCP only.

2. By new keyword: Java String is created by using a keyword "new".
   For example: `String str = new String("Best Luck ");`
   In this case two objects will be created. One is in SCP and other in Heap. One reference variable where the variable 'str' will refer to the object in the heap.

Mutable String:

- Mutable strings are those strings whose content can be changed without creating a new object. Mutable string in java can be created using StringBuffer and StringBuilder classes.
- We will learn more about mutable string in section 6.7.

Immutable String:

- Immutable means we cannot be changed. Whenever we change any string, a new instance is created. The String is Immutable in Java because String objects are cached in String pool.

Program 6.2: Demonstration of immutable string.

```
class immutable_demo
{
    public static void main(String args[])
    {
        String str="Jammu ";
        str.concat(" Kashmir");
        //concat() method appends the string at the end
        System.out.println(str);
        //will print Jammu as strings are immutable objects
    }
}
```

Output:

```
Jammu
```

- In the above program, the string where str is pointing i.e "Jammu" is not changed but a new object is created with "Jammu Kashmir". That is why string is known as immutable.

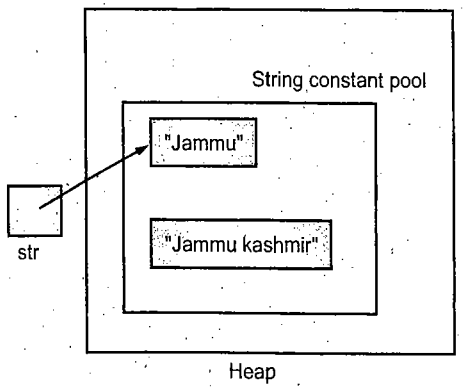


Fig. 6.9: Immutable String

Constructors of String class:

Table 6.1: Constructors of string class

Constructor	Meaning
`String s=new String();`	This will create empty string object.
`String s=new String("Hello");`	This will create string object with value "Hello"
`String s=new String(StringBuffer sb);`	This will create string object with value of StringBuffer object.
`String s=new String(char[] ch)`	This will create string object from character array.

6.5.1 Operations on strings

- The **java.lang.String** class provides a lot of methods to do different operations on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Important Methods of String Class:

Table 6.2: Methods of string

Sr. No.	Method	Meaning
1.	`char chartAt(int index)`	It returns the Character at specified index position.
2.	`int comapreTo(String str)`	This Java String Method compares two string lexicographical.
3.	`int comapreToIgnoreCase(String str)`	This method compares two strings lexicographically. While comparing, this method ignores the case difference.
4.	`String concat(String str)`	This method helps to Join the user-specified string to the end of the string and return the new string.
5.	`boolean endsWith(String suffix)`	This Java string function test whether the string is ending with the suffix that we specified or not.
6.	`boolean equals(Object anObject)`	It compares this string with the user specified object.
7.	`boolean equalsIgnoreCase(String another_String)`	It compares this string with the user-specified string (another_String). Note, while comparing, it ignores the case sensitivity.

contd. ...

8.	`static String format (String format, Object.. args)`	Java string function returns a formatted string using the user specified format and arguments.
9.	`int indexOf(int ch)`	It returns the index position of the first occurrence of a specified character. It returns -1 if the specified character not found.
10.	`int indexOf(int ch, int Starting_Index)`	It returns the index position of the first occurrence of a given character. It returns -1 if the specified character not found.
11.	`int indexOf(String str)`	It returns the index position of the first occurrence of a specified substring. It returns -1 if the specified string not found.
12.	`int indexOf(String str, int Start_Index)`	It returns the index position of the first occurrence of a specified substring. It returns -1 if the specified string not found.
13.	`boolean isEmpty()`	This function returns TRUE if the length of this string is Zero.
14.	`int lastIndexOf(int ch)`	It returns the index position of the last occurrence of a specified character. It returns -1 if the given character is not found.
15.	`int lastIndexOf(int ch, int End_Index)`	Returns the index position of the last occurrence of a specified character. It returns -1 if the specified character not found.
16.	`int lastIndexOf(String str)`	It returns the index position of the last occurrence of a given character. It returns -1 if the specified character not found.
17.	`int lastIndexOf(String str, int End_Index)`	Returns the index position of the last occurrence of a specified character. It returns -1 if the given character not found.
18.	`int length()`	This Java String method returns the length of this string.

contd. ...

19.	boolean matches(String regexp)	This function tells whether this string matches a user-specified regular expression or not.
20.	String replace(char Old_Character, char New_Character)	This Java string function search for specified character and then replace all the occurrences of Old_Character with new New_Character
21.	String replace(CharSequence Old_sequence, CharSequence New_sequence)	It searches for specified substring and then replaces all the occurrences of Old_sequence substring with new New_sequence substring.
22.	String replaceAll(String regexp, String replacement)	This Java String Method searches for a specified regular expression. Then replace all the occurrences of regular expression substring with the given replacement substring.
23.	String replaceFirst(String regexp, String replacement)	It searches for a specified regular expression and then replaces the first occurrence of regular expression substring with the given replacement substring.
24.	String[ ] split(String regex)	It Split the string into an Array of Substrings based on the separator we specified.
25.	String[ ] split(String regex, int limit)	This Java String Method Split the string into an Array of Substrings based on the separator we specified.
26.	boolean startsWith(String prefix)	It tests whether the string is starting with the prefix that we specified or not.
27.	boolean startsWith(String prefix, int limit)	It tests whether the substring of this string is starting with the prefix that we specified or not.
28.	String substring(int Starting_Index)	It extracts the characters from a string based on the specified indices.
29.	String substring(int Starting_Index, int End_Index)	It extracts the characters from a string based on the specified starting position and end position.
30.	char[ ] toCharArray()	This Java String Method convert the given string into an Array of Characters or Character Array.

contd. ...

31.	int toLowerCase()	It converts the given string into Lowercase letters using default locale
32.	String toLowerCase(Locale locale)	It converts the given string into Lowercase letters, by considering the user-specified locale
33.	String toString()	This Java String Method convert the given value into String Object
34.	String toUpperCase()	It converts the given string into Uppercase letters
35.	String toUpperCase(Locale locale)	It converts the given string into Uppercase letters, by considering the host environment's current locale
36.	string trim()	This Java String Method removes the white spaces from both ends
37.	static String ValueOf(char character)	This Java string method returns the string representation of char
38.	static String ValueOf(char[] myArray)	Returns the string representation of a character array
39.	static String ValueOf(double number)	Returns the string representation of Double value
40.	static String ValueOf(float number)	Returns the string representation of Float value
41.	static String ValueOf(int number)	Java String Method returns the string representation of Integer value
42.	static String ValueOf(Object obj)	Returns the string representation of an Object.

Program 6.3: Program to demonstrate some String class methods.

```
public class test
{
    public static void main(String[] args)
    {
        String str="Master of ";
        String str1="Computer";
        str.concat("Application");
        System.out.println(str + " length = "+str.length());
        System.out.println(str + " substring(2,4) = "+str.substring(2, 4));
```

```
        System.out.println(str + " String valueOf(5)= "+String.valueOf(3));
        String fl=String.valueOf(1.1);
        System.out.println("1.1 double to string = "+fl);
        System.out.println(str +" equals to "+ str1 +"= "+str.equals(str1));
        System.out.println(str + " in uppercase "+ str.toUpperCase());
        System.out.println(str + " contains of = "+str.contains("of"));
    }
}
```

Output:

```
Master of length = 10
Master of substring(2,4) = st
Master of String valueOf(5)= 3
1.1 double to string = 1.1
Master of equals to Computer= false
Master of in uppercase MASTER OF
Master of contains of = true
```

6.6 USING COLLECTION BASES LOOP FOR STRING, TOKENIZING A STRING

6.6.1 Collection Bases Loop for String

- Iteration is one of the basic operations carried on a collection. Basically, iteration takes elements from a collection one after another, from the first element to the last one.
- There are three common ways to iterate through a Collection in Java using either while(), for() or for-each().
- While each technique will produce more or less the same results, the for-each construct is the most elegant and easy to read and write. It doesn't require an Iterator and is thus more compact and probably more efficient. It is only available since Java 5 so you can't use it if you are restrained to Java 1.4 or earlier.
- Following, the three common methods for iterating through a Collection are presented, first using a while loop, then a for loop, and finally a for-each loop. The Collection in this example is a simple ArrayList of Strings.

1. **while loop to iterate over Collections:**

```
collection<String> collection = new ArrayList<String>();
collection.add("geeta");
collection.add("neeta");
collection.add("seeta");
Iterator<string> iterator = collection.iterator();
// while loop
while (iterator.hasNext()) {
System.out.println("value= " + iterator.next());
```


2. **for loop to iterate over Collections**

```
collection<String> collection = new ArrayList<String>();
collection.add("geeta");
collection.add("neeta");
collection.add("seeta");
// for loop
for (Iterator<String> iterator = collection.iterator();
iterator.hasNext();) {
System.out.println("value= " + iterator.next());
```

3. **for-each loop to iterate over Collections**

```
collection<String> collection = new ArrayList<String>();
collection.add("geeta");
collection.add("neeta");
collection.add("seeta");
// for-each loop
for (String s: collection) {
System.out.println("value= " + s);
```

6.6.2 Tokenizing a String

- In Java, StringTokenizer is used to break a string into tokens based on provided delimiter. Delimiter can be specified either at the time of object creation or on a per-token basis.
- The StringTokenizer is a built-in class in java used to break a string into tokens. The StringTokenizer class is available inside the java.util package (java.util.StringTokenizer).
- The StringTokenizer class object internally maintains a current position within the string to be tokenized.
- The StringTokenizer class in java has the following constructor.

Table 6.3: String Tokenizer class

Sr. No.	Method	Description
1.	StringTokenizer(String str)	Creates String Tokenizer with specified string.
2.	StringTokenizer(String str, String delim)	Creates String Tokenizer with specified string and delimiter.
3.	StringTokenizer(String str, String delim, boolean returnValue)	Creates StringTokenizer with specified string, delimiter and return Value. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Table 6.4: Methods in String Tokenizer class

Sr. No.	Method	Description
1.	`String nextToken()`	It returns the next token from the String Tokenizer object.
2.	`String nextToken(String delimeter)`	It returns the next token from the String Tokenizer object based on the delimiter.
3.	`Object nextElement()`	It returns the next token from the String Tokenizer object.
4.	`boolean hasMoreTokens()`	It returns true if there are more tokens in the String Tokenizer object. Otherwise returns false.
5.	`boolean hasMoreElements()`	It returns true if there are more tokens in the String Tokenizer object. Otherwise returns false.
6.	`int countTokens()`	It returns total number of tokens in the String Tokenizer object.

Program 6.4: Java program to demonstrate String Tokenizer class.

```
import java.util.StringTokenizer;
public class demo
{
    public static void main(String args[])
    {
        StringTokenizer st = new StringTokenizer("Hello How are you?"," ");
        while (st.hasMoreTokens())
                System.out.println(st.nextToken());
    }
}
```

Output:

```
Hello
How
are
you?
```

6.7 CREATING STRINGS USING STRINGBUFFER, STRINGBUILDER

1. StringBuffer:

- StringBuffer class is used to create (mutable) modifiable String objects. This means that we can use StringBuffer to append, reverse, replace, concatenate and manipulate Strings or sequence of characters.
- The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

StringBuffer Class Constructors:

- The following are the main constructors of string builder and string buffer classes.
  1. `StringBuffer()`: This creates an empty StringBuffer with a default capacity of 16 characters.
  2. **StringBuffer(int capacity):** This creates an empty StringBuffer with a specified capacity as length.
  3. **StringBuffer(String str):** Creates a StringBuffer corresponding to specified String.

Table 6.5: StringBuffer Class Methods

Sr. No.	Method	Description
1.	`StringBuffer append(String s)`	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2.	`StringBuffer insert(int offset, String s)`	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3.	`StringBuffer replace(int startIndex, int endIndex, String str)`	It is used to replace the string from specified startIndex and endIndex.
4.	`StringBuffer delete(int startIndex, int endIndex)`	It is used to delete the string from specified startIndex and endIndex.
5.	`int capacity()`	It is used to return the current capacity.
6.	`StringBuffer reverse()`	It is used to reverse the string.
7.	`void ensureCapacity(int minimumCapacity)`	It is used to ensure the capacity at least equal to the given minimum.
8.	`char charAt(int index)`	It is used to return the character at the specified position.
9.	`int length()`	It is used to return the length of the string i.e. total number of character.
10.	`String substring(int beginIndex)`	It is used to return the substring from the specified beginIndex.
11.	`String substring(int beginIndex, int endIndex)`	It is used to return the substring from the specified beginIndex and endIndex.

Program 6.5: Java program to demonstrate StringBuffer class.

```
public class MutableStringDemo
{
   public static void main(String args[])
   {
      StringBuffer sBuffer1=new StringBuffer("Welcome");
      System.out.println("Original String is::: " + sBuffer1 + ":: having
      length " +  sBuffer1.length());
      sBuffer1.append(" All Students");
      System.out.println("Modified String after append is:: " + sBuffer1 +
      " :: having length " + sBuffer1.length());
      sBuffer1.reverse();
      System.out.println("Modified String after Reverse is:: " +
      sBuffer1);
   }
}
```

Output:

```
Original String is::: Welcome:: having length 7
Modified String after append is:: Welcome All Students :: having length 20
Modified String after Reverse is:: stnedutS llA emocleW
```

2. StringBuilder:

- Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5. It represents growable and writable character sequence.
- The main difference between the two is that StringBuffer is thread-safe implementation whereas StringBuilder is not a thread-safe. The StringBuilder is fast, it is better in terms of performance as compared to StringBuffer, but whenever thread safety is necessary, one should go for StringBuffer.

StringBuilder Constructors:

- The following are StringBuilder Constructors:

Table 6.6: StringBuilder Constructors

Sr. No.	Constructor	Description
1.	StringBuilder()	This creates an empty StringBuilder with a default capacity of 16 characters.
2.	StringBuilder(int capacity)	This creates an empty StringBuilder with a specified capacity.
3.	StringBuilder(String str)	Creates a StringBuilder corresponding to specified String.

Table 6.7: StringBuilder Methods

Sr. No.	Method	Description
1.	StringBuilder append(String s)	Append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2.	StringBuilder insert(int offset, String s)	Insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3.	StringBuilder replace(int startIndex, int endIndex, String str)	Replace the string from specified startIndex and endIndex.
4.	StringBuilder delete(int startIndex, int endIndex)	Delete the string from specified startIndex and endIndex.
5.	int capacity()	Return the current capacity.
6.	Void ensureCapacity (int minimumCapacity)	Ensure the capacity at least equal to the given minimum.
7.	StringBuilder reverse()	It is used to reverse the string.
8.	char charAt(int index)	Return the character at the specified position
9.	int length()	Return the length of the string i.e. total number of character
10.	String substring(int beginIndex)	Return the substring from the specified beginIndex.
11.	String substring(int beginIndex, int endIndex)	Return the substring from the specified beginIndex and endIndex.

Program 6.6: Program to see how to use StringBuilder in Java.

```
public class MutableStringDemo
{
   public static void main(String args[])
   {
      StringBuilder sBuilder = new StringBuilder("Welcome All Students");
      System.out.println("Original String is " + sBuilder +" having length
      " +  sBuilder.length());
      sBuilder.replace(0,11,"Dear students");
```

```
        System.out.println("After replacement String is " + sBuilder + "
        having length " + sBuilder.length());
        sBuilder.delete(0,4);
        System.out.println("After delete String is " + sBuilder);
    }
}
```

Output:

```
Original String is Welcome All Students having length 20
After replacement String is Dear Students having length 13
After delete String is Students
```

- In the above example, we have seen how to create StringBuilder class and usage of its methods.

Table 6.8: Difference between StringBuffer and StringBuilder

Sr. No.	StringBuffer	StringBuilder
1.	StringBuffer is synchronized i.e. thread safe.	StringBuilder is non-synchronized i.e. not thread safe.
2.	StringBuffer is less efficient and slower than StringBuilder.	StringBuilder is more efficient and faster than StringBuffer.
3.	StringBuffer is old, its there in JDK from very first release.	StringBuilder is introduced much later in release of JDK 1.5

Summary

- Array is a structure that holds multiple values of the same type.
- Array is Zero Based Indexing. A numbering scheme used throughout Java in which a sequence of values is indexed starting with 0 (element 0, element 1, element 2, and so on).
- The size of an array must be specified by an int value and not long or short.
- String is a class in Java.
- String class objects are immutable.
- A string is immutable, hence the length is fixed and you cannot change the value.
- Once string are initialized, we cannot change them.
- All string objects are stored in either String Constant Pool or in Heap memory.
- String constant pool is a special type of memory dedicated to store String objects only.
- String Constant pool objects are NOT eligible for garbage collection. They will remain in memory until the life of program.
- StringBuffer and StringBuilder value are mutable and you can change the value.
- StringBuffer is thread safe and StringBuilder is not synchronized. Hence, it is not thread-safe.

Check Your Understanding

1. Given the following declarations, which of the following variables are arrays?
   int[ ] a, b;
   int c, d[ ];
   (a) a (b) a and b
   (c) a, b and d (d) a, b, c and d
2. In Java, arrays are_____.
   (a) primitive data types (b) objects
   (c) interfaces (d) Strings
3. Which one of the following is a valid statement?
   (a) char []c=new char(); (b) char []c=new char[];
   (c) char []c=new char(5); (d) char []c=new char[5];
4. Which of the following declare, construct and initialize an array?
   (a) int a[ ] = { }; (b) int a[ ] = (1 ,2 ,3);
   (c) int a[ ] [ ] = { 1 ,2 ,3 }; (d) int a[ ] = { 1 ,2 ,3 };
5. Which of these constructors is used to create an empty String object?
   (a) String() (b) String(void)
   (c) String(0) (d) None of the mentioned
6. Which of these is an incorrect statement?
   (a) String objects are immutable, they cannot be changed
   (b) String object can point to some other reference of String variable
   (c) StringBuffer class is used to store string in a buffer for later use
   (d) None of the mentioned
7. Which of these method of class StringBuffer is used to find the length of current character sequence?
   (a) length() (b) Length()
   (c) capacity() (d) Capacity()
8. Which of the following loops would adequately add 1 to each element stored in values?
   (a) for (j=1; j<values.length; j++) values[j]++;
   (b) for (j=0; j<values.length; j++) values[j]++;
   (c) for (j=0; j<=values.length; j++) values[j]++;
   (d) for (j=0; j<values.length-1; j++) values[j]++;

Answers

1. (c)	2. (b)	3. (d)	4. (d)	5. (a)	6. (c)	7. (a)	8. (b)

Practice Questions

Q.I Answer the following Questions in short:

1. How to declare and initialize one dimensional array?
2. What do you mean by bound checking?
3. What are advantages and disadvantages of array?

4. Write a note on String class.
5. Using how many ways we can iterate through a Collection in Java?
6. What is the difference between StringBuffer class and StringBuilder class.
7. What does the following code do? Assume list is an array of int values, temp is some previously initialized int value, and c is an int initialized to 0.

```
for (int j = 0; j < list.length; j++)
        if (list[j] < temp)
                c++;
```

8. To swap the 3rd and 4th elements in the int array values, you would do:

```
values[3] = values[4];
values[4] = values[3];
```

True or False?

Q.II Answer the following Questions:

1. Write a note on one dimensional array in java.
2. How 2D array is declared and initialized in java?Give example.
3. Explain StringTokenizer class with the help of example.
4. List and explain any five methods of StringBuffer class.
5. List and explain any five methods of StringBuilder class.
6. Write a program that creates and initializes a four-element int array. Calculate and display the average of its values.
7. Write a program to check for in the given list of numbers whether each number is Palindrome or not?
8. Write a program to find the Largest and Smallest number in an Array.
9. Write a program to sort given array.
10. Write a program to perform addition, subtraction and multiplication operations on matrix.

Q.III Write short note on:

1. Array
2. Multi-dimensional array
3. String
4. Mutable string
5. Immutable string
6. StringBuffer
7. StringBuilder

■■■

7... Thread

Objectives...

- To understand the concept of Thread.
- To learn Need of Multi-Threaded Programming.
- To study life cycle of Thread.
- To study different types of Thread Priorities.
- To learn synchronization and inter thread communication.
- To study the Critical Factors in Thread-Deadlock.

7.1 INTRODUCTION

- Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.
- By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.
- Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.
- The main objective of multithreading is to increase execution speed of a program and decrease idle time of CPU.

7.2 UNDERSTANDING THREADS

- A smallest unit of a program which is in executive mode is called thread. Multithreading allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

- There are basically two types of a thread, Main thread and Child thread. A thread which contains main method is called main thread and all other threads are called child threads. Execution as well as termination of a program is done from the main thread.

Methods of Thread Class:

Table 7.1: Methods of Thread Class

Method	Description
`String getName()`	Retrieves the name of running thread in the current context in String format.
`void start()`	This method will start a new thread of execution by calling run() method of Thread/runnable object.
`void run()`	This method is the entry point of the thread. Execution of thread starts from this method.
`void sleep(int sleeptime)`	This method suspend the thread for mentioned time duration in argument (sleeptime in ms).
`void yield()`	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
`void join()`	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution.
`boolean isAlive()`	This method will check if thread is alive or dead.
`void interrupt()`	This method interrupts an execution of a thread.
`void setName(String name)`	This method changes the name of this thread to be equal to the argument name.
`void stop()`	This method stops an execution of a thread permanently.

7.2.1 Thread Creation

- Thread can be created as follow:
  1. By extending Thread class.
  2. By implementing Runnable interface.

1. Thread creation by extending the Thread class:

- We create a class that extends the **java.lang.Thread** class. This class overrides the *run()* method available in the Thread class. A thread begins its life inside *run()* method. We create an object of our new class and call *start()* method to start the execution of a thread. The *start()* method invokes the *run()* method on the Thread object.

Example:

```
class MyThread extends Thread
{
   MyThread()
   {
      start();
   }
   public void run()
   {
   }
}
class MainClass
{
   public static void main(String args[])
   {
      MyThread tobj=new MyThread();
   }
}
```

- Execution of a program is starts from main thread i.e. from main() method. In this, object of child thread class is created, it will call to its constructor and executes start() method which is inside the constructor, the start() method starts an execution of a thread by calling to the run() method which contains actual code of a child thread class.

Program 7.1: Write a Java program to calculate factorial of a number.

```
import java.io.*;
class MyThread extends Thread
{
   int i,n,f;
   MyThread()
   {
      f=1;
      start();
   }
   public void run()
   {
      try
      {
         sleep(1000);
         for(i=1;i<=n;i++)
         {
            f=f*i;
         }
         System.out.println ("Factorial is " + f);
      }
```

```
        catch(Exception obj)
        {
            System.out.println (obj);
        }
    }
}
class Fact
{
    public static void main(String args[]) throws Exception
    {
     BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        MyThread obj=new MyThread();
        System.out.println("Enter the Number");
        obj.n=Integer.parseInt(br.readLine());
    }
}
```

Output:

```
Enter the Number
6
Factorial is 720
```

Program 7.2: Write a Java program to display all the alphabets from A to Z after 2 seconds on screen.

```
class MyThread  extends Thread
{
    int n;
    char ch;
    MyThread()
    {
        start();
    }
    public void run()
    {
          for(ch='A';ch<='Z';ch++)
          {
              try
              {
                  sleep(2000);
                  System.out.print(ch+ " ");
              }
              catch(Exception obj)
              {
              }
          }
    }
}
```



```
class StrDemo
{
    public static void main(String args[])
    {
        MyThread obj=new MyThread();
    }
}
```

Output:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

- Above code display all the characters between A to Z after every 2 seconds.

2. Thread creation by implementing Runnable Interface:

- The Runnable interface provides a standard set of rules for the instances of classes which wish to execute code when they are active. The most common use case of the Runnable interface is when we want only to override the run method. When a thread is started by the object of any class which is implementing Runnable, then it invokes the run method in the separately executing thread.
- A class that implements Runnable interface runs on a different thread without sub classing Thread as it instantiates a Thread instance and passes itself in as the target. This becomes important as classes should not be subclasses unless there is an intention of modifying or enhancing the fundamental behavior of the class.
- It is the easiest way to create a thread by implementing Runnable. One can create a thread on any object by implementing Runnable. To implement a Runnable, one has only to implement the run method.
- To create a thread using Runnable, use the following code:

```
Runnable runnable = new MyRunnable ();
Thread thread = new Thread (runnable);
thread.start ();
```

- **Syntax:**

```
class MyThread implements Runnable
{
    Thread T1;
    MyThread()
    {
        T1 = new Thread(this, "Name Of Thread");
        T1.start ();
    }
    public void run()
    {
    }
}
class MainThread
{
    public static void main(String args[])
    {
        MyThread tobj = new MyThread ();
    }
}
```

Program 7.3: Write a Java program to display all the odd numbers between 1 to n after 1 second on to the screen.

```
import java.io.*;
class MyThread implements Runnable
{
   int i,n;
   Thread t1;
   MyThread()
   {
      t1=new Thread (this, "Odd");
      t1.start();
   }
   public void run ()
   {
      try
      {
         t1.sleep(1000);
         for(i=1;i<=n;i=i+2)
         {
            t1.sleep (1000);
            System.out.println (i);
         }
      }
      catch (Exception obj)
      {
         System.out.println (obj);
      }
   }
}
class Odd
{
   public static void main (String args[]) throws Exception
   {
      MyThread obj=new MyThread();
      BufferedReader br=new BufferedReader(new InputStreamReader
                                                        (System.in));
      System.out.println ("Enter the Number");
      obj.n=Integer.parseInt (br.readLine ());
   }
}
```

Output:

```
Enter the Number
5
1
3
5
```


Program 7.4: Write a Java program to display the details of a Thread.

```
class MyThread implements Runnable
{
   Thread t1;
   MyThread()
   {
      t1=new Thread (this, "Demo");
      t1.start ();
   }
   public void run()
   {
      System.out.println (t1);
   }
}
class ThreadDemo
{
   public static void main (String args[]) throws Exception
   {
      MyThread obj=new MyThread ();
   }
}
```

Output:

```
Thread[Demo,5,main]
```

7.3 NEEDS OF MULTI-THREADED PROGRAMMING

- **Multi-threading** is a process of executing two or more threads simultaneously to maximize utilization of CPU. Multi-threaded applications execute two or more threads concurrently. Hence, it is also known as Concurrency. Each thread runs parallel to each other. Threads doesn't allocate separate memory area, hence they save memory space. Also, context switching between threads takes less time.
- For increasing execution speed of a program and decreasing idle time of CPU multi-threaded programming is used.

7.4 THREAD LIFE CYCLE

- A smallest unit of program which is in executive mode is called Thread. In its life cycle, thread goes through the following phases:
  1. New/born
  2. Runnable
  3. Running
  4. Blocked / Sleeping (Waiting)
  5. Dead

Programmer Scheduler:

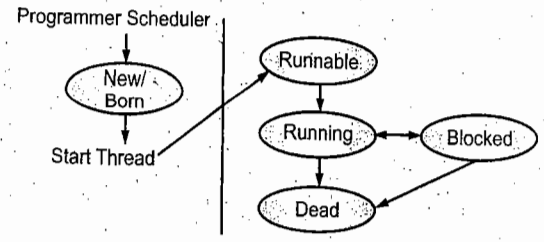


Fig. 7.1: Thread Life Cycle

- In the life cycle of thread, there are two sections:
  1. Programmer
  2. Scheduler
- If programmer inherits Thread class or implements Runnable interface in a user defined class then thread gets created, after its creation it comes in **New or Born** state.
- For starting an execution of created threads programmer should call start() method, after that all threads comes in **Runnable** state.
- In the Runnable State there is a tool provided by the operating system for the execution of the threads called scheduler.
- Scheduler stores all the threads in a data structure called pool, also called queue and assigned priority to them for their execution randomly. A thread having maximum priority scheduler selects it then that thread goes into the **Running** state and starts the execution. While its execution suppose that thread needs some resources and that resources are not available then thread goes into the **Blocked or sleep** state, after availability of the resources thread restarts its execution from a point where it has stopped.
- After successful completion of execution of a thread, it goes into the dead state. In this way thread completes its execution.

7.5 THREAD PRIORITIES

- Priority is nothing but precedence given by the operating system to the threads for their execution.
- Priority of a thread describes how early it gets executed and selected by the thread scheduler. When we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The range of the priority is between 1 to 10 and there are three constant variables which are static and used to fetch priority of a Thread. They are as following:
  1. **public static int MIN_PRIORITY:** It holds the minimum priority that can be given to a thread. The value for this is 1.
  2. **public static int NORM_PRIORITY:** It is the default priority that is given to a thread if it is not defined. The value for this is 5.
  3. **public static int MAX_PRIORITY:** It is the maximum priority that can be given to a thread. The value for this is 10.

Get and Set methods in Thread priority:

1. **public final int getPriority():** In Java, getPriority() method is in java.lang.Thread package. It is used to get the priority of a thread.
2. **public final void setPriority(int newPriority):** The setPriority(int newPriority) method is in java.lang.Thread package. It is used to set the priority to the thread. The setPriority() method throws IllegalArgumentException if the value of new priority is above minimum and maximum limit.

7.6 SYNCHRONIZING THREADS

- In Multi-threading, multiple threads are executing either simultaneously or concurrently in an area that area is called **ThreadGroup** and that execution called asynchronous execution. The limitation of asynchronous execution is: *it might be generated incorrect result.* To overcome this synchronization is used.
- Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At a time when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronized keyword creates a block of code referred to as critical section.

```
synchronized (Object)
{
}
```

- Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object's lock.

Why we need Synchronization?

- If we do not use synchronization and let two or more threads access a shared resource at the same time, it will lead to distorted results.
- Consider an example, suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.
- To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be **locked** (LOCK mode), and no other thread will be able to access or modify it until T1 returns.
- Synchronization allows to the threads for accessing common resources sequentially i.e. one by one.

Program 7.5: Write a Java program for the implementation of synchronization.

```
class Table
{
   synchronized void printTable(int n)
   {
      for(int i=1;i<=5;i++)
      {
```

```
            try
            {
                Thread.sleep(400);
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
}
class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
}
class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
}
public class TestSynchronization2
{
    public static void main(String args[])
    {
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```


Output:

```
5
10
15
20
25
100
200
300
400
500
```

7.7 INTER COMMUNICATION OF THREADS

- **Inter-thread communication** is all about allowing synchronized threads to communicate with each other.
- Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:
  1. wait()
  2. notify()
  3. notifyAll()

1. **wait():** Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
2. **notify():** Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
3. **notifyAll():** It Wakes up all threads that are waiting on this object's monitor.

Program 7.6: Write a Java program for the implementation of Interthread communication between the Threads.

```
class Customer
{
    int amount=10000;
    synchronized void withdraw(int amount)
    {
        System.out.println("going to withdraw..." +amount);
        if(this.amount<amount)
        {
            System.out.println("Less balance; waiting for deposit...");
            try
            {
                wait();
            }
```

```
            catch(Exception e)
            {
            }
        }
        this.amount-=amount;
        System.out.println ("withdraw completed... Remaining Balance Is " +
        this.amount);
    }
    synchronized void deposit(int amount)
    {
        System.out.println("going to deposit..." + amount);
        this.amount+=amount;
        System.out.println("deposit completed... ");
        System.out.println("Total Amount Is... " + this.amount);
        notify();
    }
}
class Test
{
    public static void main(String args[])
    {
        final Customer c=new Customer();
        new Thread()
        {
            public void run(){c.withdraw(15000);}
        }.start();
        new Thread()
        {
            public void run()
            {
                c.deposit(10000);
            }
        }.start();

    }
}
```

Output:

```
going to withdraw...15000
Less balance; waiting for deposit...
going to deposit...10000
deposit completed...
Total Amount Is... 20000
withdraw completed... Remaining Balance Is 5000
```


7.8 CRITICAL FACTOR IN THREAD-DEADLOCK

- Deadlock occurs when multiple threads need the same locks but obtain them in different order.
- A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

Program 7.7: Write a Java program to demonstrate deadlock condition.

```
public class TestThread
{
    public static Object Lock1 = new Object();
    public static Object Lock2 = new Object();
    public static void main(String args[])
    {
        ThreadDemo1 T1 = new ThreadDemo1();
        ThreadDemo2 T2 = new ThreadDemo2();
        T1.start();
        T2.start();
    }
    private static class ThreadDemo1 extends Thread
    {
        public void run()
        {
            synchronized (Lock1)
            {
                System.out.println("Thread 1: Holding lock 1...");
                try
                {
                    Thread.sleep(10);
                }
                catch (InterruptedException e) {}
                System.out.println("Thread 1: Waiting for lock 2...");
                synchronized (Lock2)
                {
                    System.out.println("Thread 1: Holding lock 1 & 2...");
                }
            }
        }
    }
    private static class ThreadDemo2 extends Thread
    {
        public void run()
        {
            synchronized (Lock2)
            {
                System.out.println("Thread 2: Holding lock 2...");
```

```
            try
            {
               Thread.sleep(10);
            }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");
            synchronized (Lock1)
            {
               System.out.println("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
}
```

Output:

```
Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...
```

- The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL+C.

Deadlock Solution Example:

- Let's change the order of the lock and run of the same program to see if both the threads still wait for each other –

Program 7.8: Write a Java program to demonstrate how to solve deadlock condition.

```
public class TestThread
{
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();
   public static void main(String args[])
   {
      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
   }
   private static class ThreadDemo1 extends Thread
   {
      public void run()
      {
         synchronized (Lock1)
         {
            System.out.println("Thread 1: Holding lock 1...");
```



```
            try
            {
               Thread.sleep(10);
            }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");
            synchronized (Lock2)
            {
               System.out.println("Thread 1: Holding lock 1 & 2...");
            }
         }
      }
   }
   private static class ThreadDemo2 extends Thread
   {
      public void run()
      {
         synchronized (Lock1)
         {
            System.out.println("Thread 2: Holding lock 1...");
            try
            {
               Thread.sleep(10);
            }
            catch (InterruptedException e) {}
            System.out.println ("Thread 2: Waiting for lock 2...");
            synchronized (Lock2)
            {
               System.out.println ("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
}
```

Output:

```
Thread 1: Holding lock 1...
Thread 1: Waiting for lock 2...
Thread 1: Holding lock 1 & 2...
Thread 2: Holding lock 1...
Thread 2: Waiting for lock 2...
Thread 2: Holding lock 1 & 2..
```

Summary

- In multithreading multiple threads are executing either simultaneously or concurrently.
- A smallest unit of a program in executing mode is called Thread.
- There are two types of thread
  1. **Main thread:** A thread having main method is called main thread.
  2. **Child Thread:** A thread which does not have a main method is called Child thread.
- Thread can be created in two ways:
  1. By implementing Runnable interface.
  2. By extending Thread Class.
- Thread priority is nothing but precedence given by the operating system to the threads for their execution. There are three types of Priority
  1. MAXIMUM
  2. NORMAL
  3. MINIMUM
- In multithreading multiple threads are executing either simultaneously or concurrently such execution is called Asynchronous execution.
- Synchronization is implemented by using synchronized keyword. It allows to the threads for accessing common resources one by one i.e. sequentially.
- Critical region is nothing but an area in which only one thread can execute a time and a thread which is in critical region can access only common resources.
- Interthread communication is a technique in which multiple threads communicating with each other by using the methods such as wait(), notify(), notifyAll() etc.

Check Your Understanding

1. What is multithreaded programming?
   (a) It is a process in which two different processes run simultaneously.
   (b) It is a process in which two or more parts of same process run simultaneously.
   (c) It is a process in which many different process are able to access same information.
   (d) It is a process in which a single process can access information from many sources.
2. Which of these are types of multitasking?
   (a) Process based (b) Thread based
   (c) Process and Thread based (d) None of the mentioned
3. What will happen if two thread of the same priority are called to be processed simultaneously?
   (a) Anyone will be executed first lexographically.
   (b) Both of them will be executed simultaneously.
   (c) None of them will be executed.
   (d) It is dependent on the operating system.

4. What does not prevent JVM from terminating?
   (a) Process (b) Daemon Thread
   (c) User Thread (d) JVM Thread
5. What is true about time slicing?
   (a) Time slicing is OS service that allocates CPU time to available runnable thread.
   (b) Time slicing is the process to divide the available CPU time to available runnable thread.
   (c) Time slicing depends on its implementation in OS.
   (d) Time slicing allocates more resources to thread.
6. What should not be done to avoid deadlock?
   (a) Avoid using multiple threads.
   (b) Avoid hold several locks at once.
   (c) Execute foreign code while holding a lock.
   (d) Use interruptible locks.
7. Which of the following stops execution of a thread?
   (a) Calling SetPriority () method on a Thread object.
   (b) Calling notify() method on an object.
   (c) Calling wait() method on an object.
   (d) Calling read() method on an InputStream object.
8. Which of the following will ensure the thread will be in running state?
   (a) yield() (b) notify()
   (c) wait() (d) Thread.killThread()
9. Which of these keywords are used to implement synchronization?
   (a) synchronize (b) syn
   (c) synch (d) synchronized
10. What is synchronization in reference to a thread?
    (a) It's a process of handling situations when two or more threads need access to a shared resource.
    (b) It's a process by which many thread are able to access same shared resource simultaneously.
    (c) It's a process by which a method is able to access many different threads simultaneously.
    (d) It's a method that allow too many threads to access any information.

Answers

1. (b)	2. (c)	3. (d)	4. (b)	5. (b)	6. (c)	7. (b)	8. (c)	9. (d)	10. (a)

Practice Questions

Q.I Answer the following Questions in short:

1. What is Thread?
2. What is Priority?
3. What is use of join() method?
4. What is multithreaded programming?
5. What is ThreadGroup class?
6. What is use of scheduler for the execution of a thread?
7. What is use of isAlive() method?
8. What is asynchronous execution?
9. What is critical region?
10. What is deadlock how to resolve it?

Q.II Answer the following Questions:

1. Explain different types of thread with an example.
2. Explain different types of priorities with an example.
3. Explain life cycle of thread in details.
4. Differentiate between sleep () and interrupt () method.
5. Explain synchronization with an example.
6. How interthread communication is implemented in multithreading? Explain with an example.
7. Write a Java program to display all the alphabets from a given string after 1 second.
8. Write a Java program in AWT to blink a text on to the frame.
9. Explain Thread creation technique using Runnable interface with an example.
10. Write a Java program to scroll the text on the frame from left to right continuously.

Q.III Write short note on:

1. Deadlock
2. Synchronization
3. Objective of Multithreading
4. InterruptedException
5. run() method.

■■■

8... A Collection of Useful Classes

Objectives...

- To learn Date class, Time class, Scanner class, regex class with different symbols.
- To understand about java.io package.
- To study different types of stream and its working.
- To study File handling.

8.1 INTRODUCTION

- In this chapter, we will learn java.util package which defines a number of useful classes, primarily collections classes that are useful for working with groups of objects.
- The java.util package contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a String Tokenizer, Regular expression, Scanner).
- The java.io package provides for system input and output through data streams, serialization and the file system.
- In this chapter, we will learn different classes, interfaces and methods of java.util and java.io package with examples.

8.2 UTILITY METHODS FOR ARRAYS

- Java Array class was introduced in Java 1.2 as a utility class to perform various common operations on arrays in java. This class is part of java collections framework.
- Arrays class contains overloaded methods that support primitive data types.
- The Arrays class is in java.util package provides a set of static methods for operating an arrays. You have methods for sorting and searching arrays, as well as methods for comparing two arrays of elements of a basic type.
- You also have methods for filling arrays of elements with a given value. An Array class is bundled with numerous utility methods to perform common tasks like copying, sorting, searching items, indexing, etc.
- Using Arrays, we can create, compare, sort, search, stream, and transform arrays.

Table 8.1: Utility Methods for Arrays

Return type	Method and description
static List	`asList(Object[] a)` Returns a fixed-size List backed by the specified array.
static int	`binarySearch(int[] a, int key)` Searches the specified array of int for the specified value using the binary search algorithm. **Note:** • If input list is not sorted, the results are undefined. • If search key not present it return –1. • If there are duplicates, there is no guarantee which one will be found.
static boolean	`equals(int[] a, int[] a2)` Returns true, if the two specified arrays are *equal* to one another.
static void	`fill(int[] a, int val)` Assigns the specified int value to each element of the specified array of int.
static void	`fill(int[] a, int fromIndex, int toIndex, int val)` Assigns the specified int value to each element of the specified range of the specified array of int.
static void	`sort(int[] a)` Sorts the specified array of int into ascending numerical order.

Program 8.1: Java program to demonstrate sorting methods of an array.

```
import java.util.Arrays;
public class Main
{
    public static void main(String[] args)
    {
       String[] states = {"Gujarat", "Assam", "Maharashtra", "Bihar",
       "Goa"};
       String[] str = {"geeta", "seeta", "neeta"};
       int a[]={ 10,20,30};
       Arrays.sort(states);
       System.out.println("After sorting: " + Arrays.toString(states));
       Arrays.fill(str, null);
       System.out.println("After fill: " + Arrays.toString(str));
       System.out.println(" Binary search answer is
                                          =" +Arrays.binarySearch(a,20));
    }
}
```


Output:

```
After sorting: [Assam, Bihar, Goa, Gujarat, Maharashtra]
After fill: [null, null, null]
Binary search answer is =1
```

8.3 OBSERVABLE AND OBSERVER OBJECTS

- Observers and observables are part of the MVC (Model-View-Controller) framework. The Java language supports the MVC architecture with two classes:
  1. **Observer:** Any object that wishes to be notified when the state of another object changes.
  2. **Observable:** An object whose state may be of interest, and in whom another object may register an interest.
- They are suitable for any system wherein objects need to be automatically notified of changes that occur in other objects.
- All the objects which are the instances of the sub class Observable, maintains a list of observers. At the time of updating an Observable object, it invokes the update() method of each of its observers for notifying the observers which has changed its state.
- The objects implement the Observer interface, which observes Observable objects. Whenever an observable object changes its state, its corresponding observer classes are notified.
- Observable is implemented as a class which includes methods for managing Observer lists and notifying Observers. When there is a change in an observable instance, an application calling the Observable's notifyObservers() method causes all of its observers to be notified of the change by a call to their update method.

Program 8.2: Java program for the implementation of Observable class and Observer interface.

```
import java.util.*;
import java.util.Observable;
import java.util.Observer;
public class Demo extends Observable
{
   private String message;
   public String getMessage()
   {
      return message;
   }
   public void changeMessage( String message )
   {
      this.message = message;
      setChanged();
      notifyObservers( message );
   }
```

```
    public static void main( String [] args )
    {
        Demo ob = new Demo();
        bank b1 = new bank();
        bank b2 = new bank();
        ob.addObserver( b1 );
        ob.addObserver( b2 );
        ob.changeMessage("Successfully");
    }
}
class bank implements Observer
{
    public void update(Observable o, Object arg)
    {
        System.out.println( "Observer is added: " + arg );
    }
}
```

Output:

```
Observer is added: Successfully
Observer is added: Successfully
```

8.4 DATE AND TIMES

- The Date is a built-in class in java. It is used to work with date and time. The Date class is available inside the **java.util** package.
- The class Date represents a specific instant in time, with millisecond precision. The Date class implements Serializable, Cloneable and Comparable interface.
- Java provides a class called a **SimpleDateFormat** that allows you to format and parse dates in the as per your requirements.

Table 8.2: Constructor of Date class

Sr. No.	Constructor	Description
1	`Date( )`	It creates a Date object that represents current date and time.
2	`Date(long milliseconds)`	It creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.

Table 8.3: Methods of Date class

Sr. No.	Method	Description
1.	`long getTime()`	It returns the time represented by this date object.
2.	`boolean after(Date date)`	It returns true, if the invoking date is after the argumented date.

contd. ...

3.	`boolean before(Date date)`	It returns true, if the invoking date is before the argumented date.
4.	`Date from(Instant instant)`	It returns an instance of Date object from Instant Date.
5.	`void setTime(long time)`	It changes the current Date and Time to given Time.
6.	`int compareTo(Date date)`	It compares current Date with given Date.
7.	`boolean equals(Date date)`	It compares current Date with given Date for equality.
8.	`int hashCode()`	It returns the hash code value of the invoking Date object.
9.	`Instant toInstant()`	It converts current Date into Instant object.
10.	`Object clone( )`	It duplicates the invoking Date object.

Program 8.3: Java program to display system's Date and Time.

```
import java.util.Date;
public classDate_Demo
{
    public static void main(String args[])
    {
      // Instantiate a Date object
          Date ob=new Date();
          System.out.println (ob.toString());
    }
}
```

Output:

```
Thu Jul 30 10:45:37 UTC 2020
```

8.5 USING SCANNER

- The Scanner class is a built-in class used for read the input from the user. The Scanner class is found in the **java.util** package.
- The Scanner class implements **Iterator** interface. The Scanner object breaks its input into tokens using a delimiter pattern, the default delimiter is whitespace.
- To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class.

Table 8.4: Methods of scanner class to read data of various types

Sr. No.	Methods with Description
1.	`String next()` It reads the next complete token from the invoking scanner.
2.	`String next(Pattern pattern)` It reads the next token if it matches the specified pattern.

contd. ...

3.	**String next(String pattern)** It reads the next token if it matches the pattern constructed from the specified string.
4.	**Boolean nextBoolean()** It reads a boolean value from the user.
5.	**byte nextByte()** It reads a byte value from the user.
6.	**double nextDouble()** It reads a double value from the user.
7.	**float nextFloat()** It reads a floating-point value from the user.
8.	**int nextInt()** It reads an integer value from the user.
9.	**long nextLong()** It reads a long value from the user.
10.	**short nextShort()** It reads a short value from the user.
11.	**String nextLine()** It reads a string value from the user.
12.	**Boolean hasNext()** It returns true if the invoking scanner has another token in its input.
13.	**void remove()** It is used when remove operation is not supported by this implementation of Iterator.
14.	**void close()** It closes the invoking scanner.

Program 8.4: Java program to accept details of student (Roll no, Name, Percentage) using scanner class and display them.

```
import java.util.Scanner;
public class Scanner_demo
{
   public static void main(String[] args)
   {
      Scanner scan = new Scanner(System.in);
      System.out.print("Enter your roll no. ");
      int roll = scan.nextInt();
```



```
      System.out.print("Enter any name: ");
      String name = scan.next();
      System.out.print("Enter your percentage: ");
      double per = scan.nextDouble();
      System.out.println(" Students details are: ");
      System.out.println("Roll No:" + roll);
      System.out.println("Name:" + name);
      System.out.println("Percentage:" + per);
   }
}
```

Output:

```
Enter your roll no. 10
Enter any name: Sandesh
Enter your percentage: 84.63
Students details are:
Roll No:10
Name:Sandesh
Percentage:84.63
```

8.6 REGULAR EXPRESSION

- The Regular Expression is an API to define a pattern for searching or manipulating strings. The abbreviation for regular expression is *regex*. It is widely used to define the constraint on strings such as password and email validation.
- The search pattern can be anything from a simple character, a fixed string or a complex expression containing special characters describing the pattern. The pattern defined by the regex is applied on the text from left to right.
- Regular expressions are used for defining String patterns that can be used for searching, manipulating and editing a text.
- Regular Expressions are provided under **java.util.regex** package which consists of 1 interface and 3 classes.

Interface in java.util.regex:

- **Java.util.regex.MatchResult interface:** This interface contains query methods used to determine the results of a match against a regular expression. The match boundaries, groups and group boundaries can be seen but not modified through a MatchResult.
- The java.util.regex package primarily consists of the following three classes:
  1. **java.util.regex.Pattern Class:** Defines a pattern (to be used in a search).
  2. **java.util.regex.Matcher Class:** Used for performing match operations on text using patterns.
  3. **PatternSyntaxException Class:** PatternSyntaxException is thrown if the regular expression syntax is not correct.

Table 8.5: Symbols used in Regular Expression

Subexpression	Matches
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character except newline. Using m option allows it to match the newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
\A	Beginning of the entire string.
\z	End of the entire string.
\Z	End of the entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of the preceding expression.
re+	Matches 1 or more of the previous thing.
re?	Matches 0 or 1 occurrence of the preceding expression.
re{ n}	Matches exactly n number of occurrences of the preceding expression.
re{ n,}	Matches n or more occurrences of the preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of the preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers the matched text.
(?: re)	Groups regular expressions without remembering the matched text.
(?> re)	Matches the independent pattern without backtracking.
\w	Matches the word characters.
\W	Matches the nonword characters.
\s	Matches the whitespace. Equivalent to [\t\n\r\f].
\S	Matches the nonwhitespace.
\d	Matches the digits. Equivalent to [0-9].
\D	Matches the nondigits.
\A	Matches the beginning of the string.
\Z	Matches the end of the string. If a newline exists, it matches just before newline.
\z	Matches the end of the string.
\G	Matches the point where the last match finished.
\n	Back-reference to capture group number "n".
\b	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
\B	Matches the nonword boundaries.
\n, \t	Matches newlines, carriage returns, tabs.
\Q	Escape (quote) all characters up to \E.
\E	Ends quoting begun with \Q.

Table 8.6: Methods for matching pattern

Sr. No.	Method and Description
1.	`public boolean lookingAt()` Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2.	`public boolean find()` Attempts to find the next subsequence of the input sequence that matches the pattern.
3.	`public boolean find(int start)` Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4.	`public boolean matches()` Attempts to match the entire region against the pattern.

Program 8.5: Program to count the number of times the word "cat" appears in the input string.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class regex_demo
{
   private static final String REGEX = "\\bcat\\b";
   private static final String INPUT = "cat cat cat cattie cat";
   public static void main( String args[])
   {
      Pattern p = Pattern.compile(REGEX);
      Matcher m = p.matcher(INPUT);
      int count = 0;
      while(m.find())
      {
         count++;
      }
      System.out.println ("Total count = "+count);
   }
}
```

Output:

```
Total count = 4
```

8.7 INPUT/OUTPUT OPERATION IN JAVA (java.io PACKAGE)

- The package **java.io** contains the classes that handle fundamental input and output operations in Java. The I/O classes can be grouped as follows:
  - Classes for reading input from a stream of data.
  - Classes for writing output to a stream of data.
  - Classes that manipulate files on the local filesystem.
  - Classes that handles object serialization.
- I/O in Java is based on streams. A stream represents a flow of data or a channel of communication.

Following is a list of important commonly used Java I/O classes under java.lang package:

1. **java.io.FileOutputStream Class:** FileOutputStream creates an OutputStream that you can use to write bytes to a file. It implements the AutoCloseable, Closeable, and Flushable interfaces.
2. **java.io.FileInputStream Class:** FileInputStream class creates an InputStream that you can use to read bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.
3. **java.io.ByteArrayOutputStream Class:** ByteArrayOutputStream is an implementation of an output stream that uses a byte array as the destination.
4. **java.io.ByteArrayInputStream Class:** ByteArrayInputStream is an implementation of an input stream that uses a byte array as the source.
5. **java.io.BufferedWriter Class:** BufferedWriter class writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
6. **java.io.BufferedReader Class:** Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
7. **java.io.BufferedOutputStream Class:** Java BufferedOutputStream class is used for buffering an output stream. It internally uses a buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.
8. **java.io.BufferedInputStream Class:** Java BufferedInputStream class is used to read information from the stream. It internally uses a buffer mechanism to make the performance fast.
9. **java.io.FileWriter Class:** FileWriter creates a Writer that you can use to write to a file. FileWriter is convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
10. **java.io.FileReader Class:** The FileReader class creates a Reader that you can use to read the contents of a file. FileReader is meant for reading streams of characters. For reading streams of raw bytes,consider using a FileInputStream.
11. **java.io.DataOutStream Class:** Java DataOutputStream class allows an application to write primitive Java data types to the output stream in a machine-independent way.
12. **java.io.DataInputStream Class:** Java DataInputStream class allows an application to read primitive data from the input stream in a machine-independent way. Java application generally uses the data output stream to write data that can later be read by a data input stream.

13. **java.io.ObjectOutputStream Class:** An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
14. **java.io.ObjectInputStream Class:** ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.

8.8 STREAMS AND THE NEW I/O CAPABILITIES

8.8.1 Understanding Streams

- In Java, all input and output is handled through Input/Output streams. A stream means continuous flow of data.
- A stream is composed of bytes. Java encapsulates Stream under **java.io** package. In java, a stream is a logical container of data that allows us to read from and write to it.
- A stream can be linked to a data source, or data destination, like a console, files or network connection by java I/O system.
- The stream-based I/O operations are faster than normal I/O operations.

8.8.2 The Classes for Input and Output

- Java defines two types of streams. They are:

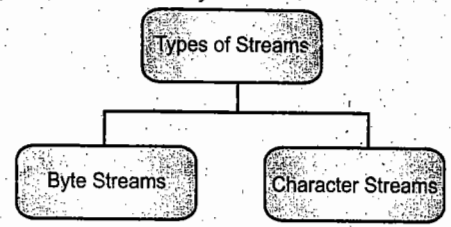


Fig. 8.1: Types of Streams

1. **Byte Stream:** It provides a convenient means for handling input and output of byte. The following picture shows the classes used for byte stream operations.

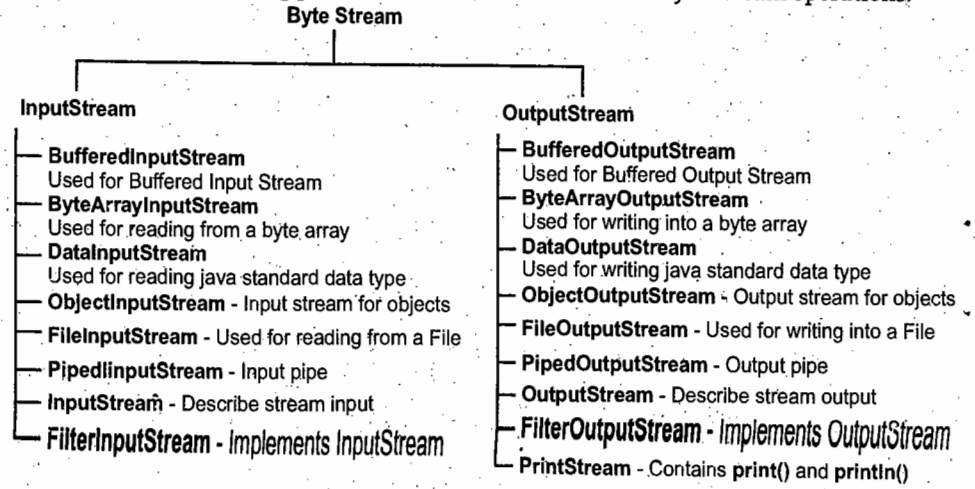


Fig. 8.2: Byte stream

- Let's consider following example, that illustrates reading data using BufferedInputStream. BufferedInputStream class is used to read data from the console. The BufferedInputStream class use a method read( ) to read a value from the console, or file, or socket.

Program 8.6: Java program to demonstrate how to read a character using BufferedInputStream and display it.

```
import java.io.*;
public class Demo
{
    public static void main(String[] args)
    {
        try
        {
            BufferedInputStream br = new BufferedInputStream(System.in);
            System.out.print("Enter any character: ");
            char ch = (char)br.read();
            System.out.println("Inputted char is '" + ch);
        }
        catch (IOException e)
        {
            System.out.println("Can't read character");
        }
    }
}
```

Output:

```
Enter any character: M
Inputted char is 'M
```

2. **Character Stream:** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized. The following picture shows the Character stream.

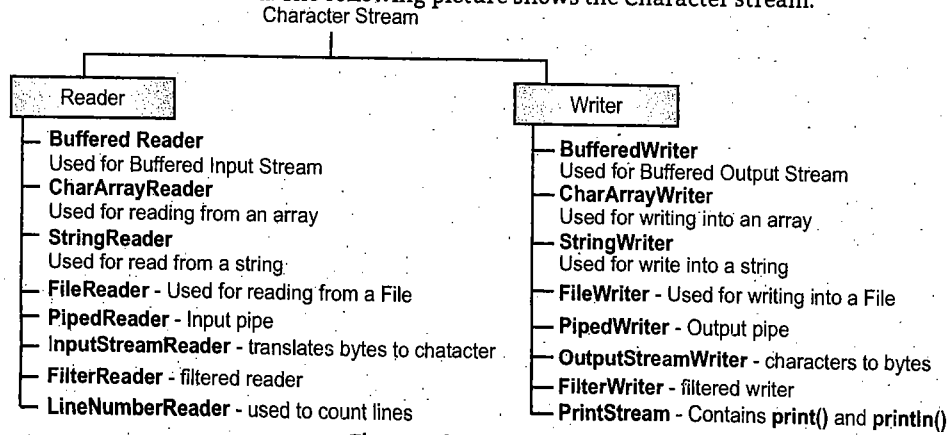


Fig. 8.3: Character Stream

- The read() method is used with BufferedReader object to read characters and the write() method is used with BufferedWriter object to write characters.
- Let's consider following example, that illustrates reading data using BufferedReader class. It reads the data from the console or file or socket using read () method.

Program 8.7: Java program to demonstrate how to read string using InputStreamReader and display it.

```
import java.io.*;
public class Demo
{
    public static void main(String[] args)
    {
        try
        {
            InputStreamReader ir = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(ir);
            String name = "";
            System.out.print("Enter your name: ");
            name =br.readLine();
            System.out.println("Hello, " + name);
        }
        catch (IOException e)
        {
            System.out.println("Can't read name");
        }
    }
}
```

Output:

```
Enter your name: Avani
Hello, Avani
```

8.8.3 The Standard Streams

- Every Java program creates 3 streams automatically. All these streams are attached with the console.

1. **System.out:** Standard output stream. Standard output is a stream to which a program writes its output data.
2. **System.in:** Standard input stream. Standard input is a stream from which a program reads its input data.
3. **System.err:** Standard error stream. Standard error is another output stream typically used by programs to output error messages or diagnostics.

Example:

```
System.out.println("simple message");
System.err.println("error message");
int a=System.in.read();//returns ASCII code of 1st character
```

8.9 WORKING WITH FILE OBJECT

- Java file class is mainly used for creating and listing new directories, it also has many methods to list the common attributes of files and directories. It is also used for file searching, file deletion, renaming a file, etc. This is an abstract representation of a physical file on the system. In short, we can say this file class is a part of **java.io** package.

8.9.1 File I/O Basics

- The File class of the java.io package is used to perform various operations on files and directories.
- To create an object of File, we need to import the java.io.File package first. Once we import the package, here is how we can create objects of file.

```
File f = new File("abc");
```

- It is created by passing a string with the name of a file, string, or another file object.

Table 8.7: The File class constructors

Constructor	Description
`File(File parent, String child)`	Creates a new File instance from a parent abstract pathname and a child pathname string.
`File(String pathname)`	Creates a new File instance by converting the given pathname string into an abstract pathname.
`File(String parent, String child)`	Creates a new File instance from a parent pathname string and a child pathname string.
`File(URI uri)`	Creates a new File instance by converting the given file: URI into an abstract pathname.

Table 8.8: The File class Methods

Sr. No.	Method and Description
1.	`public String getName()` Returns the name of the file or directory denoted by this abstract pathname.
2.	`public String getParent()` Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.

contd. ...

3.	`public File getParentFile()` Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
4.	`public String getPath()` Converts this abstract pathname into a pathname string.
5.	`public boolean isAbsolute()` Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise.
6.	`public String getAbsolutePath()` Returns the absolute pathname string of this abstract pathname.
7.	`public boolean canRead()` Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
8.	`public boolean canWrite()` Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
9.	`public boolean exists()` Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise.
10.	`public boolean isDirectory()` Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
11.	`public boolean isFile()` Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise.
12.	`public long lastModified()` Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.

contd. ...

13.	`public long length()` Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
14.	`public boolean createNewFile() throws IOException` Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.
15.	`public boolean delete()` Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.
16.	`public void deleteOnExit()` Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
17.	`public String[] list()` Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
18.	`public String[] list(FilenameFilter filter)` Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
20.	`public File[] listFiles()` Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
21.	`public File[] listFiles(FileFilter filter)` Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
22.	`public boolean mkdir()` Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise.
23.	`public boolean mkdirs()` Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.

contd. ...

24.	`public boolean renameTo(File dest)` Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise.
25.	`public boolean setLastModified(long time)` Sets the last-modified time of the file or directory named by this abstract pathname. Returns true if and only if the operation succeeded; false otherwise.
26.	`public boolean setReadOnly()` Marks the file or directory named by this abstract pathname so that only read operations are allowed. Returns true if and only if the operation succeeded; false otherwise.
27.	`public static File createTempFile(String prefix, String suffix, File directory) throws IOException` Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. Returns an abstract pathname denoting a newly-created empty file.
28.	`public static File createTempFile(String prefix, String suffix) throws IOException` Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. Invoking this method is equivalent to invoking createTempFile(prefix, suffix, null). Returns abstract pathname denoting a newly-created empty file.
29.	`public int compareTo(File pathname)` Compares two abstract pathnames lexicographically. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
30.	`public int compareTo(Object o)` Compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
31.	`public boolean equals(Object obj)` Tests this abstract pathname for equality with the given object. Returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.
32.	`public String toString()` Returns the pathname string of this abstract pathname. This is just the string returned by the getPath() method.

Program 8.8: Java program to change the file permissions.

```
import java.io.*;
public class modify_file_permission
{
   public static void main(String[] args)
   {
      // creating a new file instance
      File f = new File("C:\\Users\\DELL\\Desktop\\sample.txt");
      if(f.exists() == true)     // check if file exists
      {
         // changing the file permissions
         f.setExecutable(true);
         f.setReadable(true);
         f.setWritable(false);
         System.out.println("File permissions has changed.");
         System.out.println("Is Executable? " + file.canExecute());
         System.out.println("Is Readable? " + file.canRead());
         System.out.println("Is Writable? "+ file.canWrite());
      }
      else
      {
         System.out.println("File not found.");
      }
   }
}
```

Output:

```
File permissions has changed.
Is Executable? true
Is Readable? true
Is Writable? false
```

8.9.2 Reading and Writing to Files

- FileWriter and FileReader classes are very frequently used to write and read data from text files (they are character stream classes).

FileReader class:

- FileReader uses for reading the data, which are in the form of characters, and it is done from a 'text' file. It returns data in byte format like **FileInputStream** class.
- Java FileReader uses for particularly reading streams of character. For reading streams of raw bytes, FileInputStream can use.

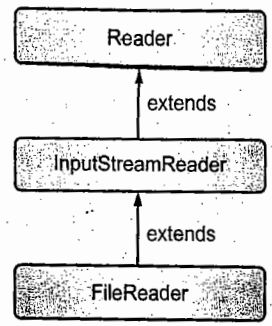


Fig. 8.4: FileReader Class

Constructors:

- **`FileReader (File file)`:** It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
- **`FileReader (String file)`:** It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Methods of FileReader class:

- **`int read()`:** This method reads a single character. It returns -1 at the end of file.
- **`void close()`:** It is used to close the FileReader class.

Program 8.9: The following program display contents of file 'temp.txt' on the console.

```
import java.io.FileReader;
public class FileReaderDemo
{
   public static void main(String args[])throws Exception
   {
      FileReader fr=new FileReader("D:\\temp.txt");
      int i;
      while((i=fr.read())!=-1)
      System.out.print((char)i);
      fr.close();
   }
}
```

- In the above example, we have created an object of FileReader named *'fr'*. It is now linked with the *temp.txt* file. To read all data from the *temp.txt* file, we have used the *read()* method of FileReader. So above program display all contents of file *temp.txt*

FileWriter class:

- The FileWriter class of the java.io package can be used to write data (in characters) to files. We use the FileWriter class together with its write() method to write some text to the file. It extends the OutputStreamWriter class.

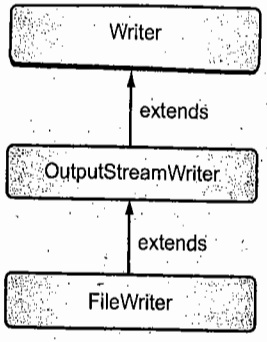


Fig. 8.5: FileWriter class

FileWriter Constructors:

- `FileWriter(File file)`: This constructor creates a FileWriter object given a File object.
- `FileWriter (String fileName)`: This constructor creates a FileWriter object, given a file name.
- `FileWriter(String fileName, boolean append)`: This constructor creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.
- `FileWriter(File file, boolean append)`: This constructor creates a FileWriter object given a File object with a boolean indicating whether or not to append the data written.

Methods of FileWriter:

- The FileWriter class provides implementations for different methods present in the Writer class.
  - `write()`: Writes a single character to the writer.
  - `write(char[] array)`: Writes the characters from the specified array to the writer.
  - `write(String data)`: Writes the specified string to the writer.

Program 8.10: Demonstration of FileWriter class to write data to a File.

```
import java.io.FileWriter;
public class FileWriterDemo
{
   public static  void  main(String args[])
   {
      String str = "This is the data to be write in the output file";
      try
      {
         FileWriter fw = new FileWriter("output.txt");
         fw.write(str);
         fw.close();
      }
```



```
      catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

- In the above example, we have created an object of FileWriter named *'fw'*. It is now linked with the *output.txt* file. To write data which *str* contains into *output.txt* file, we have used the *write()* method of FileWriter. So above program writes " *Line to write: Best Luck* " into the file *output.txt*.

Program 8.11: Demonstration of how to read data from one file using FileInputStream and write into another file using FileOutputStream class.

```
import java.io.*;
class FileCopy
{
   public static void main(String args[]) throws Exception
   {
      int b;
      FileInputStream fin=new FileInputStream(args[0]);
      FileOutputStream fout=new FileOutputStream(args[1]);
      while((b=fin.read())!=-1)
      {
         fout.write(b);
      }
      fclose(fin);
      fclose(fout);
   }
}
```

- It will copy the data from one file into the another file.

8.9.3 Buffer and Buffer Management

- Buffer classes are the foundation upon which java.nio is built.
- A Buffer object can be termed as container for a fixed amount of data. It acts as a holding tank, or temporary staging area, where data can be stored and later retrieved.
- Buffers are defined inside **java.nio** package. Channels are actual portals through which I/O transfers take place; and buffers are the sources or targets of those data transfers.
- For outward transfers, data you want to send is placed in a buffer, which is passed to an out channel. For inward transfers, a channel stores data in a buffer you provide and then data is copied from buffer to in channel.

NIO Buffer Characteristics:

1. Buffers are the basic building blocks of Java NIO.
2. Buffers provide a fixed size container to read and write data.
3. Every buffer is readable, but only chosen buffers are writable.
4. Buffers are endpoints of Channels.
5. In a read-only buffer content is not mutable but, its mark, position and limit are mutable.
6. By default buffers are not thread-safe.

Types of Buffer:

1. There is a buffer type for each primitive type. All the buffer classes implement the Buffer interface. The most used buffer type is ByteBuffer.
2. Here is the list of core Buffer implementations in java NIO:

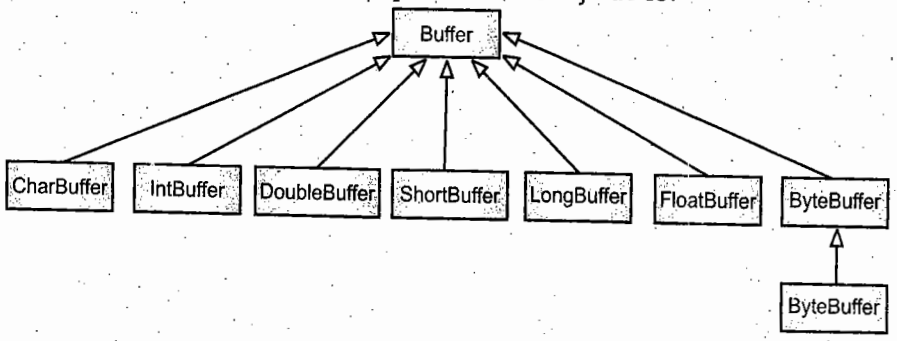


Fig. 8.6: Buffer Types

- As you can see, these Buffer types represent different data types. In other words, they let you work with the bytes in the buffer as char, short, int, long, float or double instead.

Allocating a Buffer:

- To obtain a Buffer object you must first allocate it. Every Buffer class has an allocate() method that does this.

Example:

```
ByteBuffer buf = ByteBuffer.allocate(28);
```

Above statement allocates a ByteBuffer with capacity of 28 bytes.

Writing Data to a Buffer:

- Create a buffer by allocating a size. Buffer has a method allocate(size) and returns a Buffer instance. `ByteBuffer byteBuffer = ByteBuffer.allocate(512).`
- Put data into buffer: `byteBuffer.put((byte) 0xfff);`

Reading Data from a Buffer:

- Create a buffer by allocating a size. Buffer has a method allocate(size) and returns a Buffer instance. `ByteBuffer byteBuffer = ByteBuffer.allocate(512);`

- Flip the buffer to prepare for read operation, `byteBuffer.flip();`
- Then a buffer can be read as:

```
int numberOfBytes = fileChannel.read(byteBuffer);
```

- Read from the buffer, `char c = (char)byteBuffer.get();`

8.9.4 Read/Write Operations with File Channel

- Channels are the second major innovation of java.nio after buffers which we have learned in previous topic 8.9.3. Channels provide direct connections to I/O services.
- A Channel is a medium that transports data efficiently between byte buffers and the entity on the other end of the channel (usually a file or socket).
- The FileChannel class provided in the Java NIO library. A Java NIO FileChannel is a channel that is connected to a file. It gives an alternate way to read data from a file, it provides better performance than InputStream or OutputStream.
- A FileChannel cannot be set into non-blocking mode. It always runs in blocking mode.

Advantages of FileChannel:

1. Reading and writing at a specific position in a file.
2. Loading a section of a file directly into memory, that can be more efficient.
3. We can transfer file data from one channel to another at a faster rate.
4. We can lock a section of a file to restrict access by other threads.
5. To avoid data loss, we can force writing updates to a file immediately to storage.

How to open FileChannel?

- You cannot open a FileChannel directly. You need to obtain a FileChannel via an InputStream, OutputStream, or a RandomAccessFile. Following statements shows how to open a FileChannel via a RandomAccessFile:

```
RandomAccessFile file = new RandomAccessFile("data.txt", "rw");
FileChannel fchannel = file.getChannel();
```

Reading Data from a FileChannel:

- To read data from a FileChannel, use method read().
- Here is an example:

```
Byte Bufferbuf = ByteBuffer.allocate(48);
Int bytesRead = fchannel.read(buf);
```

- First a Buffer is allocated. The data read from the FileChannel is read into the Buffer.
- Second the *FileChannel.read()* method is called. This method reads data from the FileChannel into the Buffer. The int returned by the *read()* method tells how many bytes were written into the Buffer. If -1 is returned, the end-of-file is reached.

Writing Data to a FileChannel:

- *Writing data to a FileChannel is done using the FileChannel.write()* method, which takes a Buffer as parameter. Here is an example:

```
String newData = "New String to write to file is ....." ;
Byte Bufferbuf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
while(buf.hasRemaining())
fchannel.write(buf);
```

- Notice, how the *FileChannel.write()* method is called inside a while-loop. There is no guarantee of how many bytes the *write()* method writes to the FileChannel. Therefore we repeat the *write()* call until the Buffer has no further bytes to write.

8.10 SERIALIZING OBJECT

- In Java, the Serialization is the process of converting an object into a byte stream so that it can be stored on to a file, or memory, or a database for future access.
- The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object.
- Using serialization and deserialization, we can transfer the Object Code from one Java Virtual machine to another.
- The Serialization is achieved with the help of interface Serializable. The class whose object needs to be serialized must implement the Serializable interface.
- We use the ObjectOutputStream class to write a serialized object to a destination. The ObjectOutputStream class provides a method writeObject() to serializing an object.
- We use the following steps to serialize an object.

  Step 1: Define the class whose object needs to be serialized; it must implement Serializable interface.

  Step 2: Create a file reference with file path using FileOutputStream class.

  Step 3: Create reference to ObjectOutputStream object with file reference.

  Step 4: Use writeObject(object) method by passing the object that wants to be serialized.

  Step 5: Close the FileOutputStream and ObjectOutputStream.

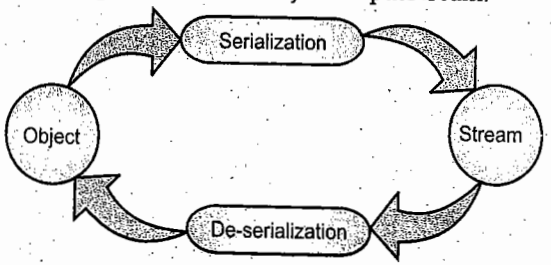


Fig. 8.7: Object Serialization

Summary

- Arrays class is in java.util provides a set of static methods for operating an arrays.
- binaryserach() method uses binary search to find a specified value. This method must be applied to sorted arrays.
- Observer is any object that wishes to be notified when the state of another object changes.
- Observable is any object whose state may be of interest, and in whom another object may register an interest.
- Observable object invokes the update() method.
- Observable's notifyObservers method Serialization is a process of writing the state of an object to a byte stream.
- The Date, Scanner class are found in the java.util package.
- Java performs I/O through Streams.
- In general, a stream means continuous flow of data.
- Java defines two types of streams, Byte Stream and Character Stream
- Byte Stream provides a convenient means for handling input and output of byte.
- Character Stream provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.
- InputStream & OutputStream are designed for byte stream.
- Reader and writer are designed for character stream.
- There are three standard streams, all of which are managed by the java.lang.System class.
- Standard input (System.in)-Used for program input, typically reads input entered by the user.
- Standard output (System.out)-Used for program output, typically displays information to the user.
- Standard error (System.err)-Used to display error messages to the user.
- FileWriter and FileReader classes are used to write and read data from text files (they are Character Stream classes).
- FileReader class uses read() to read characters from a file.
- FileInputStream class is used to read and write bytes in a file.
- A buffer is a linear, finite sequence of elements of a specific primitive type.
- ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer are types of buffer.
- The reverse operation of serialization is called deserialization where byte-stream is converted into an object.

Check Your Understanding

1. Which of this method is used to make all elements of an equal to specified value?
   (a) add() (b) fill()
   (c) all() (d) set()
2. Which of these method of Array class is used sort an array or its subset?
   (a) binarysort() (b) bubblesort()
   (c) sort() (d) insert()
3. Which of these methods calls update() method?
   (a) notify() (b) observeObject()
   (c) updateObserver() (d) notifyObserver()
4. Which of these methods is used to notify observer the change in observed object?
   (a) update() (b) notify()
   (c) check() (d) observed()
5. What is the use of Observable class?
   (a) It is used to create global subclasses
   (b) It is used to create classes that other part of the program can observe
   (c) It is used to create classes that can be accessed by other parts of program
   (d) It is used to create methods that can be accessed by other parts of program
6. Which is the correct syntax to declare Scanner class object?
   (a) Scanner objectName= Scanner();
   (b) Scanner objectName= new Scanner();
   (c) Scanner objectName= Scanner(System.in);
   (d) Scanner objectName= new Scanner(System.in);
7. Which Scanner class method is used to read integer value from the user?
   (a) next() (b) nextInteger()
   (c) nextInt() (d) readInt()
8. Which of the following is not a class of java.util.regex?
   (a) Pattern class (b) matcher class
   (c) PatternSyntaxException (d) Regex class
9. Object of which class is used to compile regular expression?
   (a) Pattern class (b) Matcher class
   (c) PatternSyntaxException (d) None of the mentioned
10. Which of these methods are used to read in from file?
    (a) get() (b) read()
    (c) scan() (d) readFileInput()

11. Which of these class contains the methods used to write in a file?
    (a) FileStream (b) FileInputStream
    (c) BufferedOutputStream (d) FileBufferStream
12. Which of these class is used to read characters in a file?
    (a) FileReader (b) FileWriter
    (c) FileInputStream (d) InputStreamReader
13. The ____ package contains a large number of stream classes that provide capabilities for processing all types of data.
    (a) java.awt (b) java.io
    (c) java.util (d) java.net
14. How an object can become serializable?
    (a) If a class implements java.io.Serializable class
    (b) If a class or any superclass implements java.io.Serializable interface
    (c) Any object is serializable
    (d) No object is serializable
15. What is serialization?
    (a) Turning object in memory into stream of bytes
    (b) Turning stream of bytes into an object in memory
    (c) Turning object in memory into stream of bits
    (d) Turning stream of bits into an object in memory

Answers

1. (b)	2. (c)	3. (d)	4. (a)	5. (b)	6. (d)	7. (c)	8. (d)	9. (a)	10. (b)
		11. (b)	12. (a)	13. (b)	14. (b)	15. (a)			

Practice Questions

Q.I Answer the following Questions in short:

1. Which package supports Date class?
2. Which classes java.util.regex supports?
3. List the standard streams in java?
4. Write a java statement to allocate buffer.
5. What is purpose of FileInputStream and FileOutputStream?
6. What is FileReader and FileWriter?
7. What is FileChannel?

Q.II Answer the following Questions:

1. Write a note on Observable and Observer Objects.
2. Explain Date class with example.
3. Explain Time class with example.
4. How to extract numbers from a string?

5. List and explain any two methods of Scanner class.
6. List and explain any five symbols used in regular expression.
7. Which are the standard streams in java? Explain.
8. Describe different stream types.
9. What is difference between Character stream and Byte stream?
10. What is Buffer? Explain Buffer types.
11. What are NIO Buffer Characteristics?
12. Write down advantages of FileChannel.
13. How to write data to a FileChannel?
14. Write different steps to serialize an object.
15. Write Java program to print date and current time.
16. Write a Java program to demonstrate the use of Scanner class.
17. Write Java program to check whether file exists or not? If it exists then convert its all contents to uppercase.
18. Write Java program to check the current file permissions.
19. Write Java program to read a file line by line.
20. Write Java program to write content into file using BufferedWriter.

Q.III Write short note on:

1. Stream
2. Character stream
3. Byte stream
4. Observer.
5. Observable.
6. Regular expression.
7. Buffer
8. BufferedInputStream
9. FileChannel
10. FileInputStream
11. FileOutputStream
12. FileReader

■■■

9 UI Programming

Objectives...

- To understand Designing Graphical User Interfaces in Java.
- To study about Components and Containers.
- To learn Layout Managers.
- To study AWT Components.
- To study how to Add a Menu to Window.
- To learn Applet programming.

9.1 INTRODUCTION

- Computer users expect to interact with their computers using a Graphical User Interface (GUI). Java can be used to write GUI programs ranging from simple applets which run on a web page to sophisticated stand-alone applications.
- There are current three sets of Java APIs for graphics programming: AWT (**A**bstract **W**indowing **T**oolkit), Swing and JavaFX.
- AWT API was introduced in JDK 1.0. Most of the AWT components have become obsolete and should be replaced by newer Swing components.
- Swing API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1. JFC consists of Swing, Java2D, Accessibility, Internationalization, and Pluggable Look-and-Feel Support APIs. JFC has been integrated into core Java since JDK 1.2.
- In this chapter we will study about UI programming.

9.2 DESIGNING GRAPHICAL USER INTERFACES IN JAVA

- GUI stands for **Graphical User Interface**, a term used not only in **Java** but in all programming languages that support the development of GUIs.
- A program's graphical user interface presents an easy-to-use visual display to the user. It is made up of graphical components (e.g., buttons, labels, windows) through which the user can interact with the page or application.
- GUI gives programmers an easy-to-use visual experience to build Java applications through which the user can interact.

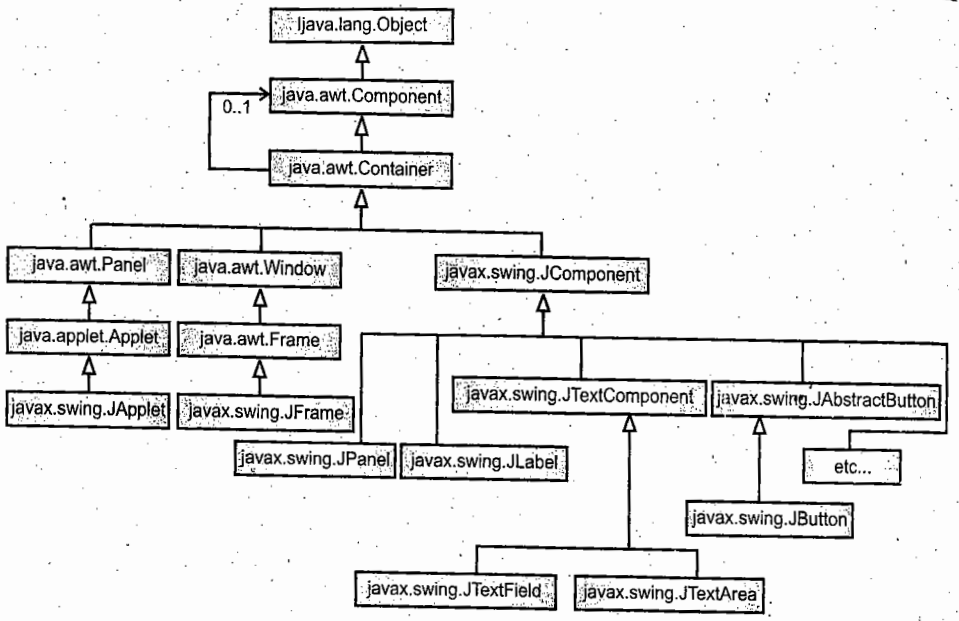


Fig. 9.1: GUI Interface

9.3 COMPONENTS AND CONTAINERS

- Java supports two types of GUI elements:
  1. **Component:** Components are elementary GUI entities, such as Button, Label, and TextField.
  2. **Container:** Containers, such as Frame and Panel, are used to *hold components in a specific layout* (such as FlowLayout or GridLayout). A container can also hold sub-containers.

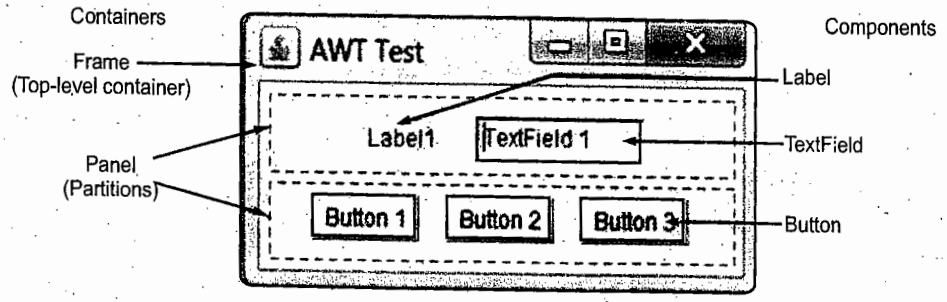


Fig. 9.2: Container and Components

- In the above figure, there are three containers: a Frame, two Panels and five components: a Label, a TextField and three Buttons. The containers and components are the built-in classes supported by Java.

9.3.1 Components

- AWT provides many ready-made and reusable GUI components in *package* **java.awt.** The frequently-used components are: Button, TextField, Label, Checkbox, CheckboxGroup (radio buttons), List, and Choice etc.
- **Three steps are necessary to create and place a GUI component:**
  1. Declare the component with an ***identifier***(*name*);
  2. Construct the component by invoking an appropriate constructor via the **new** operator;
  3. Identify the container (such as Frame or Panel) designed to hold this component. The container can then add this component onto itself via *a* Container **add(*Component*)** method. Every container has a add(Component) method. The container explicitly adds a component onto itself.

9.3.1.1 Basics of Components

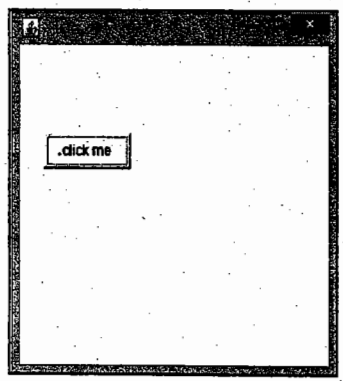
- Basic Java GUI is a collection of classes.
- The Components are the **built-in classes**, if we want to use a GUI component in a program then we have to initialize the class, create an object of the class and then we can use that object of component in our program.
- AWT, Swing is a collection of various built in components such as Button, TextField, Label, Checkbox, CheckboxGroup(radio buttons), List, and Choice etc.
- Every Component has some properties such as size, color, text. All these properties can be used and updated using built in methods such as setSize(), setText(),setColor() etc.
- **For Example:** If we want to use a Button in a program then we need to follow following steps.
  1. Declare the component i.e. **Button btn1;**
  2. Instantiate the component i.e. create an object of Button()
     `btn1=new Button ("DYP");`
  3. Add the component object on container using **add(objectName)**
     `Container.add(btn1);`

Program 9.1: Program to display object on frame.

```
import java.awt.*;
class First extends Frame
{
   First()
   {
      Button b=new Button("click me"); // Creating Button class object
      b.setBounds(30,100,80,30);       // setting button position
      add(b);                          //adding button into frame
      setSize(300,300);                //frame size 300 width and 300
                                       height
```

```
        setLayout(null);                    //no layout manager
        setVisible(true);//now frame will be visible, by default not visible
    }
    public static void main(String args[])
    {
        First f=new First();
    }// End of main
}  //End of Class
```

Output:



- All Components are derived from the abstract class Component which defines methods for setting components' properties. There are two types of components: containers (capable of storing other components) and atomic components (presenting and accepting information), also called *widgets* (abbrev. window gadget).
- A property may be read by using the method: get*XXX*() or is*XXX*() (for binary properties, of the type boolean) and set (where possible) using the method set*XXX*(...) (*XXX* is a name of a property).
- A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user.
- **The most general properties of all components are:**

Table 9.1: Common Properties of Component

Property	Description
`Size`	Setting and getting these properties has limited applications. **Example:** `setSize(width,height)`, `setWidth()`, `setHeight()`, `setLocation()`, `setBounds(x,y,widh,height)`
`Width`	
`Height`	
`Location`	
`Bounds`	

contd. ...

`MinimumSize`	The properties important for some layout managers. They may be set only for Swing components. **Example:** `setMinimunSize()`, `setPreferredSize()`, `setMaximumSize()`.
`PreferredSize`	
`MaximumSize`	
`AlignmentX`	Specifies the alignment of the component. **Example:** `setAlignmentX()`, `setAlignmentY()`
`AlignmentY`	
`Font`	A font is an object of the class Font. **Example:** `setFont(FontClass)`
`Background`	A color is an object of the class Color.
`Foreground`	**Example:** `setBackground(Color.colorName)`, `setForeground(Color.colorName)` Background color (valid for opaque components only)
`Parent`	The container holding this component a read-only property.
`Name`	Components have default names which may be altered.
`Binary properties:`	
`Visible`	Is visible? (modifiable). **Example:** `isVisible();`
`LightWeight`	Is this a lightweight component?
`Enabled`	Is an interaction with this component possible? **Example:** `isEnabled()`

- The methods of Component class are widely used is given below.

Table 9.2: Most Common used Methods of Component Class

Method	Description
`public void add(Component c)`	Add a component on another component.
`public void setSize(int width,int height)`	Sets size of the component.
`public void setLayout(LayoutManager m)`	Sets the layout manager for the component.
`public void setVisible(boolean b)`	Sets the visibility of the component. It is by default false.
`public setBounds(int x,int y,int width,int height)`	Sets the position of the component

9.3.2 Container

- AWT Container is a component and holds all other components.
- Container is a sub-class of the component class and, super-class of all container classes.

9.3.2.1 Using Container

- AWT is a collection of Container and Component classes.
- Layout Manager is assigned to every container, which helps to arrange the components using the setLayout() method.
- An Applet itself is a container. It has a default LayoutManager called FlowLayout.

The Container Class:

- The class **Container** is the super class for all the containers (such as Frame, JFrame,Panel,JPanel) of AWT. Container can hold other AWT components such as Button, TextField etc.
- **Syntax:** Following is the declaration for java.awt.Container class:

```
public class Container extends Component{ }
```

Constructor:

```
Container () // default constructor
```

- **We can add the components on the container by using add() method:**

```
add( ComponentName)
```

Basically Java consists following Containers:

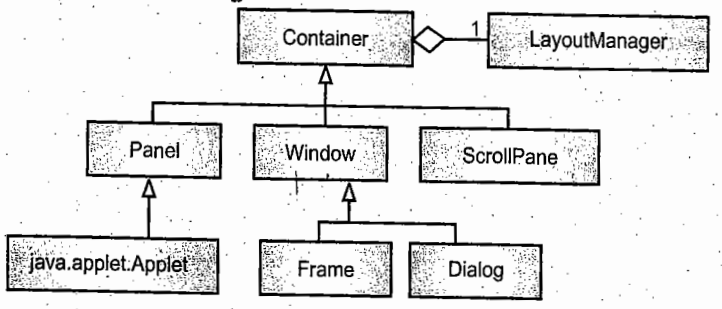


Fig. 9.3: Hierarchy of Container Classes

1. Panel:

- Panel is the smallest container class which is inherited from a Container class. It binds various components together.
- The default LayoutManager is FlowLayout.
- **Syntax**: java.awt.Panel class can be declared as follows:

```
public class Panel extends Container implements Accessible
```

Panel Class Constructor:

1. `Panel()`: Creates a new Panel using the default layout manager.
2. `Panel(LayoutManager)`: Creates a new Panel with the specified layout manager.

Steps to use Panel:

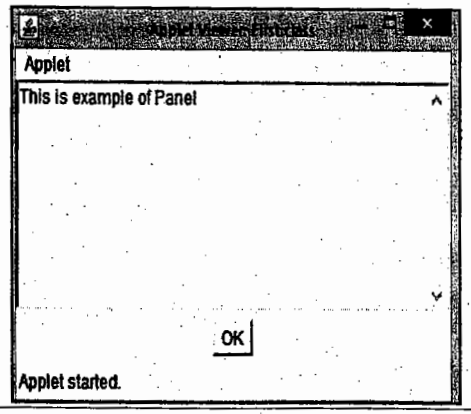
1. First create an object of Panel.
   Example: `Panel P=new Panel();`

2. Set the layout for the Panel.
   Example: `P.setLayout(LayoutManager);`
3. Add component in Panel using add() method.
   Example: `p.add(ComponentName)`
4. Add Panel on Container.
   Example: `add(P) (Here 'P' is a Panel);`

Program 9.2: Program to demonstrate Panel container.

```
import java.applet.*;
import java.awt.*;
/*<applet code="First.class" width="200" height="100"></applet>*/
public class Splesson extends Applet
{
   public void init()
   {
      this.setLayout(new BorderLayout());
      this.add(BorderLayout.CENTER, new TextArea());
      Panel p = new Panel();     //Panel as a container
      p.setLayout(new FlowLayout(FlowLayout.CENTER));
      p.add(new Button("OK"));   // Adding component on Panel
      this.add(BorderLayout.SOUTH, p);
   }
}
```

Output:



2. Window:

- The window class inherits the Container. It does not contain any borders and menu bar
- It is represented as a top-level window.

- As Window class extends from the Container class, other components like TextFields, and Buttons can be added to a window. These components can be arranged by using a LayoutManager. The default LayoutManager of Window class is BorderLayout.
- **Syntax:** java.awt.Window class can be declared as follows:

```
public class Window extends Container implements Accessible{ }
```

Window Class Constructor:

1. `Window(Frame owner)`: Constructs a new, initially invisible window with the specified Frame as its owner.

Window Class Methods: Window class supports many methods, frequently used methods are listed bellow.

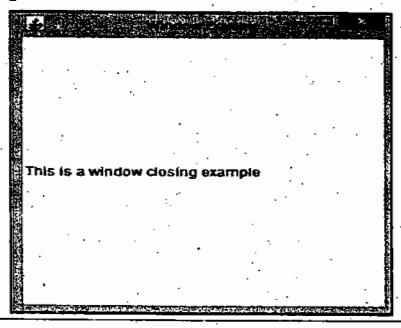
Table 9.3: Methods of Window class

Method	Description
`public void pack()`	Causes subcomponents of this window to be laid out at their preferred size.
`public void show()`	If this window is not yet visible, show makes it visible. If this window is already visible, then this method brings it to the front.
`public void dispose()`	Disposes (close) the current window.
`public void addWindowListener(WindowListener)`	Adds the specified window listener to receive window events from this window.

Program 9.3: Program to demonstrate window closing event.

```
import java.awt.*;
import java.awt.event.*;
class WindowExample extends Frame
{
    Label l1;
    WindowExample()
    {
        this.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {   System.exit(0); }
        });
        l1=new Label("This is a window closing example");
        this.add(l1);
        setTitle("Window Example");
        setSize(300,300);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        new WindowExample();
    }
}
```


Output:



3. Frame:

- Frame is considered as a container with a menu bar, borders and titles in real-time environment.
- A frame can be modified as per the requirement. The default LayoutManager is BorderLayout.
- Frames can divide the data into different windows.
- **Syntax:** java.awt.Frame class can be declared as follows:

```
public class Frame extends Window implements MenuContainer
```

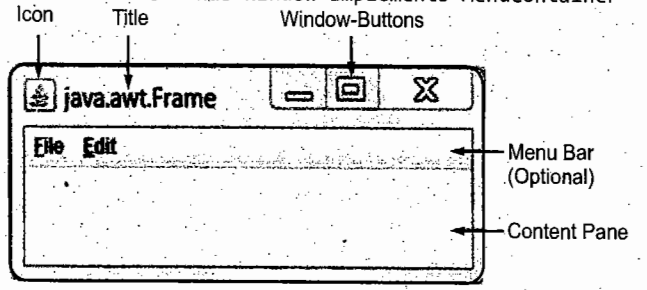


Fig. 9.4: Fame Container

Frame Class Constructor:

1. `Frame()`: Default constructor. To create a frame without a title bar, use Frame() constructor that does not have any arguments.
2. `Frame(String)`: To create a frame and allocate Window title.

Frame Class Methods:

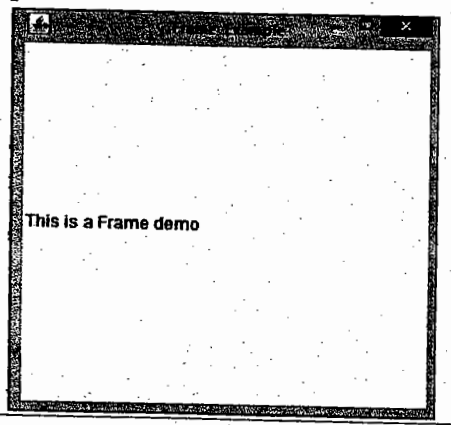
Table 9.4: Methods of Frame class

Method	Description
`public void add()`	Used to add a component in Frame.
`public void setSize(int X,int Y)`	Set the size of the Frame
`Public void setVisible(Boolean)`	Set the visibility property true/false, By default False
`String getTitle()`	Gets the title of the frame.
`String setTitle()`	Sets the title of the frame.
`void setIconImage(Image image)`	Sets the image to be displayed as the icon for this window.

Program 9.4: Program demonstrate Frame.

```
import java.awt.*;
import java.awt.event.*;
class First extends Frame
{
   Label l1;
   First()
   {
      this. addWindowListener(new WindowAdapter()
      {
         public void windowClosing(WindowEvent we)
         {   System.exit(0);
         }
      });
      l1=new Label("This is a Frame demo");
      this.add(l1);
      setTitle("Frame  Example");
      setSize(300,300);
      setVisible(true);
    }
   public static void main(String[] args)
   {
      new First();
   }
}
```

Output:



4. Dialog:

- A Dialog is a *"pop-up window"* used to interact with the users. A Dialog has a title-bar (containing an icon, a title and a close button) and a content display area, as illustrated. It is a sub window which is represented with in a Container (Frame,Window etc).

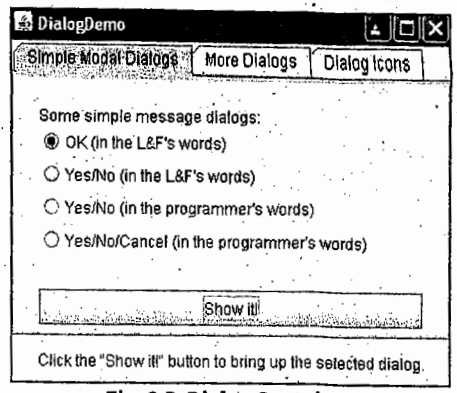


Fig. 9.5: Dialog Container

5. ScrollPane:

- Provides automatic horizontal and/or vertical scrolling for a single child component.

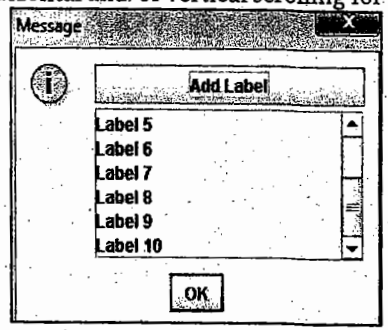


Fig. 9.6: ScrollPane Container

9.4 APPLET

- Applet is java program that can be embedded into HTML pages. Java applets run on the java enables web browsers such as Mozilla, Netscape navigator and internet explorer.
- Applet is dynamic and interactive program that can be executed inside web page by the java supportive browser.

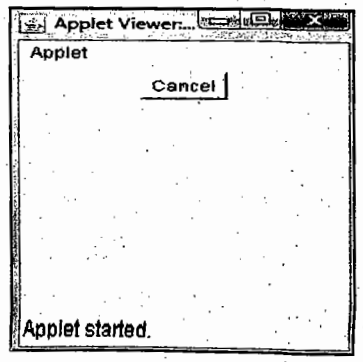


Fig. 9.7: Applet Container

9.4.1 Application Vs Applet

Table 9.5: Difference between Application and Applet

Sr. No.	Applet	Application
1.	Applets are the small programs.	Applications are the large programs.
2.	Applets don't have the main method.	An application execution starts with the main method.
3.	Applets can run in our browser's window or in an appletviewer.	An application is a standalone program that can be invoked from the command line.
4.	Applets are designed for the client site programming purpose.	The applications don't have such type of criteria.
5.	Applets are designed just for handling the client site problems.	While the java applications are designed to work with the client as well as server.
6.	Applet does not support database connectivity.	Application supports database connectivity.

9.4.2 Types of Applet

- We can embed applets into WebPages in two ways.
  - One, we can write our own applets and embed them into webpage's.
  - Second, we can download an applet from a remote computer system and they embed it into a web page.

There are two types of Applet:

1. Local Applet
2. Remote Applet

1. **Local Applet:** An applet developed locally and stored in a local system is known as a local applet. When a webpage is trying to find a local applet, it does not need to use the Internet and therefore the local system does not require any Internet connection.
2. **Remote Applet:** A remote applet is that which is developed by someone else and stored on a remote computer connected to Internet. In order to locate and load remote applet, we must know the applet's address on the web. The address is known as Uniform Resource Locator.

To write applets we need to import the following package:

- **import java.applet.*;** This brings to our program all the applet features.
- **import java.awt.*;** This brings to our program all the shapes, containers and objects that can be used in a graphical user interface.

- **import java.io.*;** This allows the applet to read and display information from and to the screen.

Syntax: `public class FirstApplet extends Applet{ }`

9.4.3 Life Cycle of an Applet

- Every applet inherits a set of default behaviors from the Applet class. As a result, when an applet is loaded, it undergoes a series of changes in its state. The applet states include:
  1. Initialization - invokes init()
  2. Running - invokes start()
  3. Display - invokes paint()
  4. Idle - invokes stop()
  5. Dead/Destroyed state - invokes destroy()

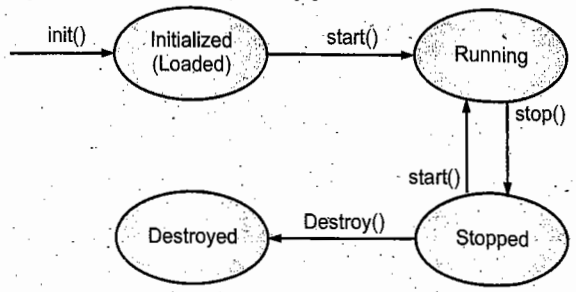


Fig. 9.8: The Lifecycle of an applet

- Following are different methods of an Applet:
  - `public void init()`: Initialization occurs when applet is first loaded. Tasks performed here are creating objects setting initial state, loading images, fonts etc. and setting parameters.
  - `public void init start()`: Starting occurs after initialization and after stop occurs. Tasks include starting threads, sending messages to helper objects, or to tell the applet to start running.
  - `public void init paint()`: This method is to display things how they are to be drawn on screen whether in the form of text, graphics or background. Redisplay occurs many times within the life of an applet.
  - `public void init stop ()`: Stopping occurs when one leaves a page that contains a running applet. Threads normally continue running but can be manually stopped.
  - `public void init destroy()`: Called if current browser session is being terminated. Frees all resources used by the applet.

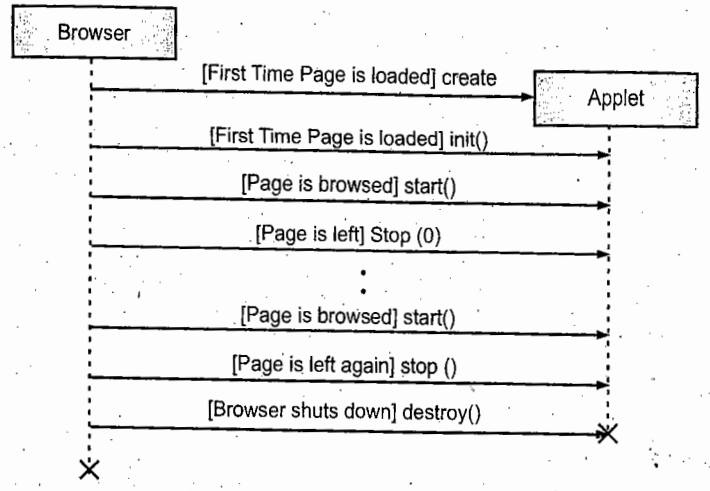


Fig. 9.9: Sequence diagram of Applet execution

Applet Tag:

Table 9.6: Applet Tag

Tag	Description
<applet></applet>	This is the starting tag for applet with parameters
Align	This parameter from applet tag is used to align applet window to an appropriate position.
Code	This parameter consist name of the applet class which is to be displayed.
Width	This parameter specifies width of the applet window.
Height	This parameter specifies height of the applet window.
	For Example: <APPLET ALIGN="CENTER" CODE="FirstApplet.class" WIDTH="800" HEIGHT="500"></APPLET>

9.4.4 Steps to write Applet Program:

1. **Import applet package and awt package.**

```
import java.awt.*;
import java. applet.*;
```

2. **Create applet class with the access specifier public and extend Applet class.**

```
public class FirstApplet extends Applet
```

3. **Use init method if we want some initialization to be done.**

```
public void init()
{
   int x=0,y=45;
   setSize (250,300);
}
```


4. **Write the paint method for displaying message.**

```
public void paint (graphics g)
{
   g.drawString("First Applet Example",x,y);
}
```

5. **Now add html code for executing applet with the help of java compatible browser. There are two steps for writing html code. Create a separate html file with the following code:**

```
file Name: FirstApple.html
<html><body>
<applet align="CENTER" code="FirstApplet.class" width="800"
height="500"></applet>
</body></html>
```

Save the above code by the name of FirstApplet.html. Compile the FirstApplet.java on command prompt.

```
C:\jdk1.5.0\bin>javac FirstApplet.java
C:\jdk1.5.0\bin>appletviewer FirstApplet.html
```

OR

Add the above code in java file before the class name in a comment.

```
/* <applet align="CENTER" code="FirstApplet.class" width="800"
height="500"></applet> */
```

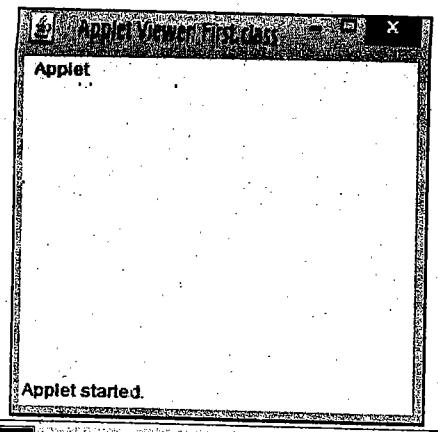
6. **Save the above code by the name of FirstApplet.java. Compile the FirstApplet.java on command prompt.**

```
C:\jdk1.5.0\bin>javac FirstApplet.java
C:\jdk1.5.0\bin>appletviewer FirstApplet.java
```

Program 9.5: Program to demonstate Applet.

```
import java.awt.*;
import java. applet.*;
/*<applet align="CENTER" code="First.class" width="800"
                                                height="500"></applet>*/
public class First extends Applet
{
   int x,y;
   public void init()
   {
      int x=0,y=45;
      setSize (250,300);
   }
   public void paint (Graphics g)
   {
      g.drawString ("First Applet Example",x,y);
   }
}//class closed
```

Output:



9.5 LAYOUT MANAGER

- Layout managers are used to arrange the components on container such as Frmae, Panel etc.
- Every Container, by default, has a layout manager. It implements the LayoutManager interface.
- There are various layout managers are available. We can select any of them according to our programs need.
- The AWT supports following layout managers –
  - Very simple (FlowLayout and GridLayout)
  - Special purpose (BorderLayout and CardLayout)
  - Ultra-flexible (GridBagLayout).

Methods Supporting to Layout manager:

- `setLayout(null)`: This method does not apply any layout to the frame.
- `setLayout(LayoutManager layoutObj)`: This method applies the desired layout manager which is specified.
- `No Layout Manager`: If we are not using any layout within container then it takes by default FlowLayout. The following code give demonstration of program without using layout manager.

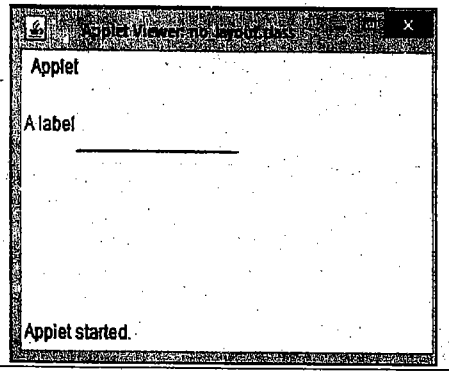
Program 9.6: Program without using layout manager.

```
import java.awt.*;
import java.applet.Applet;
public class no_layout extends Applet
{
   public void init()
   {
      Label a_label;
```



```
      TextField a_textfield;
      setLayout(null);
      a_label = new Label("A label");
      add(a_label);
      //change the size of the Label
      a_label.resize(40,50);
      a_textfield = new TextField(20);
      add(a_textfield);
      //set the size and position of the TextField
      a_textfield.reshape(20,40,140,10);
   }
}
```

Output:



9.5.1 FlowLayout

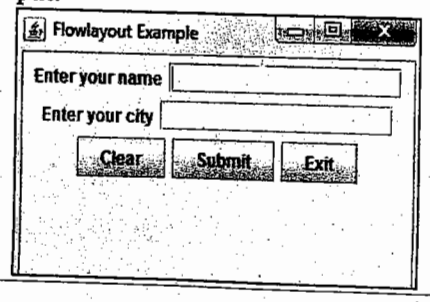
- This is the default Layout Manager that every container uses unless you use the setLayout() method to change it.
- It keeps adding components to the right side of the container. Once it completes adding the components on the first row then it starts with the next row and so on until all the components are added on container.

Program 9.7: Program using FlowLayout demonstration.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class A extends JFrame
{
   JLabel label1, label2, label3;
   JTextField field1, field2;
   JButton button1, button2, button3;
   JFrame jf;
```

```
    A()
    {
        jf = new JFrame("FlowLayout Example");
        jf.setSize(400,150);
        label1 = new JLabel("Enter your name");
        label2 = new JLabel("Enter your city");
        field1 = new JTextField(15); //calling TextField(int widthOfTextField)
        field2 = new JTextField(15);
        button1 = new JButton("Clear");
        button2 = new JButton("Submit");
        button3 = new JButton("Exit");
        //Setting the positioning of the components in container.
        jf.setLayout(new FlowLayout());
        // calling the first constructor of FlowLayout class, which
        // positions each line of components in the center of JFrame
        jf.add(label1);
                    //Adding the first Jlabel component to JFrame container
        jf.add(field1);
                  //Adding the first JTextField component to JFrame container
        jf.add(label2);
                   //Adding the second JLabel component to JFrame container
        jf.add(field2);
                  //Adding the second JTextField component to JFrame container
        jf.add(button1);
                    //Adding the first JButton component to JFrame container
        jf.add(button2);
                   //Adding the second JButton component to JFrame container
        jf.add(button3);
                    //Adding the third JButton component to JFrame container
        jf.setSize(300,200);
        jf.setVisible(true);
    }
    public static void main(String args[])
    {
       new A();// call to constructor
    }
}
```

Output:



9.5.2 GridLayout

- GridLayout places components in a grid format (rows and columns). Each component takes all the available space within its cell, and each cell is exactly the same size.
- If you resize the GridLayout window, you will see that the GridLayout changes the cell size as per the size of the Container(Frame or Panel).

Constructors:

```
public GridLayout(int rows, int columns);
```

- This constructor specifies number of rows and columns a frame consists.

```
public GridLayout(int rows, int columns, int horizontalGap, int verticalGap);
```

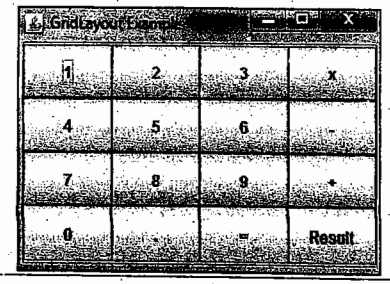
- The horizontalGap and verticalGap arguments to the second constructor allows you to specify the number of pixels between cells. If you don't specify gaps, then it takes by default zero value.

Program 9.8: Program using GridLayout.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class GridDemo extends JFrame
{
   JFrame jf;
   GridDemo()
   {
      jf = new JFrame("GridLayout Example");
      JButton button1 = new JButton("1");
      JButton button2 = new JButton("2");
      JButton button3 = new JButton("3");
      JButton button4 = new JButton("x");

      JButton button5 = new JButton("4");
      JButton button6 = new JButton("5");
      JButton button7 = new JButton("6");
      JButton button8 = new JButton("-");

      JButton button9 = new JButton("7");
      JButton button10 = new JButton("8");
      JButton button11 = new JButton("9");
      JButton button12 = new JButton("+");

      JButton button13 = new JButton("0");
      JButton button14 = new JButton(".");
      JButton button15 = new JButton("=");
      JButton button16 = new JButton("Result");
```

```
        //Setting the layout of the components in container, JFrame, to
        GridLayout
        jf.setLayout(new GridLayout(4,4));
        //GridLayout constructor is called with 4 number of rows  and 4
        number of columns, to hold 16 buttons

        jf.add(button1);
        jf.add(button2);
        jf.add(button3);
        jf.add(button4);

        jf.add(button5);
        jf.add(button6);
        jf.add(button7);
        jf.add(button8);

        jf.add(button9);
        jf.add(button10);
        jf.add(button11);
        jf.add(button12);

        jf.add(button13);
        jf.add(button14);
        jf.add(button15);
        jf.add(button16);

        jf.setSize(300,200);
        jf.setVisible(true);
    }
    public static void main(String args[])
    {
        new GridDemo();//call to constructor
    }
}
```

Output:



9.5.3 BorderLayout

- BorderLayout has five areas: **North, South, East, West, and Center.**
- You can add one component to each of these five areas.
- BorderLayout makes room for the items in the four border areas (referred to as North, South, East, and West), and then whatever is left over is assigned to the Center area.
- This layout is nice if you want to place scrollbars around a panel.

Constructor:

- `public BorderLayout();`
- `public BorderLayout(int horizontalGap, int verticalGap);`

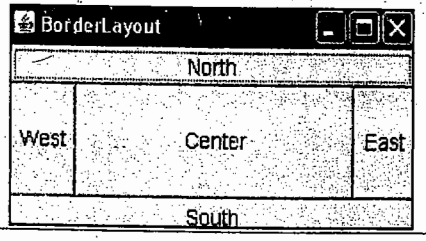
Method:

- To add any component in frame by using borderLayout you can use add() method:
  - **`void add(Componenet compObj,Object region);`** This method is used to add component at specified position. Here, compObj is the component to be added and region specifies where the component will be added.

Program 9. 9: Program using Border Layout Manager.

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET ALIGN="CENTER" CODE="BorderDemo" WIDTH="400"
HEIGHT="200"></APPLET>*/
public class BorderDemo extends Applet
{
    public void init()
    {
        setLayout(new BorderLayout());
        setFont(new Font("Helvetica", Font.PLAIN, 14));

        add("North", new Button("North"));
        add("South", new Button("South"));
        add("East", new Button("East"));
        add("West", new Button("West"));
        add("Center", new Button("Center"));
    }
}
```

Output:



9.5.4 CardLyout

- The CardLayout class is unique among the other layout managers. It stores several different layouts.
- The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructor:

- `CardLayout()`: This creates a default card layout.
- `CardLayout(int horz, int vert)`: This allows you to specify the horizontal and vertical space left between components.

Method:

- Below are all the CardLayout methods that let you choose a component. For each method, the first argument is the container for which the CardLayout is the layout manager.

Table 9.7: CardLayout Methods

Methods	Description
`public void first(Component comp)`	This method flips to the first card in the container.
`public void next(Component comp)`	This method flips to the next card in the container.
`public void last(Component comp)`	This method flips to the last card in the container.
`public void show(Component comp,String name)`	This method shows a specific card in the container with a specific name.
`public void setHgap(int horizontalGap)`	This method sets the horizontal space between components.
`public void setVgap(int verticalGap)`	This method sets the vertical space between components.
`void add(component panelObj,Object name)`	Here, name is a string that specifies the name of the card whose panel is specified by panelObj.

Program 9.10: Program using CardLayout Manager.

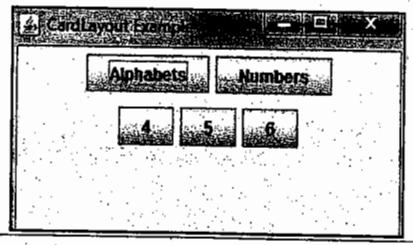
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class CardDemo extends JFrame implements ActionListener
{
    JFrame jf;
    JPanel parentPanel;
    CardLayout cd;
```

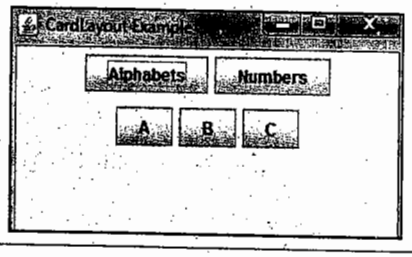


```
CardDemo()
{
    jf = new JFrame("CardLayout Example");
    cd = new CardLayout();

    //Creating a main parent panel that will contain two child panels
    parentPanel = new JPanel();

    //Creating two child panels.
    JPanel childPanel1 = new JPanel();
    JPanel childPanel2 = new JPanel();

    JButton button1 = new JButton("Alphabets");
    JButton button2 = new JButton("Numbers");

    JButton button3 = new JButton("4");
    JButton button4 = new JButton("5");
    JButton button5 = new JButton("6");

    //Adding buttons to childPanel1
    childPanel1.add(button3);
    childPanel1.add(button4);
    childPanel1.add(button5);

    JButton button6 = new JButton("A");
    JButton button7 = new JButton("B");
    JButton button8 = new JButton("C");

    //Adding buttons to childPanel2
    childPanel2.add(button6);
    childPanel2.add(button7);
    childPanel2.add(button8);

    //Setting the positioning of the components in parentPanel,
    JPanel(that contains childPanel1 and childPanel2) to CardLayout
    parentPanel.setLayout(cd);

    //Adding childPanel1 and childPanel2 to parentPanel
    parentPanel.add(childPanel1, "Num");
    parentPanel.add(childPanel2, "Alp");

    button1.addActionListener(this);
    button2.addActionListener(this);

    //Setting container JFrame's layout to FlowLayout.
    jf.setLayout(new FlowLayout());

    //Adding two buttons to JFrame, these buttons will remain commonly
    visible to all panels
```

```
        jf.add(button1);
        jf.add(button2);

        //Adding JPanel, parentPanel to JFrame
        jf.add(parentPanel);

        jf.setSize(300,200);
        jf.setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
    {
        //If "Numbers" button is clicked, open the JPanel with buttons
        showing numbers.
        if(ae.getActionCommand()=="Numbers")
        cd.show(parentPanel,"Num");

        //If "Alphabets" button is clicked, open the JPanel with buttons
        showing alphabets.
        if(ae.getActionCommand()=="Alphabets")
        cd.show(parentPanel,"Alp");
    }
    public static void main(String args[])
    {
        new CardDemo();//call to constructor
    }
}
```

Output:





9.6 AWT COMPONENTS

- AWT stands for **Abstract Windowing Toolkit**. It contains all classes to write the GUI program. You can use the AWT package to develop user interface objects like buttons, checkboxes, radio buttons and menus etc.
- The AWT supports following types of controls:
  1. Labels
  2. Push Buttons
  3. Check Boxes

  4. Choice lists
  5. Lists
  6. Scroll Bars
  7. Text Editing

Some components of Java AWT are explained bellow:

1. Labels:

- This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in your application. Label never performs any type of action.
- **To create a label, use one of the following constructors:**
  - `Label()`: Creates an empty label, with its text aligned left.

    ```
    Label label_name = new Label ();
    ```

  - `Label(String)`: Creates a label with the given text string, also aligned left.

    ```
    Label label_name = new Label ("This is the label text.");
    ```

    Above code simply represents the text for the label.
  - `Label(String, int)`: Creates a label with the given text string and the given alignment.

    ```
    Label label_name = new Label ("This is the label text.", Label.CENTER);
    ```

- The available alignment numbers are stored in class variables in Label, making them easier to remember: Label.RIGHT, Label.LEFT, and Label.CENTER.
- Justification of label can be left, right or centered. Above declaration uses the center justification of the label using the Label.CENTER.

Table 9.8: Methods Supporting to Label

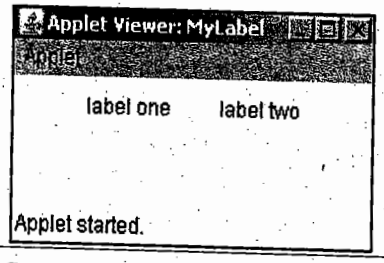
Sr. No.	Method	Description
1.	`getText(String)`	Returns a string containing this label's text.
2.	`setText(String)`	Changes the text of this label.
3.	`getAlignment()`	Returns an integer representing the alignment of this label: 0 is Label.LEFT, 1 is Label.CENTER , 2 is Label.RIGHT
4.	`setAlignment(int)`	Changes the alignment of this label to the given integer.

Program 9.11: Program for Lable component of Java AWT.

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET ALIGN="CENTER" CODE="MyLabel" WIDTH="200"
HEIGHT="200"></APPLET>*/
```

```
public class MyLabel extends Applet
{
   public void init()
   {
   add(new Label("label one"));
   add(new Label("label two", Label.RIGHT));
   }
}
```

Output:



2. Button:

- The Button class is used to create a labeled button. The application result in some action when the button is clicked/pushed.
- **To create a button, use one of the following constructors:**
  - `Button()`: Creates an empty button with no label.
    `Button button_name = new Button ("Ok");`
  - `Button(String)`: Creates a button with the given string as a label.
    `Button button_name = new Button ("Ok");`

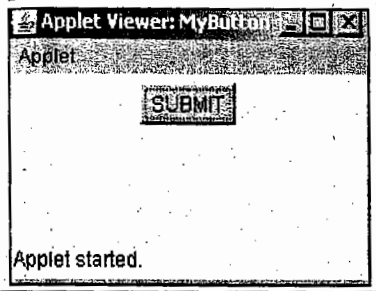
Methods Supporting to Button:

1. `getLabel()`: Returns a string containing this button's text.
2. `setLabel(String)`: Changes the text of this button.
3. `add (button_name)`: Buttons are added to the container using the method.

Program 9.12: Program using Button component.

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET ALIGN="CENTER" CODE="MyButton" WIDTH="400"
HEIGHT="200"></APPLET>*/

public class MyButton extends Applet
{
   public void init()
   {
      Button button = new Button("SUBMIT");
      add(button);
   }
}
```


Output:



3. CheckBoxes:

- This component of Java AWT allows you to create checkboxes in your applications.
- **To create a check Boxes, use one of the following constructors:**
  - `Checkbox()`: Creates an empty(unselected) checkbox.
    `CheckBox checkbox_name = new Checkbox();`
  - `Checkbox(String)`: Creates a checkbox with the given string as a label.
    `CheckBox checkbox_name = new Checkbox ("DYPatil");`
  - `Checkbox(String, null, boolean)`: Creates a checkbox that is either selected or deselected based on whether the boolean argument is true or false, respectively.
    `CheckBox checkbox_name = new Checkbox ("Optional check box 1", false);`
- Above code constructs the unchecked Checkbox by passing the boolean valued argument false with the Checkbox label through the Checkbox() constructor.

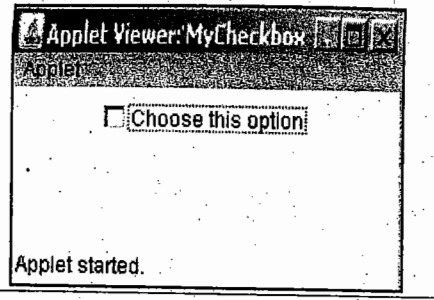
Methods Supporting to CheckBox:

- `getLabel()`: Returns a string containing this Checkbox's text.
- `setLabel (String)`: Changes the text of this Checkbox.
- `add (Checkbox_name)`: Checkbox are added to the container using the method.
- `setState(boolean)`: Sets the state of the checkbox.
- `getState()`: Returns the state of the checkbox.

Program 9.13: Program using Checkbox component.

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET ALIGN="CENTER" CODE="MyCheckbox" WIDTH="400"
HEIGHT="200"></APPLET>*/
public class MyCheckbox extends Applet
{
   public void init()
   {
      Checkbox c = new Checkbox("Choose this option");
      add(c);
   }
}
```

Output:



4. Radio Button:

- This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes with a common group name. Only one Checkbox from a Checkbox Group can be selected at a time.
- **Syntax for creating radio buttons is as follows:**

```
CheckboxGroup chkgp = new CheckboxGroup();
add (new Checkbox ("One", chkgp, false);
add (new Checkbox ("Two", chkgp, false);
add (new Checkbox ("Three",chkgp, false);
```

- In the above code we are making three checkboxes with the label "One", "Two" and "Three". If you mention more than one true valued for checkboxes then your program takes the last true and show the last checkbox as checked.

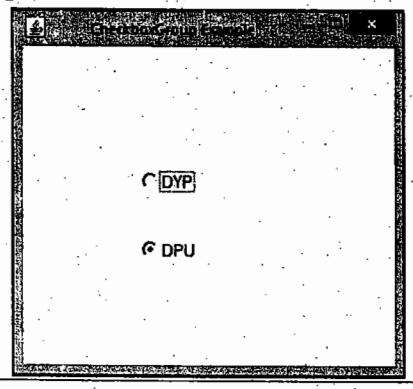
Program 9.14: Program using radio button.

```
import java.awt.*;
public class CheckboxGroupExample
{
        CheckboxGroupExample(){
        Frame f= new Frame("CheckboxGroup Example");
         CheckboxGroup cbg = new CheckboxGroup();
         Checkbox checkBox1 = new Checkbox("DYP", cbg, false);
         checkBox1.setBounds(100,100, 50,50);
         Checkbox checkBox2 = new Checkbox("DPU", cbg, true);
         checkBox2.setBounds(100,150, 50,50);
         f.add(checkBox1);
         f.add(checkBox2);
         f.setSize(400,400);
         f.setLayout(null);
         f.setVisible(true);
    }
```



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

Output:



5. Text Area:

- This is the text container component of Java AWT package. The TextArea contains plain text.
- **To create a TextArea, use the following constructors:**
  - `TextArea txtArea_name = new TextArea()`: Defines a default empty TextArea.
  - `TextArea txtArea=new TextArea(int rows, int columns)`: Defines an empty TextArea with the specified number of rows and columns.
  - `TextArea txtArea=new TextArea(String the_contents)`: Defines a TextArea that contains the specified string.
  - `TextArea txtArea=new TextArea(String the_contents, int rows, int columns)`: Defines a TextArea containing the specified string and with a set number of rows and columns.

Methods Supporting to TextArea:

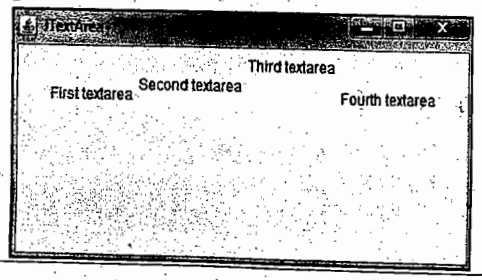
- `setEditable (boolean)`: Makes the Text Area editable. If you pass the Boolean valued argument false then the Text area will be non-editable otherwise it will be editable. By default it is in editable mode.
- `setText(String)`: This method sets the text in the text area.
- `appendText(String)`: Appends the specified string to the current contents of the TextArea.
- `int getColumns()`: Returns the current width of the TextArea in columns.
- `int getRows()`: Returns the current number of rows in a TextArea.

- `insertText(String text, int where_to_add)`: Inserts the specified string at the specified location.
- `replaceText(String new_text, int start, int stop)`: Takes the text between start and stop, inclusive, and replaces it with the specified string.

Program 9.15: Program using TextArea.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class TextDemo extends JFrame
{
    JFrame jf;
    JTextArea textArea1, textArea2, textArea3, textArea4;
    TextDemo()
    {
        jf= new JFrame("JTextArea");
        textArea1 = new JTextArea();     //JTextArea()
        textArea2 = new JTextArea(2,2); //JTextArea(int rows, int columns)
        textArea3 = new JTextArea("Third textarea", 4,4);
        textArea4 = new JTextArea("Fourth textarea");
        textArea1.append("First textarea");
        textArea2.append("Second textarea");
        jf.add(textArea1);
        jf.add(textArea2);
        jf.add(textArea3);
        jf.add(textArea4);

        jf.setLayout(new FlowLayout());
        jf.setSize(400,200);
        jf.setVisible(true);
    }
    public static void main(String args[])
    {
        new TextDemo();//call to constructor
    }
}
```

Output:



6. Text Field:

- TextField class is a text component that allows editing of a single line text. It inherits TextComponent class.
- **To create a text field, use one of the following constructors:**
  - `TextField txtfield = new TextField();`
    Creates an empty TextField that is 0 characters wide (it will be resized by the current layout manager).
  - `TextField thxfield=new TextField(int);`
    Creates an empty text field. The integer argument indicates the minimum number of characters to display.
    `TextField txtfield = new TextField(20);`
  - `TextField txtfield=new TextField(String);`
    Creates a text field initialized with the given string. The field will be automatically resized by the current layout manager.
    `TextField txtfield = new TextField("Hello");`
  - `TextField txtfield=TextField(String, int);`
    Creates a text field containing the given string. If the string is longer than the width, you can select and drag portions of the text within the field, and the box will scroll left or right.
    `TextField txtfield = new TextField("Enter Your Name", 30);`

Methods Supporting to TextField:

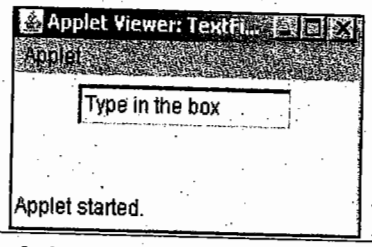
- `getText()`: Returns the text this text field contains (as a string).
- `setText(String)`: Puts the given text string into the field.
- `getColumns()`: Returns the width of this text field.
- `select(int, int)`: Selects the text between the two integer positions (positions start from 0).
- `selectAll()`: Selects all the text in the field.
- `isEditable()`: Returns true or false based on whether the text is editable.
- `setEditable(boolean)`: True (the default) enables text to be edited; false freezes the text.
- `getEchoChar()`: Returns the character used for masking Input.
- `echoCharIsSet()`: Returns true or false based on whether the field has a masking character.

Program 9.16: Program using TextField.

```
import java.awt.*;
import java.applet.Applet;
/*<APPLET ALIGN="CENTER" CODE="TextFieldDemo" WIDTH="400"
HEIGHT="200"></APPLET>*/
```

```
public class TextFieldDemo extends Applet
{
   public void init()
   {
      TextField tf = new TextField("Type in the box");
      add(tf);
   }
}
```

Output:



7. Choice:

- The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.
- Choice menus are pop-up menus of items from which you can choose one item.
- **To create a choice menu:**
  - Create an instance of the Choice class.
  - Use the addItem() method to add individual items to it in the order in which they should appear.
  - Finally, add the entire choice menu to the panel in the usual way.
- **Methods Supporting to Choice Menu:**
  - `getItem(int)`: Returns the string item at the given position (items inside a choice begin at 0, just like arrays).
  - `countItems()`: Returns the number of items in the menu.
  - `getSelectedIndex()`: Returns the index position of the item that's Selected.
  - `getSelectedItem()`: Returns the currently selected item as a string.
  - `select(int)`: Selects the item at the given position.
  - `select(String)`: Selects the item with the given string.
  - `addItem(String name) or add(String name)`: Adds the specified item in list.

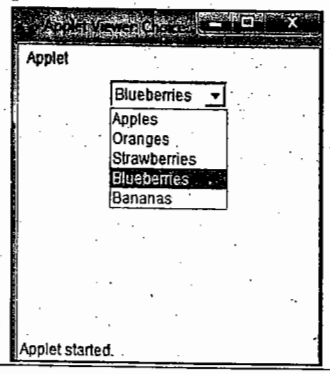
Program 9.17: Program using Choice control.

```
import java.awt.*;
import java.applet.*;
/*<applet height=200 width=200 code="ChoiceTest"></applet>*/
public class ChoiceTest extends Applet
{
```



```
   public void init()
   {
      Choice c = new Choice();
      c.addItem("Apples");
      c.addItem("Oranges");
      c.addItem("Strawberries");
      c.addItem("Blueberries");
      c.addItem("Bananas");
      add(c);
   }
}
```

Output:



8. List:

- The List class provides a compact, multiple choices, scrolling selection list.
- **To create a List, use one of the following constructors:**
  - `List()`: This creates a List control that allows only one item to be selected at one time.
  - `List(int numRows)`: This creates a List that specifies the number of entries in the list that will always be visible.
  - `List(int numRows, boolean multipleselect)`: This creates a List that can be used to select multiple items at a time if multipleselect is true.

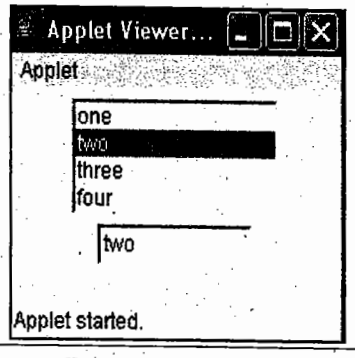
Methods Supporting to List:

- `void add(String name)`: Adds the given item in the list
- `void add(String name, int index)`: Adds the specified item in given index.
- `String getSelectedItem()`: Returns the string containing name of the item.
- `int getSelectedIndex()`: Returns the index of the item.
- `void select(int index)`: Selects the item at the given position
- `int getItemCount()`: Returns the number of items in a list.

Program 9.18: Program using List Component.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet height=200 width=200 code="ListTest"></applet>*/
public class ListTest extends Applet implements ActionListener
{
   List lst;
   TextField tf;
   public void init()
   {
      tf=new TextField(10);
      lst=new List();
      lst.add("one");
      lst.add("two");
      lst.add("three");
      lst.add("four");
      lst.addActionListener(this);
      add(lst);
      add(tf);
   }
   public void actionPerformed(ActionEvent ae)
   {
      String s=lst.getSelectedItem();
      tf.setText(""+s);
   }
}
```

Output:



9. Scrollbar:

- **Scrollbar** class is used to create a horizontal and vertical Scrollbar.
- A Scrollbar can be added to a top-level container like Frame or a component like Panel.

- Scrollbar class is another component in the AWT package which extends the Component class.
- Creates a Scrollbar with the specified orientation, value, extent, minimum, and maximum.

```
public Scrollbar(int orientation, int value int extent, int min, int max)
```

- Creates a Scrollbar with the specified orientation, value, extent, minimum, and maximum, where :
  - **Orientation:** This parameter specifies the Scrollbar to be a horizontal or a vertical Scrollbar.
  - **Value:** This parameter specifies the starting position of the knob of Scrollbar on the track of a Scrollbar.
  - **Extent:** This parameter specified the width of the knob of Scrollbar.
  - **Min:** This parameter specifies the minimum width of the track on which Scrollbar moves.
  - **Max:** This parameter specifies the maximum width of the track.

Methods:

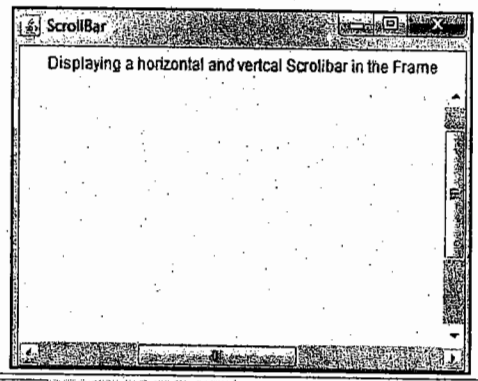
- `public addAdjustmentListener(AdjustmentListener al):` Adds an AdjustmentListener.
- `public int getValue():` Gets the Scrollbar's current position value.
- `public int setValue(int value):` Sets the Scrollbar's current position value.

Program 9.19: Program using ScrollBar.

```
import java.awt.*;
import javax.swing.*;
public class ScrollEx1 extends Frame
{
Button b1, b2, b3, b4, b5;
Frame frame;
Panel panel;
Label label1, label2;

ScrollEx1()
{
frame = new Frame("Scrollbar");
panel = new Panel();
label1 = new Label("Displaying a horizontal and vertical Scrollbar in the
Frame",Label.CENTER);

Scrollbar scrollB1 = new Scrollbar(Scrollbar.HORIZONTAL, 10, 40, 0, 100);
Scrollbar scrollB2 = new Scrollbar(Scrollbar.VERTICAL, 10, 60, 0, 100);

frame.add(label1,BorderLayout.NORTH);
frame.add(scrollB2,BorderLayout.EAST);
frame.add(scrollB1,BorderLayout.SOUTH);

frame.setSize(370,270);
frame.setVisible(true);
}
```

```
public static void main(String args[])
{
    new ScrollEx1();
}
}
```

Output:



9.7 ADDING A MENU TO WINDOW

- Java gives a built in component which can be used to add menu and menu bar in the frame/container using JMenuBar, JMenu and JMenuItem class.
- The JMenuBar class is used to display menu bar on the window or frame. It may have several menus.
- The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.
- The object of JMenuItem class adds a simple labeled menu item. The items used in a menu must belong to the JMenuItem or any of its subclass.
- **JMenuBar class declaration:**

  `public class JMenuBar extends JComponent implements MenuElement, Accessible`
- **JMenu class declaration:**

  `public class JMenu extends JMenuItem implements MenuElement, Accessible`
- **JMenuItem class declaration:**

  `public class JMenuItem extends AbstractButton implements Accessible, MenuElement`

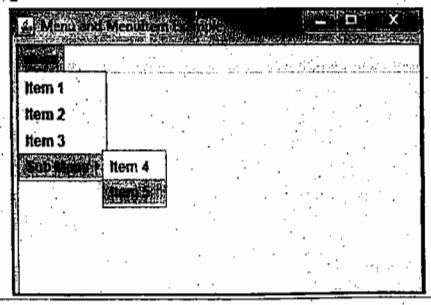
Steps For using Menubar in program:

1. First create a MenuItem using JMenuItem class.
2. Create a Menu using JMenu class.
3. Add MenuItem in Menu class.
4. JMenuItem class adding all the MenuItem in all Menu, finally add the Menu in JMenuBar.
5. Finally add JMenuBar on the Frame.

Program 9.20: Java Program for JMenuItem and JMenu.

```
import javax.swing.*;
class MenuExample
{
    JMenu menu, submenu; //declaring menu and submenu
    JMenuItem i1, i2, i3, i4, i5; //declaring menu item
    MenuExample()
    {
        JFrame f= new JFrame("Menu and MenuItem Example");
        JMenuBar mb=new JMenuBar(); //create MenuBar
        menu=new JMenu("Menu");
        submenu=new JMenu("Sub Menu");
        i1=new JMenuItem("Item 1");
        i2=new JMenuItem("Item 2");
        i3=new JMenuItem("Item 3");
        i4=new JMenuItem("Item 4");
        i5=new JMenuItem("Item 5");
        menu.add(i1); menu.add(i2); menu.add(i3); //add menuitem in menu
        submenu.add(i4); submenu.add(i5); //add menuitem in submenu
        menu.add(submenu); //add submenu in Menu
        mb.add(menu); //add Menu in MenuBar
        f.setJMenuBar(mb); //add MenuBar on frame using setJMenuBar()
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        new MenuExample();
    }
}
```

Output:



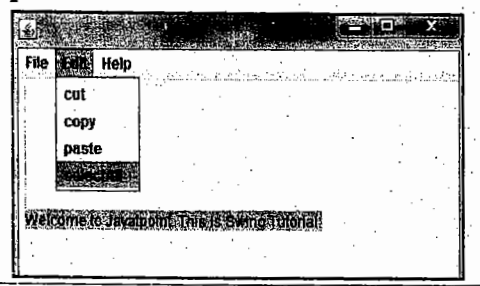
Program 9.21: Program for creating Edit menu for Notepad.

```
import javax.swing.*;
import java.awt.event.*;
public class MenuExample implements ActionListener
{
  JFrame f;
  JMenuBar mb;
  JMenu file,edit,help;
  JMenuItem cut,copy,paste,selectAll;
  JTextArea ta;
  MenuExample()
  {
     f=new JFrame();
     cut=new JMenuItem("cut");
     copy=new JMenuItem("copy");
     paste=new JMenuItem("paste");
     selectAll=new JMenuItem("selectAll");
     cut.addActionListener(this);
     copy.addActionListener(this);
     paste.addActionListener(this);
     selectAll.addActionListener(this);
     mb=new JMenuBar();
     file=new JMenu("File");
     edit=new JMenu("Edit");
     help=new JMenu("Help");
     edit.add(cut);edit.add(copy);edit.add(paste);edit.add(selectAll);
     mb.add(file);mb.add(edit);mb.add(help);
     ta=new JTextArea();
     ta.setBounds(5,5,360,320);
     f.add(mb);f.add(ta);
     f.setJMenuBar(mb);
     f.setLayout(null);
     f.setSize(400,400);
     f.setVisible(true);
  }
  public void actionPerformed(ActionEvent e)
  {
     if(e.getSource()==cut)
     ta.cut();
     if(e.getSource()==paste)
     ta.paste();
     if(e.getSource()==copy)
```



```
     ta.copy();
     if(e.getSource()==selectAll)
     ta.selectAll();
  }
  public static void main(String[] args) {
  new MenuExample();
  }
}
```

Output:



9.7.1 Using Java JPopupMenu

- PopupMenu can be dynamically popped up at specific position within a component. It inherits the JComponent class.
- **Syntax:** Let's see the declaration for javax.swing.JPopupMenu class.

```
public class JPopupMenu extends JComponent implements Accessible,
MenuElement
```

Constructors:

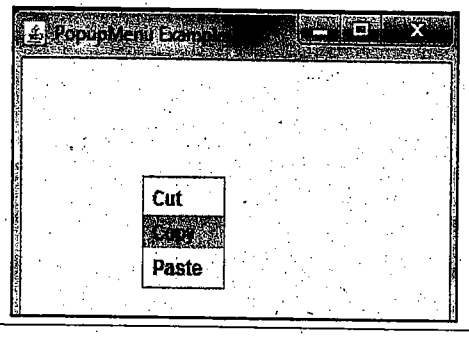
- `JPopupMenu()`: Constructs a JPopupMenu without an "invoker".
- `JPopupMenu(String label)`: Constructs a JPopupMenu with the specified title.

Program 9.22: Program for Java JPopupMenu.

```
import javax.swing.*;
import java.awt.event.*;
class PopupMenuExample
{
   PopupMenuExample()
   {
      final JFrame f= new JFrame("PopupMenu Example");
      final JPopupMenu popupmenu = new JPopupMenu("Edit");
      JMenuItem cut = new JMenuItem("Cut");
      JMenuItem copy = new JMenuItem("Copy");
      JMenuItem paste = new JMenuItem("Paste");
      popupmenu.add(cut); popupmenu.add(copy); popupmenu.add(paste);
```

```
        f.addMouseListener(new MouseAdapter()
        {
           public void mouseClicked(MouseEvent e)
           {
              popupmenu.show(f, e.getX(), e.getY());
           }
        });
        f.add(popupmenu);
        f.setSize(300,300);
    f.setLayout(null);
    f.setVisible(true);
    }
    public static void main(String args[])
    {
        new PopupMenuExample();
    }
}
```

Output:



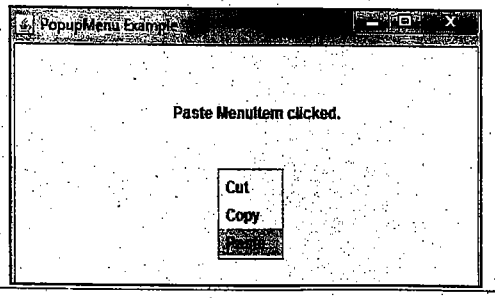
Program 9.23: Program for JPopupMenu with MouseListener and ActionListener.

```
import javax.swing.*;
import java.awt.event.*;
class PopupMenuExample
{
    PopupMenuExample()
    {
        final JFrame f= new JFrame("PopupMenu Example");
        final JLabel label = new JLabel();
        label.setHorizontalAlignment(JLabel.CENTER);
        label.setSize(400,100);
        final JPopupMenu popupmenu = new JPopupMenu("Edit");
        JMenuItem cut = new JMenuItem("Cut");
        JMenuItem copy = new JMenuItem("Copy");
        JMenuItem paste = new JMenuItem("Paste");
        popupmenu.add(cut); popupmenu.add(copy); popupmenu.add(paste);
```



```
        f.addMouseListener(new MouseAdapter()
        {
           public void mouseClicked(MouseEvent e)
           {
              popupmenu.show(f, e.getX(), e.getY());
           }
        });
        cut.addActionListener(new ActionListener()
        {
           public void actionPerformed(ActionEvent e)
           {
              label.setText("cut MenuItem clicked.");
           }
        });
        copy.addActionListener(new ActionListener()
        {
           public void actionPerformed(ActionEvent e)
           {
              label.setText("copy MenuItem clicked.");
           }
        });
        paste.addActionListener(new ActionListener()
        {
           public void actionPerformed(ActionEvent e)
           {
              label.setText("paste MenuItem clicked.");
           }
        });
        f.add(label); f.add(popupmenu);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        new PopupMenuExample();
    }
}
```

Output:



9.8 EXTENDING GUI FEATURES USING SWING COMPONENTS

- Swing is a set of classes that provides more powerful and flexible components than AWT.
- Swing is important to develop Java programs with a graphical user interface (GUI). There are many components which are used for the building of GUI in Swing.
- These components are also helpful in developing interactive Java applications.
- Following are some of the components which are included in Swing toolkit:
  - list controls
  - buttons
  - labels
  - tree controls
  - table controls
- All AWT flexible components can be handled by the Java Swing. Swing toolkit contains more components than the simple component toolkit.
- The Java Foundation Classes (JFC) which supports many more features inportant to a GUI program comprises of Swing.
- The features which are supported by Java Foundation Classes (JFC) are the ability to create a program that can work in different languages, the ability to add rich graphics functionality etc.

Java Swing class Hierarchy:

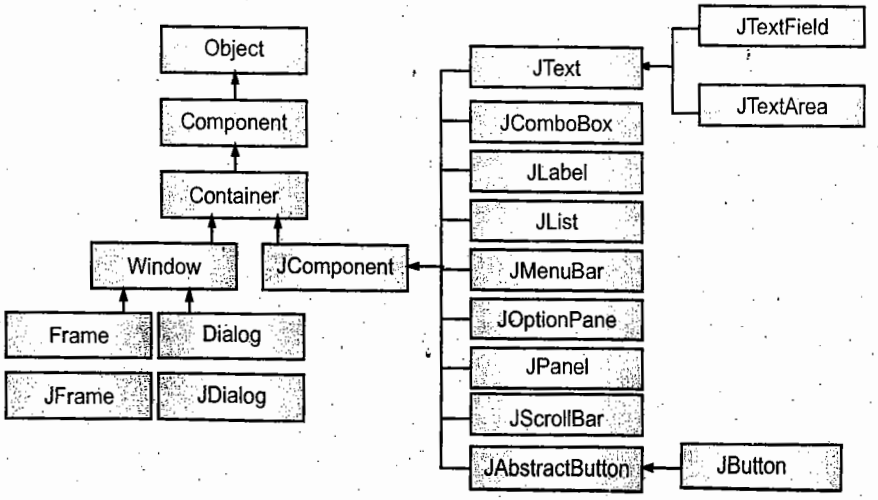


Fig. 9.10: Java Swing Class Hierarchy

9.8.1 Swing GUI Components

- There are several components contained in Swing toolkit such as checkboxes, buttons, tables, text etc.
- Some very simple components also provide sophisticated functionality.
- For creating java standalone application you must provide GUI for a user.

9.8.1.1 Swing Container

- Containers are the on screen windows that contains controls or sub container.

1. JFrame:

- The components added to the frame are referred to as its contents; these are managed by the contentPane.
- To add a component to a JFrame, we must use its contentPane.

JFrame Constructor:

- `JFrame()`: Constructs a new frame that is initially invisible.
- `JFrame(String title)`: Creates a new, initially invisible Frame with the specified title.

JFrame Methods:

- `setSize(width,height)`: This method sets the size of JFrame.
- `setVisible(Boolean)`: This method set the visibility of JFrame. If set true displays the Frame else hides the frame.
- `setTitle(String str)`: This method sets the title for the frame.
- `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`: This method sets the close operation to Exit the application using the System exit method.
- `setResizable(Boolean)`: The method has been used to make the frame resizable or not. If you pass the boolean value false then the frame will be non-resizable otherwise frame will be resizable.

Steps for Crating a JFrame Window:

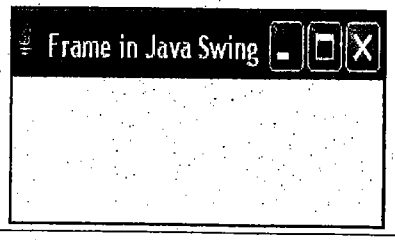
Step 1 : Construct an object of the JFrame class.
Step 2 : Set the size of the JFrame.
Step 3 : Set the title of the JFrame to appear in the title bar
Step 4 : Set the default close operation. When the user clicks the close button, the program stops running.
Step 5 : Make the JFrame visible.

Program 9.24: Write Java program using JFrame.

```
import javax.swing.*;
public class SwingFrame
{
    public static void main(String[] args)
    {
```

```
        JFrame frame = new JFrame("Frame in Java Swing");
        frame.setSize(200, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}.
```

Output:



2. JPanel:

- The Swing JPanel class is similar to the AWT's Panel class.
- JPanel supports all of the AWT layout managers but also provides some new layouts

Constructor:

- `JPanel jp1=new JPanel(); // uses a default constructor`

Method:

- `JPanel.add(Component)`: This method is used to add component to JPanel.

3. JApplet:

- JApplet is the Swing equivalent of the AWT Applet class.

Constructor:

- `JApplet()`: This constructor creates a new JApplet instance.

Methods:

- `public void setContentPane(Container contentPane)`: Sets the contentPane property. This method is called by the constructor.
- `public Container getContentPane()`: Returns the contentPane object for this frame.

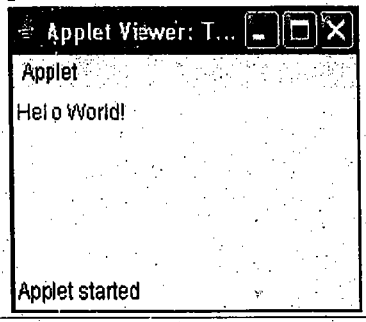
Program 9.25: Java program using JPanel and JApplet.

```
import java.awt.*;
import javax.swing.*;
/*<applet code="TestApplet" height=200 width=200></applet> */
public class TestApplet extends JApplet
{
   public void init()
   {
      setSize( 200, 100 );
      setBackground( Color.gray );
```



```
      JPanel topPanel = new JPanel();
      topPanel.setLayout( new BorderLayout() );
      getContentPane().add( topPanel );

      Label labelHello = new Label( "Hello World!" );
      topPanel.add( labelHello, BorderLayout.NORTH );
   }
}
```

Output:



9.8.1.2 Swing Components

1. JButton:

- JButton are very similar to Button in AWT.
- You can create a JButton with a String as a label, and then add it in a window.
- Events are normally handled just as with a Button in AWT.
- JButton can be registered with ActionListener using the addActionListener() method.

JButton Constructor:

- `JButton(String, Icon)`
- `JButton(String)`
- `JButton(Icon)`
- `JButton()`

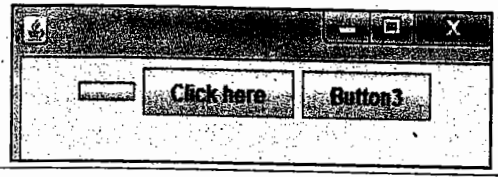
JButton Methods:

- **`void setText(String)` and `String getText()`:** Set or get the text displayed by the button.
- **`void setIcon(Icon)` and `Icon getIcon()`:** Set or get the image displayed by the button when the button isn't selected or pressed.
- **`void setSelected(boolean)` and `boolean isSelected()`:** Set or get whether the button is selected. Makes sense only for buttons that have on/off state, such as checkboxes.
- **`void setActionCommand(String)` and `String getActionCommand(void)`:** Set or get the name of the action performed by the button.
- **`void addActionListener(ActionListener)` and `ActionListener removeActionListener()`:** Add or remove an object that listens for action events fired by the button.

Program 9.26: Java program for JButton.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class ButtonDemo
{
   JFrame jf;
   JButton button1, button2, button3;
   ButtonDemo()
   {
      jf= new JFrame();
      button1= new JButton();
      button2= new JButton("Click here");
      button3= new JButton();
      button3.setText("Button3");
      jf.add(button1);
      jf.add(button2);
      jf.add(button3);
      jf.setLayout(new FlowLayout());
      jf.setSize(300,100);
      jf.setVisible(true);
   }
  public static void main(String args[])
  {
     new ButtonDemo();
  }
}
```

Output:



2. JTextField:

- JTextField is a class which allows you to create and edit one line text.

JTextField Constructor:

- JTextField()
- JTextField(int cols)
- JTextField(String s,int cols)
- JTextField(String s)

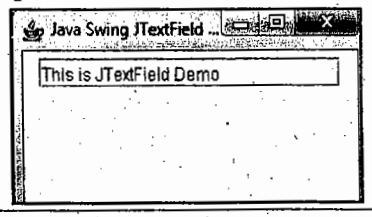
JTextField Methods:

- void setText(String) and String getText(): Set or get the text displayed by the JTextField.

Program 9.27: Program usingf JTextFiled.

```
import java.awt.*;
import javax.swing.*;
public class JTextFieldDemo extends JFrame
{
   public  JTextFieldDemo()
   {
      Container content = getContentPane();
      content.setBackground(Color.white);
      content.setLayout(new FlowLayout());
      JTextField text1= new JTextField(20);
      content.add(text1);
      setTitle("Java Swing JTextField Demo");
      setVisible(true);
      setSize(200,200);
   }
   public static void main(String[] args)
   {
      new JTextFieldDemo();
   }
}
```

Output:



3. JCheckBoxes:

- The JCheckBox class is a concrete implementation of AbstractButton. It is used to create checkbox object and use it in the Frame.

JCheckBox Constructor:

- JCheckBox(Icon i)
- JCheckBox(Icon i,boolean state)
- JCheckBox(String s)
- JCheckBox(String s,boolean state)
- JCheckBox(String s,Icon i)

- `JCheckBox(String s,Icon I,boolean state)`
- `JCheckBox(String s,Icon i)`
- `JCheckBox(String s,Icon I,boolean state)`

  Here,
  - **String:** Specifies the text that the checkbox should display.
  - **Icon:** Specifies the image that the checkbox should display. Unless you specify an image, the images defined by the look-and-feel are used.
  - **Boolean:** Specifies whether the checkbox is selected. By default, it's false (not selected).

JCheckBox Methods:

- `String getText()`: This method gets the text for the check box and uses it to set the text inside the text field
- `void setSelected(Boolean state)`: The state of the checkbox can be changed
- `void itemStateChanged(ItemEvent ie)`: This method handles the checkbox event.
- getItem(): This method gets the JCheckBox object that generated the event

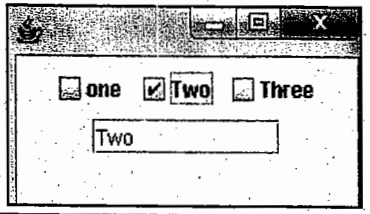
Program 9.28: Java program for JCheckBox.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class JCheckDemo extends JFrame implements ItemListener
{
   JTextField jf;
   public JCheckDemo()
   {  Container cont=getContentPane();
      cont.setLayout(new FlowLayout());

      JCheckBox cb=new JCheckBox("one");
      cb.addItemListener(this);
      cont.add(cb);

      JCheckBox cb1=new JCheckBox("Two");
      cb1.addItemListener(this);
      cont.add(cb1);

      JCheckBox cb2=new JCheckBox("Three");
      cb2.addItemListener(this);
      cont.add(cb2);

      jf=new JTextField(10);
      cont.add(jf);

      setSize(200,200);
      setVisible(true);
   }
```



```
   public void itemStateChanged(ItemEvent ie)
   {  JCheckBox cb=(JCheckBox)ie.getItem();
      jf.setText(cb.getText());
   }
   public static void main(String args[])
   {
      new JCheckDemo();
   }
}
```

Output:



4. JRadioButtons:

- Radio buttons are supported by the JRadioButton class,which is a concrete implementation of Abstract Button. Radio buttons must be configured into a group.
- Only one of the buttons can be selected at a time.

JRadioButtons Constructor:

- `JRadioButton(Icon i)`: Creates an initially unselected radio button with no set text.
- `JRadioButton(Icon i,boolean state)`: Creates a radio button with the specified image and selection state, but no text.
- `JRadioButton(String s)`: Creates an unselected radio button with the specified text.
- `JRadioButton(String s,boolean state)`: Creates a radio button with the specified text and selection state.
- `JRadioButton(String s,Icon i)`: Creates a radio button that has the specified text and image, and that is initially unselected.
- `JRadioButton(String s,Icon I,boolean state)`: Creates a radio button that has the specified text, image, and selection state.

  where

  String: Specifies the text that the radio button should display.

  Icon: Specifies the image that the radio button should display. Unless you specify an image, the images defined by the look-and-feel are used.

  Boolean: Specifies whether the radio button is selected. By default, it's false (not selected).

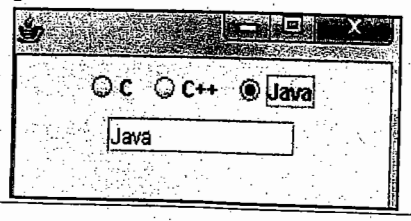
JRadioButtons Methods:

- `String isSelected()`: This method tell whether the radio button is selected or not.
- `void setSelected(boolean state)`: The state of the radio buttons can be changed
- `void add(AbstractButton ab)`: Elements are added to the button group.

Program 9.29: Program using of JRdioButton.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class JRadioDemo extends JFrame implements ActionListener
{  JTextField jf;
   public JRadioDemo()
   {
      Container cont=getContentPane();
      cont.setLayout(new FlowLayout());
      JRadioButton b1=new JRadioButton("C");
      b1.addActionListener(this);
      cont.add(b1);

      JRadioButton b2=new JRadioButton("C++");
      b2.addActionListener(this);
      cont.add(b2);

      JRadioButton b3=new JRadioButton("Java");
      b3.addActionListener(this);
      cont.add(b3);

      jf=new JTextField(10);
      cont.add(jf);

      setSize(200,200);
      setVisible(true);
   }
   public void actionPerformed(ActionEvent ae)
   {
      jf.setText(ae.getActionCommand());
   }
   public static void main(String args[])
   {
      new JRadioDemo();
   }
}
```

Output:



5. JCombo Boxes:

- Combo box are supported by JComboBox class which extends JComponent.
- A combobox normally displays one entry. It displays a drop down list that allows a user to select a different entry.

JComboBox Constructor:

- **`JComboBox()`:** Creates a JComboBox with a default data model.
- **`JComboBox(ComboBoxModel aModel)`:** Creates a JComboBox that takes it's items from an existing ComboBoxModel.
- **`JComboBox(Object[] items)`:** Creates a JComboBox that contains the elements in the specified array.
- **`JComboBox(Vector items)`:** Creates a JComboBox that contains the elements in the specified Vector.

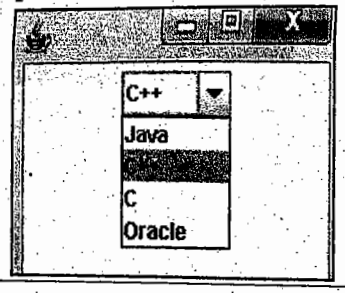
JComboBox Methods:

- **`void addItem(Object obj)`:** Using this method items are added to the list.
- **`void itemStateChanged(ItemEvent ie)`:** This method handles the combo box event.

Program 9.30: Program using JComboBox.

```
import java.awt.*;
import javax.swing.*;
import jaba.awt.event.*;
public class JCombodemo exteds JFrame
{  JComboBox jc;
   public JCombodemo()
   {  Container cont=getContentPane();
      cont.setLayout(new FlowLayout());
      jc=new ComboBox();
      jc.addItem("Java");
      jc.addItem("C++");
      jc.addItem("C");
      jc.addItem("Oracle");
      cont.add(jc);
      setSize(200,200);
      setVisible(true);
      JFrame.setDefaultCloseOperation("EXIT_ON_CLOSE");
   }
   public static void main(String args[])
   {new JCombobemo();}
}
```

Output:



6. JTabbed Panes:

- A tabbed pane is a component that appears as a group of folders in a file.
- Each folder has a title. Each folder consist its content. Only one of the folder may be selected as a time.
- To create a tabbed pane, you simply instantiate JTabbedPane, create the components you wish it to display, and then add the components to the tabbed pane using the [PENDING] method.

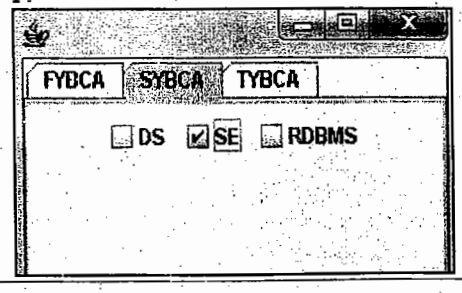
Program 9.31: Java program for JTabbedPane.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class JTabbedDemo extends JFrame
{
   public JTabbedDemo()
   {  JTabbedPane jtf=new JTabbedPane(); //declare and initialize
      jtf.addTab("FYBCA",new FyPanel());  //adding new Tab
      jtf.addTab("SYBCA",new SyPanel());
      jtf.addTab("TYBCA",new TyPanel());
      getContentPane().add(jtf); //add JTabbedPane on container
      setVisible(true);
      setSize(200,200);
   }
   public static void main(String args[])
   {
      new JTabbedDemo();
   }
}
//FyPanel class
class FyPanel extends JPanel
{
   public FyPanel()
   {  JButton b1=new JButton("PPA");
```



```
      add(b1);
      JButton b2=new JButton("C Language");
      add(b2);
      JButton b3=new JButton("DBMS");
      add(b3);
   }
}
//SyPanel class
class SyPanel extends JPanel
{
   public SyPanel()
   {
      JCheckBox cb1=new JCheckBox("DS");
      add(cb1);
      JCheckBox cb2=new JCheckBox("SE");
      add(cb2);
      JCheckBox cb3=new JCheckBox("RDBMS");
      add(cb3);
   }
}
//TyPanel class
class TyPanel extends JPanel
{
   public TyPanel()
   {
      JComboBox jcb=new JComboBox ();
      jcb.addItem("CoreJava");
      jcb.addItem(".Net");
      jcb.addItem("Networking");
      add(jcb);
   }
}
```

Output:



7. Working with DialogBox:

- User interaction can also be possible by using Message Dialog control such as Yes/No, Input Message Dialog box etc.
- This can be achieved using JOptionPane class.

JOptionPane class:

- The JOptionPane class is used to provide standard dialog boxes such as message dialog box, confirm dialog box and input dialog box.
- These dialog boxes are used to display information or get input from the user.
- The JOptionPane class inherits JComponent class.
- **Syntax:** `public class JOptionPane extends JComponent implements Accessible`

JOptionPane Constructor:

- `JOptionPane()`: It is used to create a JOptionPane with a test message.
- `JOptionPane(Object message)`: It is used to create an instance of JOptionPane to display a message.
- `JOptionPane(Object message, int messageType)`: It is used to create an instance of JOptionPane to display a message with specified message type and default options.

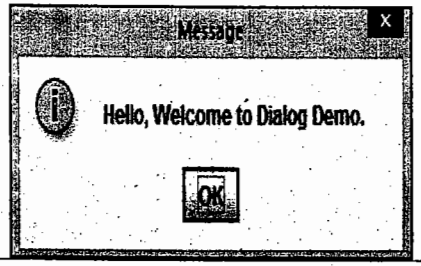
Table 9. 9: Methods of JOptionPane class

Methods	Description
`JDialog createDialog(String title)`	It is used to create and return a new parentless JDialog with the specified title.
`static void showMessageDialog(Component parentComponent, Object message)`	It is used to create an information-message dialog titled "Message".
`static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)`	It is used to create a message dialog with given title and messageType.
`static int showConfirmDialog(Component parentComponent, Object message)`	It is used to create a dialog with the options Yes, No and Cancel; with the title, Select an Option.
`static String showInputDialog(Component parentComponent, Object message)`	It is used to show a question-message dialog requesting input from the user parented to parentComponent.
`void setInputValue(Object newValue)`	It is used to set the input value that was selected or input by the user.

Program 9.32: Java program for JOptionPane to use showMessageDialog().

```
import javax.swing.*;
public class OptionPaneExample
{
    JFrame f;
    OptionPaneExample()
    {
        f=new JFrame();
        JOptionPane.showMessageDialog(f,"Hello, Welcome to Dialog Demo.");
    }
    public static void main(String[] args)
    {
        new OptionPaneExample();
    }
}
```

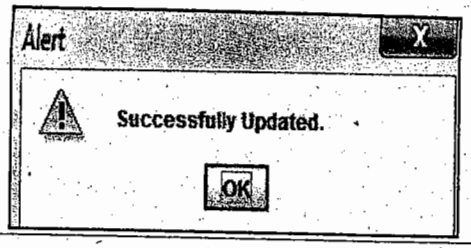
Output:



Program 9.33: Java program for JOptionPaneshowMessageDialog() with information.

```
import javax.swing.*;
public class OptionPaneExample
{
    JFrame f;
    OptionPaneExample()
    {
        f=new JFrame();
        JOptionPane.showMessageDialog(f,"Successfully
                                Updated.","Alert",JOptionPane.WARNING_MESSAGE);
    }
    public static void main(String[] args)
    {
        new OptionPaneExample();
    }
}
```

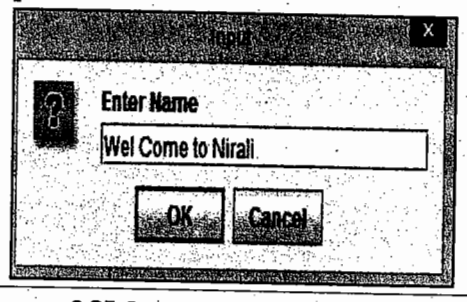
Output:



Program 9.34: Program using JOptionPane showInputDialog().

```
import javax.swing.*;
public class OptionPaneExample
{
    JFrame f;
    OptionPaneExample()
    {
        f=new JFrame();
        String name=JOptionPane.showInputDialog(f,"Enter Name");
    }
    public static void main(String[] args)
    {
        new OptionPaneExample();
    }
}
```

Output:



Program 9.35: Java program using JOptionPane showConfirmDialog()

```
import javax.swing.*;
import java.awt.event.*;
public class OptionPaneExample extends WindowAdapter
{
    JFrame f;
    OptionPaneExample()
    {
        f=new JFrame();
```



```
        f.addWindowListener(this);
        f.setSize(300, 300);
        f.setLayout(null);
        f.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        f.setVisible(true);
    }
    public void windowClosing(WindowEvent e)
    {
        int a=JOptionPane.showConfirmDialog(f,"Are you sure?");
        if(a==JOptionPane.YES_OPTION)
        {
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
    }
    public static void main(String[] args)
    {
        new OptionPaneExample();
    }
}
```

Output:

8. Working with Java JTable:

- The JTable class is used to display data in tabular form. It is composed of rows and columns.
- It is available in javax.swing.JTable class.

JTable Constructors:

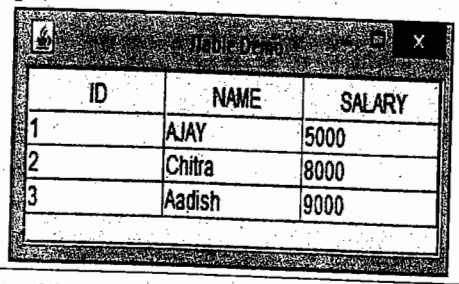
- **JTable():** Creates a table with empty cells.
- **JTable(Object[][] rows, Object[] columns):** Creates a table with the specified data.

Program 9.36: Java program for JTable.

```
import javax.swing.*;
public class TableExample
{
    JFrame f;
    TableExample()
    {
```

```
        f=new JFrame("JTable Demo");
        String data[][]={ {"1","AJAY","5000"},
                          {"2","Chitra","8000"},
                          {"3","Aadish","9000"}};
        String column[]={"ID","NAME","SALARY"};
        JTable jt=new JTable(data,column);
        jt.setBounds(30,40,200,300);
        JScrollPane sp=new JScrollPane(jt);
        f.add(sp);
        f.setSize(300,400);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TableExample();
    }
}
```

Output:



9. Working with JFileChooser:

- The object of JFileChooser class represents a dialog window from which the user can select file. It inherits JComponet class.
- It is in javax.swing.JFileChooser class.
- **Syntax:** `public class JFileChooser extends JComponent implements Accessible`

JFileChooser Constructors:

- **`JFileChooser()`:** Constructs a JFileChooser pointing to the user's default directory.
- **`JFileChooser(File currentDirectory)`:** Constructs a JFileChooser using the given File as the path.
- **`JFileChooser(String currentDirectoryPath)`:** Constructs a JFileChooser using the given path.

JFileChooser Methods:

- **`setFileSelectionMode(JFileChooser.SelectAnyVariable)`:** With this method we can limit the user to select either variable
  Directories only (JFileChooser.DIRECTORIES_ONLY) or
  Files only (JFileChooser.FILES_ONLY) or
  Files and Directories (JFileChooser.FILES_AND_DIRECTORIES).
  The default is FILES_ONLY.
- **`showOpenDialog(null)`:** Activates a File open dialog box.
- **`showSaveDialog(null)`:** Activates a file save dialog box.

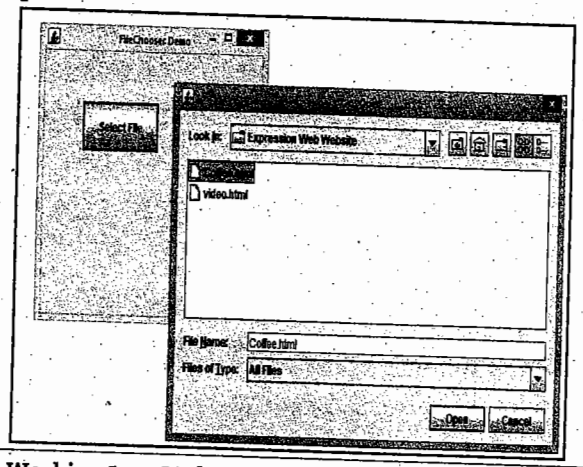
Note: When user selects positive option such as "Yes" then **JFileChooser.APPROVE_OPTION** is returned

Program 9.37: Java Program for File open dialog box: JFileChooser.

```
import java.awt.event.*;
import java.io.*;
public class FileChooserExample extends JFrame implements ActionListener
{
    JButton b1;
    FileChooserExample()
    {
        b1=new JButton("Select File");
        b1.addActionListener(this);
        setLayout(null);
        b1.setBounds(50,50,100,50);
        add(b1);

        //Setting Frame Property
        setSize(300,300);
        setTitle("FileChooser Demo");
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }// Constructor closed
    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource()==b1)
        {
            JFileChooser fc=new JFileChooser();
            int i=fc.showOpenDialog(this);
            if(i==JFileChooser.APPROVE_OPTION)
            {
                JOptionPane.showMessageDialog(null,"This is File Open Dialog
                Box");
```

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

Output:



10. Working Java JColorChooser:

- The JColorChooser class is used to create a color chooser dialog box so that user can select any color. It inherits JComponent class.
- It is available in javax.swing.JColorChooser class.
- **Syntax:** `public class JColorChooser extends JComponent implements Accessible`

JColorChooser Constructors:

- **`JColorChooser()`:** It is used to create a color chooser panel with white color initially.
- **`JColorChooser(color initialcolor)`:** It is used to create a color chooser panel with the specified color initially.

JColorChooser Methods:

- **`void addChooserPanel(AbstractColorChooserPanel panel)`:** It is used to add a color chooser panel to the color chooser.
- **`static Color showDialog(Component c, String title, Color initialColor)`:** It is used to show the color chooser dialog box.

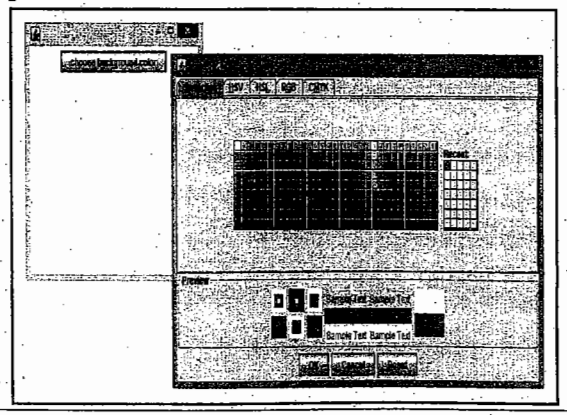
Program 9.38: Java Program for JFileChooser Class.

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
```



```
public class colors extends JFrame
{
    Container c;
    public colors()
    {
        c=getContentPane();
        JButton colorButton=new JButton("choose background color");
        setSize(300,300);
        c.setBackground(Color.white);
        c.setLayout(new FlowLayout());
        c.add(colorButton);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        colorButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                Color bgColor=JColorChooser.showDialog(null,"choose background
                color",c.getBackground());
                if(bgColor!=null)
                c.setBackground(bgColor);
            }
        });
    }
    public static void main(String a[])
    {
        new colors();
    }
}
```

Output:



Summary

- GUI plays an important role in developing interactive applications.
- In java there are many components and containers available to develop GUI, such as AWT components and Swing components etc.
- The Component are the classes which can be implemented in GUI application such as Button, TextField , Label etc.
- Every component has some specific methods which represents the properties of component such as setText(), getText(), setTitle() etc.
- The container is an area where components are placed for developing the applications. There are various containers available in java such as Frame, Applet etc.
- For arranging the components on the container different Layout Managers are used.
- Layout Manager is a technique used to arrange the components on the container in specific order.
- There are different types of Layout Manager:

  FlowLayout: It is default Layout manager which keeps the components line wise on the container.

  GridLyout: It keeps the components in rows and columns.

  BorderLayout: It keeps the components in all the direction on the border of container.
- The setLayout() method is used to set the layout to the container.
- The components can be added on container without layout manager this can be achieved by using setBounds() method.
- Applet is a web based application which can be executed inside java browser.
- Applet works as a container.
- There are two types of an applet:
  - Local
  - Remote
- Menus can be created by using JMenu, JMenuItem and JMenubar classes.

Check Your Understanding

1. Where are the following four methods commonly used?
   (i) public void add(Component c)
   (ii) public void setSize(int width,int height)
   (iii) public void setLayout(LayoutManager m)
   (iv) public void setVisible(boolean)
   (a) Graphics class (b) Component class
   (c) Both (a) and (b) (d) None of the above

2. Which is the container that doesn't contain title bar and MenuBars but it can have other components like button, textfield etc?
   (a) Window (b) Frame
   (c) Panel (d) Container
3. Give the abbreviation of AWT?
   (a) Applet Windowing Toolkit (b) Abstract Windowing Toolkit
   (c) Absolute Windowing Toolkit (d) None of the above
4. Which object can be constructed to show any number of choices in the visible window?
   (a) Labels (b) Choice
   (c) List (d) Checkbox
5. Which is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions?
   (a) Window (b) Panel
   (c) Frame (d) Container
6. Which method can set or change the text in a Label?
   (a) setText() (b) getText()
   (c) All the above (d) None of the above
7. Which are passive controls that do not support any interaction with the user?
   (a) Choice (b) List
   (c) Label (d) Checkbox
8. Which method executes only once ____.
   (a) start() method (b) init() method
   (c) stop() method (d) destroy() method
9. What does the following line of code do? Textfield text = new Textfield(10);
   (a) Creates text object that can hold 10 rows of text.
   (b) Creates text object that can hold 10 columns of text.
   (c) Creates the object text and initializes it with the value 10.
   (d) The code is illegal.
10. Which of the following methods can be used to remove a java.awt.Component object from the display?
    (a) hide() (b) delete()
    (c) remove() (d) disappear()

Answers

1. (b)	2. (c)	3. (b)	4. (c)	5. (c)	6. (a)	7. (c)	8. (b)	9. (b)	10. (a)

Practice Questions

Q.I Answer the following Questions in short:

1. What is the use of Layout manager?
2. Which method is used to specify containers layout? Write its syntax.
3. What is a difference between paint () and repaint ().
4. "An applet class can have a constructor." State whether True / False and justify.
5. Why are swing components called "lightweight" components ?
6. List the mandatory attributes of the APPLET tag.
7. What is the default layout for Frame and Panel?
8. Explain the purpose of getContentPane().

Q.II Answer the following Questions:

1. What are the different types of dialogs available in Java? Explain any two in detail.
2. Explain lifecycle of an Applet.
3. What is Layout Manager? Explain any one in detail.
4. What is AWT? Explain any four classes under java.awt package.
5. Explain any four components used in Swing.
6. Write a note on Dialog control in java.
7. What is the purpose of JTable? Explain with example.
8. What is container? What are different types of containers used in Java?
9. Explain basic methods of Component class.

Q.III Write short note on:

1. Component
2. working with Menu in java JButton
3. Dialog control
4. Layout Manager
5. Applet

■■■

10... Event Handling

Objectives...

- To learn Event-Driven Programming in Java.
- To study Event-Handling Process in detail.
- To learn Event Handling Mechanism.
- To understand the Delegation Model of Event Handling.
- To get knowledge about Event Classes, Event Sources, Event Listeners.

10.1 INTRODUCTION

- Event driven programming is the programming paradigm in the field of computer science. In this type of programming, flow of execution is determined by the events like user clicks or other programming threads or query result from database. Events are handled by event handlers or event call backs.
- In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected. Event-driven programs can be written in any programming language, although the task is easier in languages that provide high-level abstractions, such as await and closures.
- For Example: Consider the case of an Alarm, such as might be part of an AlarmClock system. The Alarm receives two kinds of signals: SIGNAL_TIMEOUT, which indicates that it is time for the Alarm to start ringing, and SIGNAL_RESET, which indicates that it is time for the Alarm to stop. We might implement this using two methods, handleTimeout() and handleReset().

```
public class Alarm
{
   Buzzer bzzz =  new Buzzer();
   public void handleTimeout()
   {
      this.bzzz.startRinging();
   }
```

```
    public void handleReset()
    {
        this.bzzz.stopRinging();
    }
}
```

10.2 EVENT-DRIVEN PROGRAMMING IN JAVA

- The basic idea of **event-driven programming** is simply to create objects with methods that handle the appropriate events or circumstances, without explicit attention to how or when these methods will be called.
- Event-driven programming serves the user with the quickest and most accurate responses and this usually translates into better user experience and business gains.
- Changing the state of an object is known as an **Event**. For example, click on button, dragging mouse etc.
- The **java.awt.event** package provides many event classes and Listener interfaces for event handling.
- When an event occurs then the event handler is activated which will invoke an appropriate action against the event.
- The factor that makes writing GUIs challenging is that the path of code execution becomes nondeterministic.
- When a GUI program is running, the user can click any of the buttons and interact with any of the other onscreen components in any order. Because the program's execution is driven by the series of events that occur, we say that programs with GUIs are ***event-driven***. In this chapter you'll learn how to handle user events so that your event-driven graphical programs will respond appropriately to user interaction.
- For example, Consider Event is switch on the button of light. Event Action Light get switched on.
- Events are an important part in any GUI program. All GUI applications are event-driven. An application reacts to different event types which are generated during its execution. Events are generated mainly by the user of an application. But they can be generated by other means as well. e.g. internet connection, window manager, timer.
- In the event model, there are three participants: Event source, Event object, Event listener.
    - **The Event source** is the object whose state changes. It generates Events.
    - **The Event object** (Event) encapsulates the state changes in the event source.
    - **The Event listener** is the object that wants to be notified. Event source object delegates the task of handling an event to the event listener.

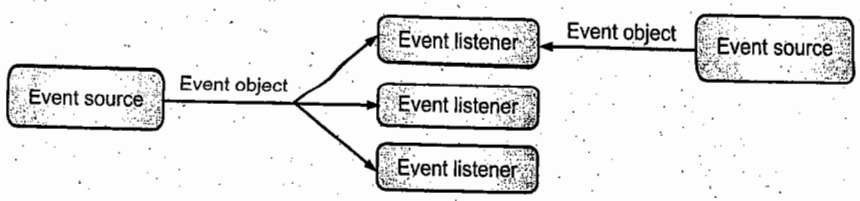


Fig. 10.1: Working of Event Source Object and Listeners

- **Java has two types of events:**
    1. **Low-Level Events:** Low-level events represent direct communication from user. A low level events are key press, key release, mouse click, drag, move or release, and so on. Following are low level events.

Table 10.1: Low –Level Events

Event	Description
`ComponentEvent`	Indicates that a component object (e.g. Button, List, TextField) is moved, resized, rendered invisible or made visible again.
`FocusEvent`	Indicates that a component has gained or lost the input focus.
`KeyEvent`	Generated by a component object (such as TextField) when a key is pressed, released or typed.
`MouseEvent`	Indicates that a mouse action occurred in a component. E.g. mouse is pressed, releases, clicked (pressed and released), moved or dragged.
`ContainerEvent`	Indicates that a container's contents are changed because a component was added or removed.
`WindowEvent`	Indicates that a window has changed its status. This low level event is generated by a Window object when it is opened, closed, activated, deactivated, iconified, deiconified or when focus is transferred into or out of the Window.

    2. **High-Level Events:** High-level events (also called as semantic events) encapsulate the meaning of a user interface component. These include following events.

Table 10.2: High –Level Events

Event	Description
`ActionEvent`	Indicates that a component-defined action occurred. This high-level event is generated by a component (such as Button) when the component-specific action occurs (such as being pressed).
`AdjustmentEvent`	The adjustment event is emitted by Adjustable objects like scrollbars.
`ItemEvent`	Indicates that an item was selected or deselected. This high-level event is generated by an ItemSelectable object (such as a List) when an item is selected or deselected by the user.
`TextEvent`	Indicates that an object's text changed. This high-level event is generated by an object (such as TextComponent) when its text changes.

10.3 THE EVENT-HANDLING PROCESS

- Event Handling provides four types of classes, they are:
  1. Event Adapters
  2. Event classes
  3. Event Sources
  4. Event Listeners

1. **Event Adapter:** In a program, when a listener has many abstract methods to override, it becomes complex for the programmer to override all of them.

   For example, for closing a frame, we must override seven abstract methods of WindowListener(), but we need only one method of them.

   For reducing complexity, Java provides a class known as "adapters" or adapter class. Adapters are abstract classes that allows user to implement only one method from the list of available abstract methods.

2. **Event classes:** Every event source generates an event and is named by a Java class. An event is generated when something changes within a graphical user interface.

   For example, the event generated by:
   - Button is known as an ActionEvent.
   - Checkbox is known as an ItemEvent.
   - All of the events are listed in the **java.awt.event** package.

3. **Event Sources:** Event Sources are responsible for generating events and are called components. The source for an event can be a Button, TextField or a Frame etc.

4. **Event Listeners:** Events are handled by a special group of interfaces, known as "listeners". Listener listens the event and take appropriate action. Example, Click event can be handled by ActionListener.

Steps involved in event handling:

1. The user clicks the event component such as Button and the event is generated.
2. Now the object of concerned event class is created automatically and information about the source and the event get populated within same object.
3. Event object is forwarded to the method of registered listener class.
4. The appropriate event method will get executed.

Steps to implement Event in Java program:

Step 1 : Select event package using `import java.awt.event.*;`

Step 2 : Select appropriate event component and event interface.

For example: *Consider Button component,* the Event interface for Button is ActionListener

```
class EventDemo extends JFrame implements ActionListener
```


Step 3 : Register the event component with EventListener.

```
JButton b1=new JButton("ok"); //Event component
b1. addActionListener(this); //Register component with event listener
```

Step 4 : Implement the interface method for event action execution. Every EventListener interface has its own event method. For ActionListener interface, actionPerformed(ActionEvent) is used. It should be implemented after constructor.

For Example:

```
public void actionPerformed(ActionEvent ae)
{  //write event action code here }
```

10.4 THE EVENT-HANDLING MECHANISM

- The Event Handling can be done in following classes.
  - The same class
  - Another class
  - Anonymously

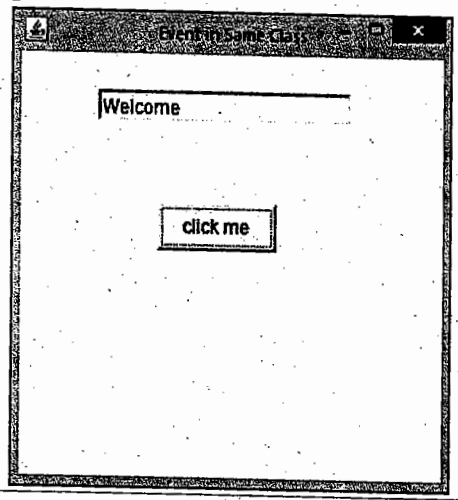
1. **Implementing event in same class:** The event can be implemented within same class.

Program 10.1: Implementing event in same class.

```
import java.awt.*;
import java.awt.event.*;
class DemoEvent extends Frame implements ActionListener
{
   TextField tf;
   DemoEvent()
   {
      //create components
      tf=new TextField();
      tf.setBounds(60,50,170,20);
      Button b=new Button("click me");
      b.setBounds(100,120,80,30);

      //register listener
      b.addActionListener(this);//"this" passing current instance

      //add components and set size, layout and visibility
      add(b);add(tf);
      setSize(300,300);
      setLayout(null);
      setTitle("Event in Same Class");
      setVisible(true);
   }
```

```
    public void actionPerformed(ActionEvent e)
    {
       tf.setText("Welcome");
    }
    public static void main(String args[])
    {
       new DemoEvent();
    }
}
```

Output:



2. **Implementing event using another class:** The event can also be implemented using different class.

Program 10.2: Implementing event using another class.

Note:

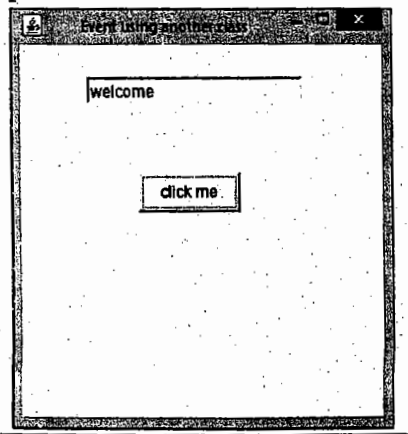
- Create two separate files DemoEvent.java and Outer.java
- First compile DemoEvent.java and then compile Outer.java
- Execute program using DemoEvent.class

```
//Class Using another class for registering listener
import java.awt.*;
import java.awt.event.*;
class DemoEvent extends Frame
{
   TextField tf;
   DemoEvent()
   {
      //create components
      tf=new TextField();
      tf.setBounds(60,50,170,20);
```



```
      Button b=new Button("click me");
      b.setBounds(100,120,80,30);
      //register listener  Note: here we use Outer class to register event
      Outer o=new Outer(this);
      b.addActionListener(o);//passing outer class instance
      //add components and set size, layout and visibility
      add(b);add(tf);
      setSize(300,300);
      setLayout(null);
      setTitle("Event using another class");
      setVisible(true);
      }
      public static void main(String args[])
      {
         new DemoEvent();
      }
}
//Another Event Class : This class is used to register the component and
implementing method
import java.awt.event.*;
class Outer implements ActionListener
{
   DemoEvent obj;
   Outer(DemoEvent obj)
   {
      this.obj=obj;
   }
   public void actionPerformed(ActionEvent e)
   {
   obj.tf.setText("welcome");
   }
}
```

Output:



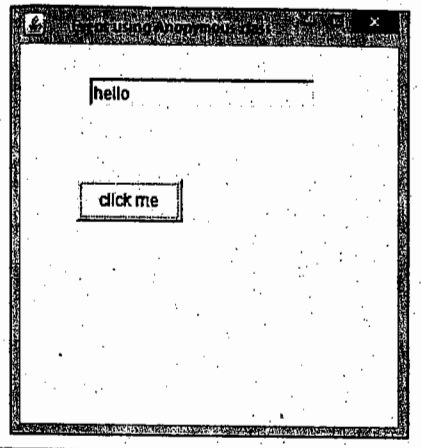
3. **Implementing event using anonymous class:** The event can also be implemented using anonymous class.

Program 10.3: Implementing event in using anonymous class.

Note: In this example we are implementing event method within the constructor only.

```
import java.awt.*;
import java.awt.event.*;
class DemoEvent extends Frame
{
   TextField tf;
   DemoEvent()
   {
      tf=new TextField();
      tf.setBounds(60,50,170,20);
      Button b=new Button("click me");
      b.setBounds(50,120,80,30);

      //implementing event method
      b.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent ae)
         {
            tf.setText("hello");
         }
      });
      add(b);add(tf);
      setSize(300,300);
      setLayout(null);
      setTitle("Event using Anonymous class");
      setVisible(true);
   }
   public static void main(String args[])
   {
      new DemoEvent();
   }
}
```


Output:



10.5 THE DELEGATION MODEL OF EVENT HANDLING

- In the delegation event model, specific objects are designated as event handlers for GUI components.
- Event handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism is known as **event handler.**
- Java uses the Delegation Event Model to handle the events. It defines standard and consistent mechanisms to generate and process events.
- Delegation Event model has quite simple concept i.e. a source generates an event and sends it to one or more listeners.
- The key participants of delegation event model of event handling are:
  - **Source:** An event source is an object which originates or "fires" event.
  - **Listener:** It is an object that generate responses for a specific event.
  - **Object:** An object describe the event for a particular class.
- In this model, a listener needs to be registered with the source object so that the listener can receive the event notification. This is one of the efficient way of handling the event.
- A component generates an event (such as mouse click, key press, etc) ,event source is activated and current active event object is passed to the event listener, event listener checks the event object and executes an appropriate method.

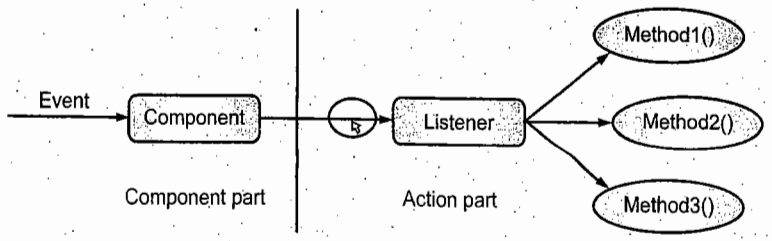


Fig. 10.2: Event Delegation Model

10.6 EVENT CLASSES, EVENT SOURCES, EVENT LISTENERS

- Java supports many event classes, event sources and event listeners. They are explained below,

10.6.1 Event Classes

- Event Classes are the classes that represent events in Java's event handling mechanism.
- Every event is represented as Event Class.
- EventObject is the super class of all events. It is in **java.util** package.
- Its constructor is: `EventObject (Object src)`
  Here, src is the objects that generates the event.
- EventObject class contains two methods: getSource() and toString()
  (i) `Object getSource()`: The getSource() method returns the source of the event.
  (ii) `String toString()`: It returns the string equivalent of the event.

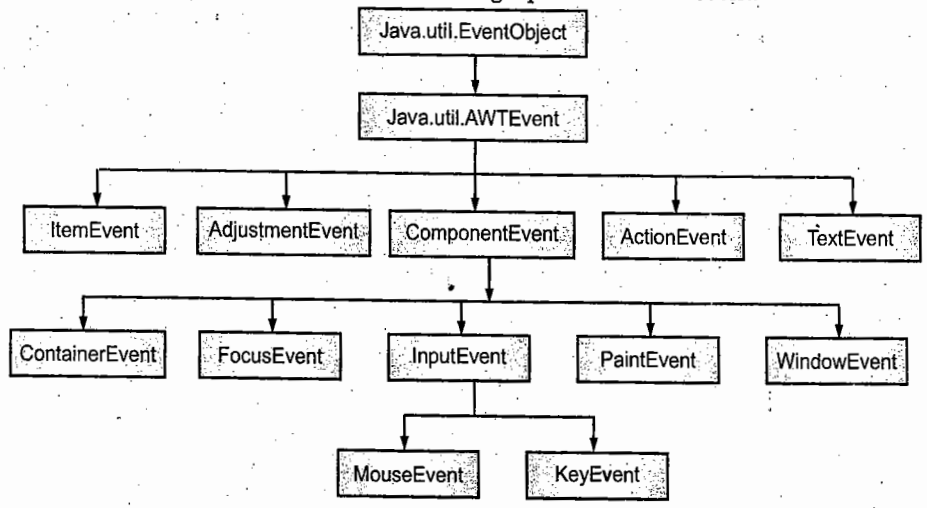


Fig. 10.3: Event class Hierarchy

- The class AWTEvent, defined within the **java.awt** package, is a subclass of EventObject. It is the superclass of all AWT-based events used by delegation event model. The **java.awt.event** package supports all type of event classes.
- The commonly used event classes are listed in bellow table.

Table 10.3: Commonly Used Event Classes in java.awt.event

Event Class	Description
`ActionEvent`	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
`AdjustmentEvent`	Generated when a scroll bar is manipulated.

contd. ...

`ComponentEvent`	Generated when a component is hidden, moved, resized, or becomes visible.
`ContainerEvent`	Generated when a component is added to or removed from a container.
`FocusEvent`	Generated when a component gains or losses keyboard focus.
`InputEvent`	Abstract superclass for all component input event classes.
`ItemEvent`	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
`KeyEvent`	Generated when input is received from the keyboard.
`MouseEvent`	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
`MouseWheelEvent`	Generated when the mouse wheel is moved.
`TextEvent`	Generated when the value of a text area or text field is changed.
`WindowEvent`	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

10.6.2 Event Source

- A source is an object that generates an event. This occurs when object changes in some way.
- Source may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- **Syntax:**

```
public void addTypeListener (TypeListener el)
   or
public void addTypeListener (this)   //this refers to the current object
```

- When an event occurs, all registered listeners are notified and receive a copy of the event object. It is called Multicasting the event.
- Some sources may allow only one listener to register.
- General form:

```
public void addTypeListener (TypeListener el) throws
                                    java.util.TooManyListenersException
```

- A Listener can also be **removed** by using following command:

```
public void removeTypeListener (TypeListener el)
```

- For adding various components, we use public methods, for example:
  - **Using Button Event**

```
void addActionListener(ActionListener a)
```

  - **Using List**

```
void addActionListener(ActionListener a)
void addItemListener(ItemListener a)
```

- For Example:

```
JButton b1=new JButton("Submit"); //declare a event component
b1.addActionListener(this); //register the event component
```

10.6.3 Event Listener

- A listener is an object that is notified when an event occurs.
- Listener must have registered with one or more sources to receive notifications about specific types of events.
- Once the listener object is registered it must implement methods to receive and process these event notifications.
- The Event listener represents the interfaces responsible to handle events. Java provides us various Event listener classes.
- Every method of an event listener has a single argument as an object which is subclass of EventObject class.
- For example, mouse event listener methods will accept instance of MouseEvent, where MouseEvent derives from EventObject.

Event Listener Interface and Methods:

Table 10.4: Event Listener Interface and Methods

Listener Interface	Methods
ActionListener	void actionPerformed (ActionEvent ae)
AdjustmentListener	void adjustmentValueChanged (AdjustmentEvent ae)
ComponentListener	void componentResized (ComponentEvent ce) void componentMoved (ComponentEvent ce) void componentShown (ComponentEvent ce) void componentHidden (ComponentEvent ce)
ContainerListener	void componentAdded (ContainerEvent ce) void componentRemoved (ContainerEvent ce)
FocusListener	void focusGained (FocusEvent fe) void focusLost (FocusEvent fe)
ItemListener	void itemStateChanged (ItemEvent ie)

contd. ...

KeyListener	void keyPressed (KeyEvent ke) void keyReleased (KeyEvent ke) void keyTyped (KeyEvent ke)
MouseListener	void mouseClicked (MouseEvent me) void mouseEntered (MouseEvent me) void mouseExited (MouseEvent me) void mousePressed (MouseEvent me) void mouseReleased (MouseEvent me)
MouseMotionListener	void mouseDragged (MouseEvent me) void mouseMoved (MouseEvent me)
TextListener	void textChanged (TextEvent te)
WindowListener	void windowActivated (WindowEvent we) void windowClosed (WindowEvent we) void windowClosing (WindowEvent we) void windowDeactivated (WindowEvent we) void windowDeiconified (WindowEvent we) void windowIconified (WindowEvent we) void windowOpened (WindowEvent we)

- The following table lists the events, their corresponding listeners and the method to add the listener to the component.

Table 10.5: Events, their corresponding Listeners and Methods

Event	Event Source	Event Listener	Method to add listener to event source
Low-level Events			
ComponentEvent	Component	ComponentListener	addComponentListener()
FocusEvent	Component	FocusListener	addFocusListener()
KeyEvent	Component	KeyListener	addKeyListener()
MouseEvent	Component	MouseListener MouseMotionListener	addMouseListener() addMouseMotionListener()
ContainerEvent	Container	ContainerListener	addContainerListener()
WindowEvent	Window	WindowListener	addWindowListener()

contd. ...

High-level Events			
ActionEvent	Button List MenuItem TextField	ActionListener	addActionListener()
ItemEvent	Choice CheckBox CheckBoxMenuItem List	ItemListener	addItemListener()
AdjustmentEvent	Scrollbar	AdjustmentListener	addAdjustmentListener()
TextEvent	TextField TextArea	TextListener	addTextLIstener()

Table 10.6: Listeners and methods used in Event handling

Listener	Event Handling Methods	Description
ItemListener	public Object getItem()	Returns the item which was clicked and triggered the ItemEvent.
	public ItemSelectable getItemSelectable()	Returns the item which was clicked.
	public int getStateChange()	Returns the current state of item which was clicked.
ActionListener	public String getActionCommand()	Returns the name over the button, item or menuitem that was clicked to trigger the ActionEvent.
	public long getWhen()	Returns the time when an ActionEvent was generated.
	Component getSource()	Returns the object on event component
KeyListener	public char getKeyChar()	Returns the character associated with the key pressed on the keyboard, which triggered the KeyEvent.
	public int getKeyCode()	Returns an int key code associated with the key pressed on the keyboard.
	public boolean isActionKey()	Returns true if key pressed was an "action" key, i.e. keys that don't generate a character, such as Cut, Copy, Paste, Page Up, Caps Lock, the arrow and function keys.

contd. ...

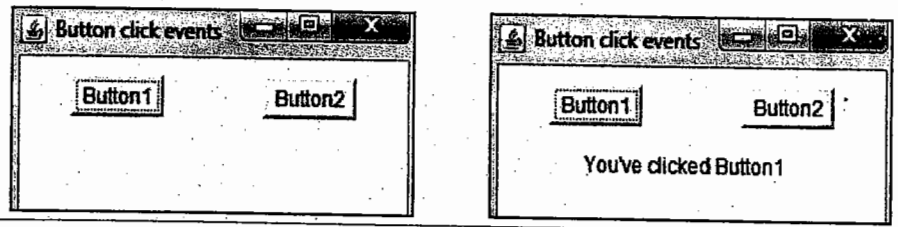
MouseListener	public int getX()	Returns the x-coordinate of the MouseEvent.
	public int getY()	Returns the y-coordinate of the MouseEvent.
MouseMotionListener	public Point getLocationOnScreen()	Returns the absolute x, y position of the MouseEvent.
	public int getX()	Returns the x-coordinate of the MouseEvent.
	public int getY()	Returns the y-coordinate of the MouseEvent.
WindowListener	public Window getWindow()	Returns the window which triggered the WindowEvent.
	public int getNewState()	Returns the new state of the window.
AdjustmentListener	public int getValue()	Gets the Scrollbar's current position value.
	public int getAdjustmentType()	Returns the type of adjustment that caused AdjustmentEvent.
TextListener	public String paramString()	Returns the String describing the TextEvent.

Program 10.4: Program to demonstrate ActionListener on Button.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ActionEx1 implements ActionListener
{
    Frame jf;
    Button button1, button2;
    Label label;

    ActionEx1()
    {
        jf= new Frame("Button click events");
        button1= new Button("Button1");
        button2= new Button("Button2");
        label = new Label();

        jf.add(button1);
        jf.add(button2);
        jf.add(label);
```

```
        //Registering the class ActionEx1 to listen to ActionEvent, when
        buttons are clicked.
        button1.addActionListener(this); //this represents current object of
        ActionEx1 class.
        button2.addActionListener(this);

        jf.setLayout(new FlowLayout(FlowLayout.CENTER,60,10));
        jf.setSize(250,150);
        jf.setVisible(true);
    }

    public void actionPerformed(ActionEvent ae)
    {
        if(ae.getActionCommand().equals("Button1"))
        {
            label.setText("You've clicked Button1");
            jf.add(label);
            jf.setVisible(true);
        }

        if(ae.getActionCommand().equals("Button2"))
        {
            label.setText("You've clicked Button2");
            jf.add(label);
            jf.setVisible(true);
        }
    }

    public static void main(String args[])
    {
        new ActionEx1();
    }
}
```

Output:



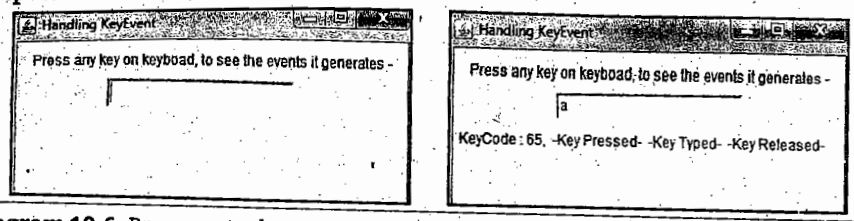
Program 10.5: Program to demonstrate KeyListener.

```
import java.awt.*;
import java.awt.event.*;
public class KeyEx1 implements KeyListener
{
    Label label1, label2;
    TextField field1;
```



```
    Frame jf;
    String str;
    KeyEx1()
    {
        jf = new Frame("Handling KeyEvent");
        label1= new Label("Press any key on keyboard, to see the events it
        generates -", Label.CENTER);
        label2= new Label();
        field1 = new TextField(20); //calling TextField(String)
        jf.setLayout(new FlowLayout());
        jf.add(label1);
        jf.add(field1);
        jf.add(label2);
        field1.addKeyListener(this); //As soon as button is clicked, data
        from all the textfields is read
        jf.setSize(360,200);
        jf.setVisible(true);
    }
    public void keyPressed(KeyEvent ke)
    {
        str= "KeyCode : " + ke.getKeyCode() + ",  -Key Pressed- ";
        label2.setText(str);
        jf.setVisible(true);
    }
    public void keyReleased(KeyEvent ke)
    {
        str+=" -Key Released- ";
        label2.setText(str);
        jf.setVisible(true);
        str="";
    }
    public void keyTyped(KeyEvent ke)
    {
        str+=" -Key Typed- ";
        label2.setText(str);
        jf.setVisible(true);
    }
    public static void main(String args[])
    {
        new KeyEx1();
    }
}
```

Output:



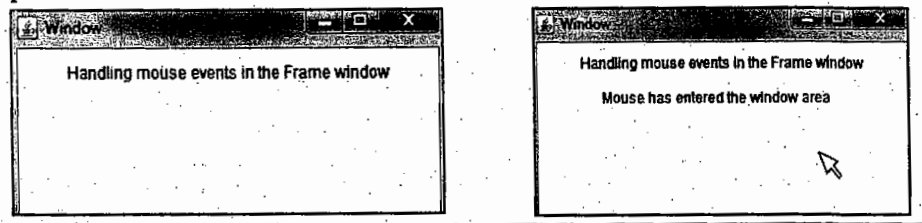
Program 10.6: Program to demonstrate MouseListener.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class MouseEx1 implements MouseListener
{
   Label label1, label2;
   Frame frame;
   String str;

   MouseEx1()
   {
      frame = new Frame("Window");
      label1= new Label("Handling mouse events in the Frame window",
      Label.CENTER);
      label2= new Label();
      frame.setLayout(new FlowLayout());
      frame.add(label1);
      frame.add(label2);

      //Registering class MouseEx1 to catch and respond to mouse events
      frame.addMouseListener(this);

      frame.setSize(340,200);
      frame.setVisible(true);
   }
   public void mouseClicked(MouseEvent we)
   {
      str+=" and Mouse button was clicked";
      label2.setText(str);
      frame.setVisible(true);
   }
   public void mouseEntered(MouseEvent we)
   {
      label2.setText("Mouse has entered the window area");
      frame.setVisible(true);
   }
```



```
   public void mouseExited(MouseEvent we)
   {
      label2.setText("Mouse has exited the window area");
      frame.setVisible(true);
   }
   public void mousePressed(MouseEvent we)
   {
      label2.setText("Mouse button is being pressed");
      frame.setVisible(true);
   }
   public void mouseReleased(MouseEvent we)
   {
      str="Mouse button is released";
      label2.setText(str);
      frame.setVisible(true);
   }
   public static void main(String args[])
   {
      new MouseEx1();
   }
}
```

Output:

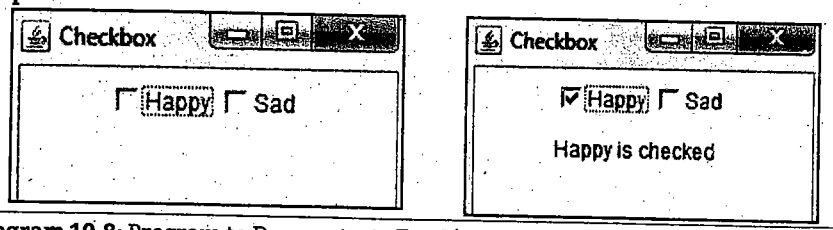


Program 10.7: Program to demonstrate ItemListener.

```
import java.awt.*;
import java.awt.event.*;
public class ItemEx1 implements ItemListener
{
   Frame jf;
   Checkbox chk1, chk2;
   Label label1;
   ItemEx1()
   {
      jf= new Frame("Checkbox");
      chk1 = new Checkbox("Happy");
      chk2 = new Checkbox("Sad");
      label1 = new Label();
```

```
        jf.add(chk1);
        jf.add(chk2);

        chk1.addItemListener(this);
        chk2.addItemListener(this);

        jf.setLayout(new FlowLayout());
        jf.setSize(220,150);
        jf.setVisible(true);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        Checkbox ch =(Checkbox)ie.getItemSelectable();
        if(ch.getState()==true)
        {
            label1.setText(ch.getLabel()+ " is checked");
            jf.add(label1);
            jf.setVisible(true);
        }
        else
        {
            label1.setText(ch.getLabel()+ " is unchecked");
            jf.add(label1);
            jf.setVisible(true);
        }
    }
    public static void main(String args[])
    {
        new ItemEx1();
    }
}
```

Output:



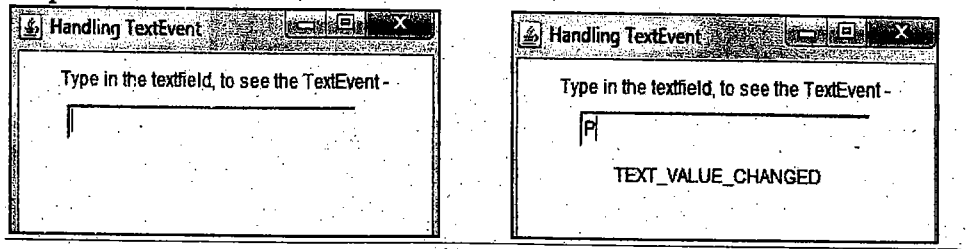
Program 10.8: Program to Demonstrate TextListener.

```
import java.awt.*;
import java.awt.event.*;

public class TextEventEx1 implements TextListener
{
    Label label1, label2;
    TextField field1;
    Frame jf;
    String str;
```



```
    TextEventEx1()
    {
        jf = new Frame("Handling TextEvent");
        label1= new Label("Type in the textfield, to see the textevents it
        generates -", Label.CENTER);
        label2= new Label();
        field1 = new TextField(25);

        jf.setLayout(new FlowLayout());

        jf.add(label1);
        jf.add(field1);
        jf.add(label2);

        //Registering the class TextEventEx1 to catch and respond to mouse
        text events
        field1.addTextListener(this);

        jf.setSize(340,200);
        jf.setVisible(true);
    }
    public void textValueChanged(TextEvent te)
    {
        label2.setText(te.paramString());
        jf.setVisible(true);
    }

    public static void main(String args[])
    {
        new TextEventEx1();
    }
}
```

Output:



Program 10.9: Program to Demonstrate ActionListener using checkbox to change background color.

```
import java.awt.*;
import java.awt.event.*;

public class checkbox extends Frame implements ActionListener
{
      Button r,g,b;
      public checkbox()
      {     r=new Button("red");
            g=new Button("green");
            b=new Button("blue");
            setLayout(new FlowLayout());
            add(r);
            add(g);
            add(b);
            r.addActionListener(this);
            g.addActionListener(this);
            b.addActionListener(this);
      }
      public static void main(String args[])
      {
            checkbox c=new checkbox();
            c.setSize(350,350);
            c.setVisible(true);
            c.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
            {
                  System.exit(0);
            }
            });
      }
      public void actionPerformed(ActionEvent e)
      {     if(e.getSource()==r)
            {
                  setBackground(Color.red);
            }
            else
            if(e.getSource()==g)
            {
                  setBackground(Color.green);
            }
            else
            {
                  setBackground(Color.blue);
            }
      }
}
```

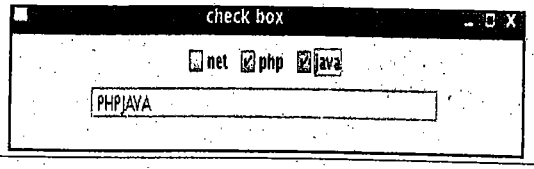
- After executing above program you will receive a Frame with three buttons red, green and blue. After clicking on each button Frame background color will get changed as per the button color.

Program 10.10: Program to Demonstrate ItemListener using checkbox.

```
import javax.swing.*;
public class CheckBoxDemo extends JFrame
{
      JCheckBox net,php,java;
      JTextField msg;
      public CheckBoxDemo()
      {     net=new JCheckBox("net");
            php=new JCheckBox("php");
            java=new JCheckBox("java");
            msg=new JTextField(30);
            setTitle("check box");
            setLayout(new FlowLayout());
            setSize(500,100);
            add(net);
            add(php);
            add(java);
            add(msg);
            setVisible(true);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            CheckBoxHandler cbh=new CheckBoxHandler();
            net.addItemListener(cbh);
            php.addItemListener(cbh);
            java.addItemListener(cbh);
      }
      public static void main(String args[])
      {
            new CheckBoxDemo();
      }
      private String s1=" ",s2=" ",s3=" ";
      class CheckBoxHandler implements ItemListener
      {
            public void itemStateChanged(ItemEvent ie)
            {
                  if(ie.getSource()==net)
                  {
                        if(ie.getStateChange()==ItemEvent.SELECTED)
                              s1="NET";
                        else
                              s1=" ";
                  }
                  if(ie.getSource()==php)
                  {
                        if(ie.getStateChange()==ItemEvent.SELECTED)
                              s2="PHP";
                        else
                              s2=" ";
                  }
                  if(ie.getSource()==java)
                  {
```

```
                if(ie.getStateChange()==ItemEvent.SELECTED)
                    s3="JAVA";
                else
                    s3=" ";
            }
            msg.setText(s1+s2+s3);
        }
    }
}
```

Output:



10.7 ADAPTER CLASSES AS HELPER CLASSES IN EVENT HANDLING

- Adapter Class provides an empty implementation of all methods in an event listener interface.
- Useful when you want to listen and process only some of the events that are handled by one particular event listener interface.
- Define your own class as subclass of this class and provide desired implementation.
- There are some event listeners that have multiple methods to implement i.e. some of the listener interfaces contain more than one method.
- For instance, the MouseListener interface contains five methods such as mouseClicked, mousePressed, mouseReleased etc.
- If you want to use only one method out of these then also you will have to implement all of them as they are abstract by nature. Thus, the methods which you do not want to implement must be defined with empty implementation i.e. with empty bodies.
- To overcome this problem we can use Adapter Class.
- **Syntax:**

```
addListenerName(new AdapterClassName()
{
    public void EventMethod(Event Object)
    {
        //Write Event code
    }
});
```


- Commonly used listener interfaces implemented by Adapter classes:

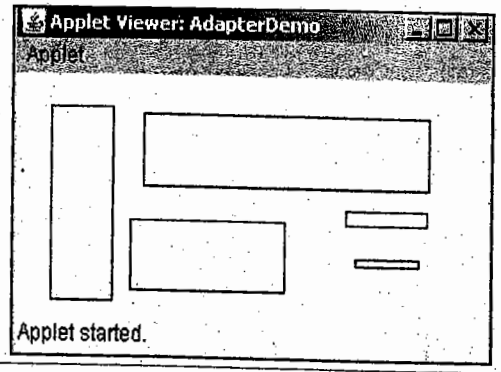
Table 10.7: Listener interfaces implemented by Adapter Classes

Adapter Class	Listener Interfaces
ComponenetAdapter	ComponenetListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Program 10.11: Following program shows use of adapter class by implementing only two methods out of five from MouseListener interface.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class AdapterDemo extends Applet
{
    public void init()
    {
      //Implementing Adapter Class
        addMouseListener(new MouseAdapter()
        {
            int topX, bottomY;
            public void mousePressed(MouseEvent me)
            {
                topX = me.getX();
                bottomY = me.getY();
            }
            public void mouseReleased(MouseEvent me)
            {
                Graphics g = AdapterDemo.this.getGraphics();
                g.drawRect(topX, bottomY, me.getX()-topX, me.getY()-bottomY);
            }
        });
    }
}
```

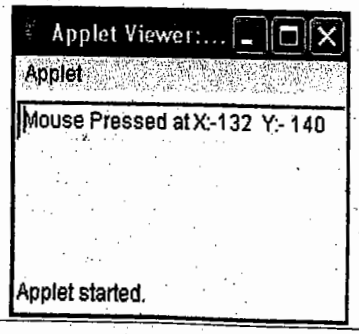
Output:



Program 10.12: Program Using MouseListener.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*<applet height=200 width=200 code="AdapterDemo"></applet>*/
public class AdapterDemo extends Applet
{
   int topX, bottomY;
   TextField tf;
   public void init()
   {
      tf=new TextField(25);
      add(tf);
      addMouseListener(new MouseAdapter()
      {
         public void mousePressed(MouseEvent me)
         {
            topX = me.getX();
            bottomY = me.getY();
            tf.setText("Mouse Pressed at X:-"+topX+"  Y:- "+bottomY);
         }
      });
  }}
```

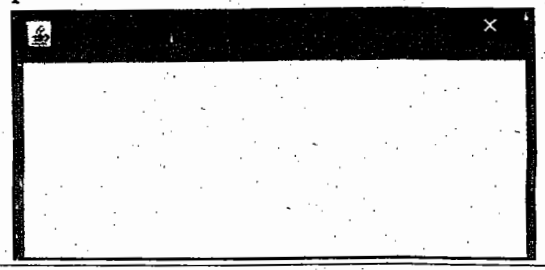
Output:



Program 10.13: Window Listener using Adapter class.

```
import java.awt.*;
import java.awt.event.*;
public class AdapterExample
{
   Frame f;
   AdapterExample()
   {
      f=new Frame("Window Adapter");
      f.addWindowListener(new WindowAdapter()
      {
         public void windowClosing(WindowEvent e)
         {
            f.dispose();
         }
      });
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
   public static void main(String[] args)
   {
      new AdapterExample();
   }
}
```

Output:



Summary

- AWT is a collection of components and container classes. The List of container classes are Panel, Frame, Applet. The List component classes are TextField, Button, Checkbox etc.
- Every component supports to the event. For every event separate event classes are available in java.awt.event package.
- The List of events are: ActionEvent, ItemEvent, MouseEvent,WindowEvent,KeyEvent etc.
- For handling events separate interfaces are available they are known as Listeners. E.g. ItemListener, ActionListener, MouseListener, MouseMotionListener, WindowListener etc.
- Delegation Model of Event Handling focused complete working from event generation till execution
- Adapter Class provides an empty implementation of all methods in an event listener interface. Commonly used listener interfaces implemented by Adapter classes
- Working of Adapter classes and how it is useful in event handling is demonstrated.
- The getSource() method plays very important role in event driven application, it returns reference of a component using which event is generated.
- The getActionCommand() method returns the caption of the event component.

Check Your Understanding

1. Which of these packages contains all the classes and methods required for even handling in Java?
   (a) java.applet (b) java.awt
   (c) java.event (d) java.awt.event
2. What is an event in delegation event model used by Java programming language?
   (a) An event is an object that describes a state change in a source.
   (b) An event is an object that describes a state change in processing.
   (c) An event is an object that describes any change by the user and system.
   (d) An event is a class used for defining object, to create events.
3. Which of these methods are used to register a keyboard event listener?
   (a) KeyListener() (b) addKistener()
   (c) addKeyListener() (d) eventKeyboardListener()
4. Which of these methods are used to register a mouse motion listener?
   (a) addMouse() (b) addMouseListener()
   (c) addMouseMotionListner() (d) eventMouseMotionListener()

5. What is a listener in context to event handling?
   (a) A listener is a variable that is notified when an event occurs
   (b) A listener is a object that is notified when an event occurs
   (c) A listener is a method that is notified when an event occurs
   (d) None of the mentioned
6. Event class is defined in which of these libraries?
   (a) java.io (b) java.lang
   (c) java.net (d) java.util
7. Which of these methods can be used to determine the type of event?
   (a) getID() (b) getSource()
   (c) getEvent() (d) getEventObject()
8. Which of these class is super class of all the events?
   (a) EventObject (b) EventClass
   (c) ActionEvent (d) ItemEvent
9. Which of these events will be notified if scroll bar is manipulated?
   (a) ActionEvent (b) ComponentEvent
   (c) AdjustmentEvent (d) WindowEvent
10. Which of these events will be generated if we close an applet's window?
    (a) ActionEvent (b) ComponentEvent
    (c) AdjustmentEvent (d) WindowEvent

Answers

1. (d)	2. (a)	3. (c)	4. (c)	5. (b)	6. (d)	7. (a)	8. (a)	9. (c)	10. (d)

Practice Questions

Q.I Answer the following Questions in short:

1. Give the syntax and use of method getSource().
2. What is the advantage of using Adapter classes?
3. Write the Syntax and state the purpose of getActionCommamnd()?
4. Write the Syntax and state the purpose of itemStateChanged()?
5. What is Listener?
6. What is Event?
7. Write the Syntax and state the purpose of addListenerType(Listener)?

Q.II Answer the following Questions:

1. Explain any four interfaces used for event handling. Give syntax and suitable example.
2. Write steps for event handling in java.

3. What is an adapter class Give suitable example of implementing MouseListener.
4. Write a note on Event class, Event Source and Event Listener.
5. Explain the process of Event Handling.
6. Explain Event Delegation Model.

Q.III Write short note on:

1. Event Handler in AWT
2. Listener
3. Adapter class
4. Event Driven
5. Event-Handling

■■■

11... The Collection Framework

Objectives...

- To understand the concept of collection framework.
- To study different types of Collection.
- To understand java utilities.
- To understand the concept of hashing.
- To study the implementation of Map interface.

11.1 INTRODUCTION TO JAVA FRAMEWORKS

- The collection framework is the collection of classes and interfaces.
- A Java collection framework provides architecture to store and manipulate a group of objects. A Java collection framework includes the following:
  - Interfaces
  - Classes
  - Algorithm

Interfaces:

- Interface in Java refers to the abstract data types. They allow Java collections to be manipulated independently from the details of their representation. Also, they form a hierarchy in object-oriented programming languages. The List of interfaces are:
  - Collection
  - Set
  - List
  - Map
  - SortedMap
  - Enumeration

Classes:

- Classes in Java are the implementation of the collection interface. It basically refers to the data structures that are used again and again. The List of classes are:
  - ArrayList
  - LinkedList
  - HashSet
  - TreeSet

Algorithm:

- Algorithm refers to the methods which are used to perform operations such as searching and sorting on objects that implement collection interfaces. Algorithms are polymorphic in nature as the same method can be used to take many forms or you can say perform different implementations of the Java collection interface.
- The collections framework was designed to meet several goals, such as:
  - The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and Hashtables) were to be highly efficient.
  - The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - The framework had to extend and/or adapt a collection easily.

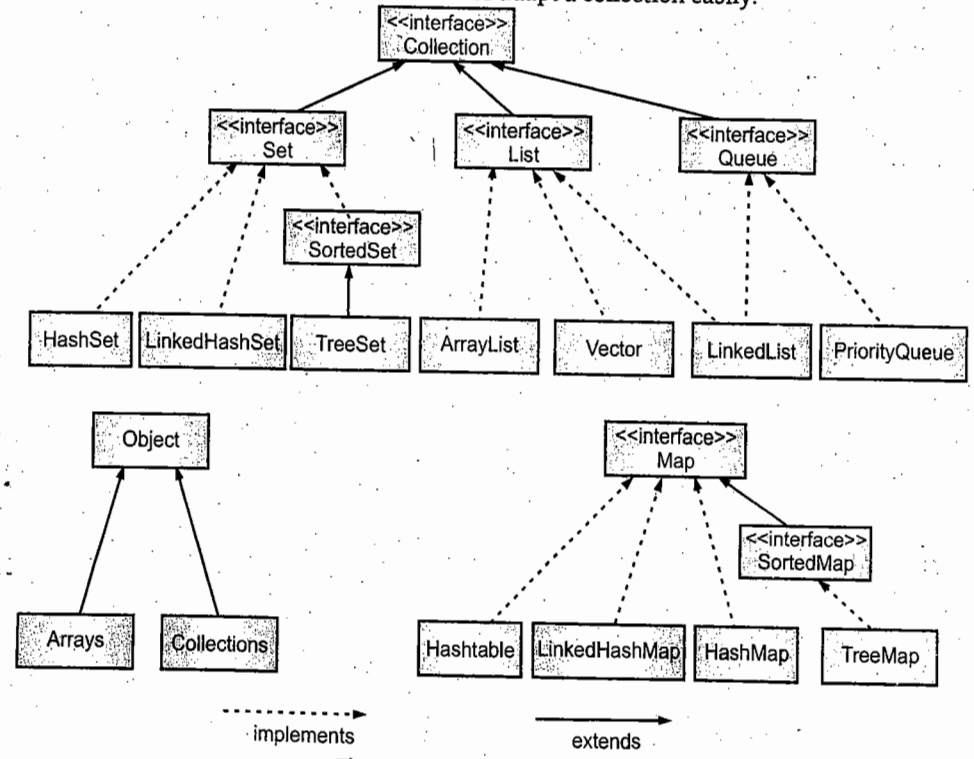


Fig. 11.1: Collection Framework

11.2 COLLECTION

- The group of data or the set of data which is binding in a unit in object form that unit is called Collection.
- Collection is an interface present at topmost position in collection framework.
- For the implementation of collection interface we can use any class which is at last level as leaf node in collection framework. Below few are mentioned:

```
Collection c=new HashSet();
Collection c=new ArrayList();
Collection c=new Vector();
```

Methods of collection:

Table 11.1: Methods of Collection

Sr. No.	Method and Description
1.	`boolean add(Object obj)` Adds obj to the invoking collection. Returns true if obj is added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
2.	`boolean addAll(Collection c)` Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.
3.	`void clear( )` Removes all elements from the invoking collection.
4.	`boolean contains(Object obj)` Returns true if obj is an element of the invoking collection. Otherwise, returns false.
5.	`boolean containsAll(Collection c)` Returns true if the invoking collection contains all elements of **c**. Otherwise, returns false.
6.	`boolean equals(Object obj)` Returns true if the invoking collection and obj are equal. Otherwise, returns false.
7.	`int hashCode( )` Returns the hash code for the invoking collection.
8.	`boolean isEmpty( )` Returns true if the invoking collection is empty. Otherwise, returns false.
9.	`Iterator iterator( )` Returns an iterator for the invoking collection.
10.	`boolean remove(Object obj)` Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.

contd. ...

11.	**boolean removeAll(Collection c)** Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
12.	**boolean retainAll(Collection c)** Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.
13.	**int size( )** Returns the number of elements held in the invoking collection.
14.	**Object[ ] toArray( )** Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
15.	**Object[ ] toArray(Object array[ ])** Returns an array containing only those collection elements whose type matches that of array.

Program 11.1: Write a Java Program to accept n city names from user and display it.

```
import java.util.*;
public class CollDemo
{
   public static void main(String args[])
   {
      int i,n;
      String cn;
      Collection c=new ArrayList();
      Scanner ob=new Scanner(System.in);
      System.out.println ("How Many");
      n=ob.nextInt ();
      for(i=1;i<=n;i++)
      {
         System.out.println ("City Name");
         cn=ob.next();
         c.add(cn);
      }
      System.out.println ("Entered Data Is " + c);
   }
}
```

Output:

```
How Many
3
City Name
pune
City Name
Kolhapur
City Name
satara
Entered Data Is [pune, kolhapur, satara]
```


Program 11.2: Write a java program to accept n integer form user and display only negative Integers on the screen.

```
import java.util.*;
public class CollDemo
{
   public static void main(String args[])
   {
      int i,n,m;
      Integer obj;
      Collection c=new ArrayList();
      Scanner ob=new Scanner (System.in);
      System.out.println ("How Many");
      n=ob.nextInt ();
      for(i=1;i<=n;i++)
      {
         System.out.println ("Number");
         m=ob.nextInt();
         c.add(m);
      }
      Iterator it=c.iterator ();
      while(it.hasNext())
      {
         obj=(Integer)it.next();
         if(obj<0)
         {
            System.out.println("Negative number : " +obj);
         }
      }
   }
}
```

Output:

```
How Many
4
Number
1
Number
-2
Number
3
Number
-4
Negative number : -2
Negative number : -4
```

Iterator:

- 'Iterator' is an interface which belongs to collection framework. It allows us to traverse the collection, access the data element and remove the data elements of the collection.
- java.util package has public interface Iterator and contains three methods:
  1. `boolean hasnext()`: It returns true if Iterator has more element to iterate.
  2. `Object next()`: It returns the next element in the collection until the hasNext() method return true. This method throws NoSuchElementException' if there is no next element.
  3. `void remove()`: It removes the current element in the collection. This method throws 'IllegalStateException' if this function is called before next() is invoked.

Program 11.3: Write a java program to accept n employee names through command line and display them.

```
import java.util.*;
public class IteratorDemo
{
    public static void main(String args[])
    {
        int i,n;
        String str;
        Integer obj;
        Collection c=new ArrayList ();
        n=args.length;
        for(i=0;i<n;i++)
        {
            c.add(args[i]);
        }
        Iterator it=c.iterator ();
        while(it.hasNext())
        {
            str=  (String)it.next();
            System.out.println (str);
        }
    }
}
```

Output:

```
C:\jdk1.8.0_221\bin>java IteratorDemo ACS BCS MCA
ACS
BCS
MCA
```


11.2.1 Collections of Objects

- Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue) and classes (ArrayList, Vector, LinkedList, Priority Queue, HashSet, LinkedHashSet, TreeSet).
- The Java language supports arrays to store several objects. An array is initialized with a predefined size during instantiation. To support more flexible data structures the core Java library provides the collection framework. A collection is a data structure which contains and processes a set of data. The data stored in the collection is encapsulated and the access to the data is only possible via predefined methods. For example the programmer can add elements to a collection via a method. Collections use internally arrays for their storage but hide the complexity of managing the dynamic size from the programmer.
- In java collection, data can be stored into an object form; object can be String, Integer or Float types.
- For storing the data into an object form we can use different types of Collection interfaces and classes such as List, Set, Collection, ArrayList, LinkedList, Hashtable, Vector etc.(repeat)

11.2.2 Collection Types

- A collection, as name implies, is group of objects. Java Collections framework is consist of the interfaces and classes which help in working with different types of collections such as List, Set, Map, Stack and Queue etc.
- These ready-to-use collection classes solve lots of very common problems where we need to deal with group of homogeneous as well as heterogeneous objects.
- The common operations in Collection involve: add, remove, update, sort, search and more complex algorithms. These collection classes provide very transparent support for all such operations using Collections APIs.

11.2.2.1 Sequence (List)

- A *Sequence* is a generalized array or list: Zero or more values treated as a compound value.
- List in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector.

Table 11.2: Methods of List

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.
int hashcode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean contains(Object o)	It returns true if the list contains the specified element
boolean containsAll(Collection<?> c)	It returns true if the list contains all the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

contd. ...

E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

Program 11.4: Write a java program to accept n elements from user, store them in LinkedList collection, search an element into the collection and display the message accordingly.

```
import java.util.*;
class ListDemo
{
    public static void main(String args[])
    {
        int i,n,m, data;
        Scanner ob=new Scanner(System.in);
        List l=new LinkedList();
        System.out.println ("How Many");
        n=ob.nextInt ();
        for(i=1;i<=n;i++)
        {
          System.out.println ("Enter the Number");
          m=ob.nextInt ();
          l.add(m);
        }
```

```
        System.out.println ("Enter the data for searching");
        data=ob.nextInt();
        boolean b=l.contains (data);
        if(b)
        {
          System.out.println ("Data Is Present");
        }
        else
        {
          System.out.println ("Not available");
        }
    }
}
```

Output:

```
How Many
3
Enter the Number
101
Enter the Number
103
Enter the Number
6
Enter the data for searching
103
Data Is Present
```

ListIterator:

- Iterator interface is used to traverse List or Set interface in forward direction only, ListIterator interface traverse List or Set interface in both the directions (backward and forward). ListIterator interface is inherited from Iterator interface.

Methods of ListIterator:

- **void add(E e):** Inserts the specified element into the list.
- **boolean hasNext():** Returns true if this ListItrator has more elements when traversing the list in the forward direction.
- **boolean hasPrevious():** Returns true if this ListIterator has more elements when traversing the list in the reverse direction.
- **E next():** Returns the next element in the list and advances the cursor position.
- **int nextIndex():** Returns the index of the element that would be returned by a subsequent call to next().
- **E previous ():** Returns the previous element in the list and moves the cursor position backwards.
- **int previousIndex():** Returns the index of the element that would be returned by a subsequent call to previous().

- **void remove():** Removes from the list the last element that was returned by next() or previous().
- **void set(E e):** Replaces the last element returned by next() or previous() with the specified element.

Program 11.5: Write a java program to accept N student names from user and display them in reverse order.

```
import java.util.*;
class ListDemo
{
    public static void main(String args[])
    {
        int i,n;
        String sn;
        Scanner ob=new Scanner (System.in);
        List l=new LinkedList();
        System.out.println ("How Many");
        n=ob.nextInt ();
        for(i=1;i<=n;i++)
        {
            System.out.println ("Student Name");
            sn=ob.next();
            l.add(sn);
        }
        ListIterator lst=l.listIterator (n);
        System.out.println("Student Names in Reverse Order");
        while(lst.hasPrevious())
        {
            sn=(String)lst.previous();
            System.out.println(sn);
        }
    }
}
```

Output:

```
How Many
4
Student Name
Advait
Student Name
Sagar
Student Name
Vishal
```

```
Student Name
Avinash
Student Names in Reverse Order
Avinash
Vishal
Sagar
Advait
```

11.2.2.2 Set

- A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- Unordered collection of elements are called Set interface. We can implement it in following ways:

```
Set s=new HashSet()
Set s=new TreeSet();
```

Table 11.3: Methods of Set

Sr. No.	Method and Description
1.	`add( )` Adds an object to the collection.
2.	`clear( )` Removes all objects from the collection.
3.	`contains( )` Returns true if a specified object is an element within the collection.
4.	`isEmpty( )` Returns true if the collection has no elements.
5.	`iterator( )` Returns an Iterator object for the collection, which may be used to retrieve an object.
6.	`remove( )` Removes a specified object from the collection.
7.	`size( )` Returns the number of elements in the collection.

Program 11.6: Write a java program to accept the data from a user, store it into the LinkedList Collection, removes duplicate data from the List and display it into the sorted order.

```
import java.util.*;
class SettDemo
{
   public static void main(String args[])
   {
```



```
      int i,n;
      String sn;
      Scanner ob=new Scanner (System.in);
      List l=new LinkedList ();
      System.out.println ("How Many");
      n=ob.nextInt ();
      for(i=1;i<=n;i++)
      {
         System.out.println ("Enter Data");
         sn=ob.next();
         l.add(sn);
      }
      System.out.println ("Entered Data " + l);
      Set s=new TreeSet(l);
      System.out.println ("Data After sorting " + s);
   }
}
```

Output:

```
How Many
3
Enter Data
DPU
Enter Data
DYP
Enter Data
ACS
Entered Data [DPU, DYP, ACS]
Data After sorting [ACS, DPU, DYP]
```

11.2.2.3 Map

- A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical function abstraction.
- The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap.
- A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.
- A Map is useful if you have to search, update or delete elements on the basis of a key.

Table 11.4: Methods of Map

Method	Description
`V put(Object key, Object value)`	It is used to insert an entry in the map.
`void putAll(Map map)`	It is used to insert the specified map in the map.
`V putIfAbsent(K key, V value)`	It inserts the specified value with the specified key in the map only if it is not already specified.
`V remove(Object key)`	It is used to delete an entry for the specified key.
`boolean remove(Object key, Object value)`	It removes the specified values with the associated specified keys from the map.
`Set keySet()`	It returns the Set view containing all the keys.
`Set<Map.Entry<K,V>> entrySet()`	It returns the Set view containing all the keys and values.
`void clear()`	It is used to reset the map.
`V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
`V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)`	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
`V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
`boolean containsValue(Object value)`	This method returns true if some value equal to the value exists within the map, else return false.
`boolean containsKey(Object key)`	This method returns true if some key equal to the key exists within the map, else return false.
`boolean equals(Object o)`	It is used to compare the specified Object with the Map.
`void forEach(BiConsumer<? super K,? super V> action)`	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
`V get(Object key)`	This method returns the object that contains the value associated with the key.

contd. ...

`V getOrDefault(Object key, V defaultValue)`	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
`int hashCode()`	It returns the hash code value for the Map
`boolean isEmpty()`	This method returns true if the map is empty; returns false if it contains at least one key.
`V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)`	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
`V replace(K key, V value)`	It replaces the specified value for a specified key.
`boolean replace(K key, V oldValue, V newValue)`	It replaces the old value with the new value for a specified key.
`void replaceAll(BiFunction<? super K,? super V,? extends V> function)`	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
`Collection values()`	It returns a collection view of the values contained in the map.
`int size()`	This method returns the number of entries in the map.

Program 11.7: Write a java program for frequency table. (It maps each word to the number of times it occurs in the argument list).

```
import java.util.*;
public class Freq
{
    public static void main(String[] args)
    {
        Map<String, Integer> m = new HashMap<String, Integer>();
        for (String a : args)
        {
            Integer freq = m.get (a);
            m.put (a, (freq == null)? 1 : freq + 1);
        }
        System.out.println (m.size () + "distinct words:");
        System.out.println (m);
    }
}
```

Input: DPU, DYP, CS, DYP

Output: DPU =1 DYP=2, CS=1

Program 11.8: Write a java program for the implementation of Map Interface.

```
import java.util.*;
public class MapExample1
{
   public static void main(String[] args)
   {
      Map map=new HashMap();
      map.put (1,"Satyavan");
      map.put (5,"Maithili");
      map.put (2,"Advait");
      map.put (6,"");
      Set set=map.entrySet();
      Iterator itr=set.iterator();
      while (itr.hasNext())
      {
         Map.Entry entry=(Map.Entry)itr.next();
         System.out.println (entry.getKey ()+" "+entry.getValue());
      }
   }
}
```

Output:

```
1 Satyavan
2 Advait
5 Maithili
6
```

11.3 UNDERSTANDING HASHING

- In **hashing** there is a hash function that maps keys to some values. But this hashing function may lead to collision that is two or more keys are mapped to same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
- Let us create a hash function, such that our hash table has 'N' number of buckets. To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.

  Example: `hashIndex = key % noOfBuckets`
- **Insert:** Move to the bucket corresponds to the above calculated hash index and insert the new node at the end of the list.
- **Delete:** To delete a node from hash table, calculate the hash index for the key, move to the bucket corresponds to the calculated hash index, search the list in the current bucket to find and remove the node with the given key (if found).

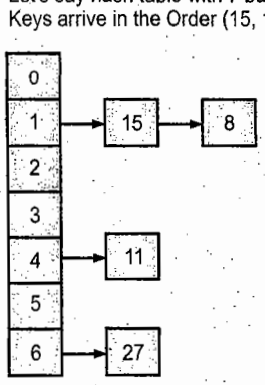


Fig. 11.2: Hashtable with key mapping

- For the implementation of hashing in java following classes are used.
  - HashSet
  - HashMap
  - Hashtable
- These are all generic type classes. The *HashSet*, for example, is derived from the *Set* interface that incorporates a hashing mechanism to the collection *Set*. This means that the implementation cannot contain duplicates and removes any duplicates in the *Collection* when it is constructed. Observe the output in the following example of how duplicate values in the *HashSet* are removed even though the array of *String* contains duplicate values.

Program 11.9: Write a java program for the implementation of HashSet

```
import java.util.*;
class HashSetDemo
{
   public static void main(String[] args)
   {
      String[] colors={"purple", "green", "red","magenta", "blue", "red",
      "purple"};
      Set set=new HashSet();
      for(int i=0;i<colors.length;i++)
      {
         set.add(colors[i]);
      }
      System.out.println(set);
   }
}
```

Output:

```
[red, magenta, green, blue, purple]
```

- The class *HashMap* is derived from the *Map* interface. The principle with the *Map* interface is that it stores as a key value pairs and cannot contain duplicate keys. That means each key can map only one associated value according to the one-to-one mapping function.
- The main difference between ***Map*** and ***Set*** is that *Map* maintains key value pairs that *Set* does not maintain. The ***Hashtable*** class also implements the *Map* interface. But, there is an important distinction between *Hashtable* and *HashMap*, though they are almost similar in functionality. ***HashMap*** is unsynchronized and allows null keys and null values. Due to its unsynchronized nature, multiple threads can modify it concurrently, but it must be synchronized explicitly. As per the Java API documentation, to prohibit unsynchronized access to this map, it is better to synchronize it at the time of creation.

```
Map<String, Integer> map=Collections.synchronizedMap
(new HashMap<>());
```

Program 11.10: Java Program for the implementation of HashMap class.

```
import java.util.HashMap;
public class MyClass
{
   public static void main(String[] args)
   {
      HashMap<String, String> cn = new HashMap<String, String>();
      cn.put("England", "London");
      cn.put("Germany", "Berlin");
      cn.put("Norway", "Oslo");
      cn.put ("USA", "Washington DC");
      System.out.println (cn);
   }
}
```

Output:

```
{USA=Washington DC, Norway=Oslo, England=London, Germany=Berlin}
```

Hashtable:

- It is an area in which information is stored in <key, value> pair for temporary purpose that area is called Hashtable. It is used for the implementation of Map Interface.

Program 11.11: Write a java program for the implementation of Hashtable class.

```
import java.util.*;
class HashDemo
{
   public static void main(String args[])
   {
      int i,n,sal;
```



```
      String en;
      Hashtable ht=new Hashtable ();
      Scanner ob=new Scanner (System.in);
      System.out.println ("How Many");
      n=ob.nextInt ();
      for(i=1;i<=n;i++)
      {
         System.out.println("Ename and Salary");
         en=ob.next();
         sal=ob.nextInt();
         ht.put(en,sal);
      }
      System.out.println("Entered Data " + ht);
   }
}
```

Output:

```
How Many
3
Ename and Salary
Sachin 89991
Ename and Salary
Yuvraj 67676
Ename and Salary
Rohit 87562
Entered Data {Sachin 89991, Yuvraj=67676, Rohit 87562}
```

11.4 USE OF ARRAYLIST AND VECTOR

11.4.1 ArrayList

- **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package.
- The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here. The ArrayList maintains the insertion order internally.
- ArrayList implements *List interface* so we can use all the methods of List interface for ArrayList class.

- The important points about ArrayList class are:
  - ArrayList class can contain duplicate elements.
  - ArrayList class maintains insertion order.
  - ArrayList class is non-synchronized.
  - ArrayList allows random access because array works at the index basis.
  - In ArrayList, manipulation is little bit slower than the LinkedList because a lot of shifting needs to occur if any element is removed from the array list.

Program 11.12: Write a Java program for the implementation of ArrayList class.

```
import java.util.*;
class ArrayListDemo
{
   public static void main (String args[])
   {
      int i,n;
      n=args.length;
      ArrayList l=new ArrayList ();
      for (i=0; i<n; i++)
      {
         l.add (args[i]);
      }
      System.out.println (l);
   }
}
```

Input: Uruli KunjirWadi Saswad Pimpri

Output: [Uruli, KunjirWadi, Saswad, Pimpri]

11.4.2 Vector

- **Vector** is a class belongs to package java.util used to store the data in object form. The generic dynamic array is called Vector.
- Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collections framework.
- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change size over the lifetime of a program.

Table 11.5: Constructor of Vector

Sr. No.	Constructor & Description
1.	`Vector( )` This constructor creates a default vector, which has an initial size of 10.
2.	`Vector(int size)` This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3.	`Vector(int size, int incr)` This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4.	`Vector(Collection c)` This constructor creates a vector that contains the elements of collection c.

Table 11.6: Methods of Vector

Sr. No.	Method and Description
1.	`void add(int index, Object element)` Inserts the specified element at the specified position in this Vector.
2.	`boolean add(Object o)` Appends the specified element to the end of this Vector.
3.	`boolean addAll(Collection c)` Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
4.	`boolean addAll(int index, Collection c)` Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5.	`void addElement(Object obj)` Adds the specified component to the end of this vector, increasing its size by one.
6.	`int capacity()` Returns the current capacity of this vector.
7.	`void clear()` Removes all of the elements from this vector.
8.	`Object clone()` Returns a clone of this vector.

contd. ...

9.	`boolean contains(Object elem)` Tests if the specified object is a component in this vector.
10.	`boolean containsAll(Collection c)` Returns true if this vector contains all of the elements in the specified Collection.
11.	`void copyInto(Object[] anArray)` Copies the components of this vector into the specified array.
12.	`Object elementAt(int index)` Returns the component at the specified index.
13.	`Enumeration elements()` Returns an enumeration of the components of this vector.

Program 11.13: Write a java program to display all the files having extension .java.

```
import java.util.*;
class VectDemo
{
   public static void main(String args[])
   {
      int i,n;
      n=args.length;
      Vector v=new Vector ();
      String str[]=new String[n];
      for(i=0;i<n;i++)
      {
         v.addElement (args[i]);
      }
      v.copyInto (str);
      for(i=0;i<n;i++)
      {
         if((str[i].endsWith(".java")))
         {
            System.out.println(str[i]);
         }
      }
   }
}
```

Output:

```
A.java
B.java
```

It will display all the files having .java extension from the given input.

11.5 JAVA UTILITIES (JAVA.UTIL PACKAGE)

- The java.util package contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes. This reference will take you through simple and practical methods available in java.util package.
- **Following are the Important Classes in Java.util package:**

1. **AbstractCollection:** This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.
2. **AbstractList:** This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
3. **AbstractMap<K,V>:** This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.
4. **AbstractMap.SimpleEntry<K,V>:** An Entry maintaining a key and a value.
5. **AbstractMap.SimpleImmutableEntry<K, V>:** An Entry maintaining an immutable key and value.
6. **AbstractQueue:** This class provides skeletal implementations of some Queue operations.
7. **AbstractSequentialList:** This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
8. **AbstractSet:** This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.
9. **ArrayDeque:** Resizable-array implementation of the Deque interface.
10. **ArrayList:** Resizable-array implementation of the List interface.
11. **Arrays:** This class contains various methods for manipulating arrays (such as sorting and searching).
12. **BitSet:** This class implements a vector of bits that grows as needed.
13. **Calendar:** The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.
14. **Collections:** This class consists exclusively of static methods that operate on or return collections.
15. **Currency:** Represents a currency.
16. **Date:** The class Date represents a specific instant in time, with millisecond precision.

17. **Dictionary<K,V>:** The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
18. **EnumMap,V>:** A specialized Map implementation for use with enum type keys.
19. **EnumSet:** A specialized Set implementation for use with enum types.
20. **EventListenerProxy:** An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.
21. **EventObject:** The root class from which all event state objects shall be derived.
22. **FormattableFlags:** FomattableFlags are passed to the Formattable.formatTo() method and modify the output format for Formattables.
23. **Formatter:** An interpreter for printf-style format strings.
24. **GregorianCalendar:** GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
25. **HashMap<K,V>:** Hash table based implementation of the Map interface.
26. **HashSet:** This class implements the Set interface, backed by a hash table (actually a HashMap instance).
27. **Hashtable<K,V>:** This class implements a hash table, which maps keys to values.
28. **IdentityHashMap<K,V>:** This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
29. **LinkedHashMap<K,V>:** Hash table and linked list implementation of the Map interface, with predictable iteration order.
30. **LinkedHashSet:** Hash table and linked list implementation of the Set interface, with predictable iteration order.
31. **LinkedList:** Doubly-linked list implementation of the List and Deque interfaces.
32. **ListResourceBundle:** ListResourceBundle is an abstract subclass of ResourceBundle that manages resources for a locale in a convenient and easy to use list.
33. **Locale – Set 1, Set 2:** A Locale object represents a specific geographical, political, or cultural region.
34. **Locale.Builder:** Builder is used to build instances of Locale from values configured by the setters.
35. **Objects:** This class consists of static utility methods for operating on objects.
36. **Observable:** This class represents an observable object, or "data" in the model-view paradigm.
37. **PriorityQueue:** An unbounded priority queue based on a priority heap.
38. **Properties:** The Properties class represents a persistent set of properties.
39. **PropertyPermission:** This class is for property permissions.

40. **PropertyResourceBundle:** PropertyResourceBundle is a concrete subclass of ResourceBundle that manages resources for a locale using a set of static strings from a property file.
41. **Random:** An instance of this class is used to generate a stream of pseudorandom numbers.
42. **ResourceBundle:** Resource bundles contain locale-specific objects.
43. **ResourceBundle.Control:** ResourceBundle.Control defines a set of callback methods that are invoked by the ResourceBundle.getBundle factory methods during the bundle loading process.
44. **Scanner:** A simple text scanner which can parse primitive types and strings using regular expressions.
45. **ServiceLoader:** A simple service-provider loading facility.
46. **SimpleTimeZone:** SimpleTimeZone is a concrete subclass of TimeZone that represents a time zone for use with a Gregorian calendar.
47. **Stack:** The Stack class represents a last-in-first-out (LIFO) stack of objects.
48. **StringTokenizer:** The string tokenizer class allows an application to break a string into tokens.
49. **Timer:** A facility for threads to schedule tasks for future execution in a background thread.
50. **TimerTask:** A task that can be scheduled for one-time or repeated execution by a Timer.
51. **TimeZone:** TimeZone represents a time zone offset, and also figures out daylight savings.
52. **TreeMap<K,V>:** A Red-Black tree based NavigableMap implementation.
53. **TreeSet:** A NavigableSet implementation based on a TreeMap.
54. **UUID:** A class that represents an immutable Universally Unique Identifier (UUID).
55. **Vector:** The Vector class implements a growable array of objects.
56. **WeakHashMap<K,V>:** Hash table based implementation of the Map interface, with weak keys.

Program 11.14: Write a java program to display date and time of a system.

```
import java.util.*;
class DateTimeDemo
{
   public static void main(String args[])
   {
      Date d=new Date();
      System.out.println("Date and Time of System is " + d);
   }
}
```

Output:

```
Date and Time of System is Sat Aug 29 14:29:59 PDT 2020
```

Summary

- Collection framework is a group of classes and interfaces. The List of classes are: ArrayList, LinkedList, HashSet, TreeSet. The List of interfaces are Collection, Set, List etc.
- Collection is used to bind group of data in object form. It is present at topmost position in collection framework.
- We can implement Collection framework in four ways:
  - Collection c=new ArrayList()
  - Collection c=new LinkedList()
  - Collection c=new HashSet()
  - Collection c=new TreeSet()
- List is an interface, inherited from the Collection interface used to store data in object form.
- Ordered collection of elements is called List Interface.
- In List interface we can store duplicate data.
- Unordered collection of elements is collectively called Set Interface.
- In Set interface we can store duplicate data. It is inherited from Collection interface. It also stores the data in sorted order.
- Iterator interface is used to traverse the data.(Traversing means to visit each and every element at least once)
- ListIterator interface is inherited from Iterator interface used to traverse the data in both the directions.
- Map interface is used to store the data in <key, value> pair. Its implementation can be done either by using HashMap or Hashtable class.
- Vector class is used to store the data in object form. The generic dynamic array is also called Vector.
- Hashtable class mostly used for the implementation of Map interface. A temporary storage area in which data is stored in <key, value> pair.
- Hashing is used to map the keys with values.

Check Your Understanding

1. Which of these classes is not part of Java's collection framework?
   (a) Maps (b) Array
   (c) Stack (d) Queue
2. What is Collection in Java?
   (a) A group of objects (b) A group of classes
   (c) A group of interfaces (d) None of the mentioned
3. Which of these return type of hasNext() method of an iterator?
   (a) Integer (b) Double
   (c) Boolean (d) Collections Object
4. Which of these iterators can be used only with List?
   (a) Setiterator (b) ListIterator
   (c) Literator (d) None of the mentioned
5. Which of these exceptions is thrown by remover() method?
   (a) IOException (b) SystemException
   (c) ObjectNotFoundExeception (d) IllegalStateException
6. Which of this collection class has the ability to grow dynamically?
   (a) Array (b) Arrays
   (c) ArrayList (d) None

7. The accuracy and efficiency of a HashMap can be guaranteed with_____.
   (a) override equals method (b) override hashCode method
   (c) None (d) (a) and (b)
8. Which is faster and uses less memory?
   (a) ListEnumeration (b) Iterator
   (c) Enumeration (d) ListIterator
9. Which is best suited to a multi-threaded environment?
   (a) WeakHashMap (b) Hashtable
   (c) HashMap (d) ConcurrentHashMap
10. Which provides better performance for the insertion and removal from the middle of the list?
    (a) Vector (b) ArrayList
    (c) LinkedList (d) All of these

Answers

1. (a)	2. (a)	3. (c)	4. (b)	5. (d)	6. (c)	7. (d)	8. (c)	9. (c)	10. (c)

Practice Questions

Q.I Answer the following Questions in short:

1. What is Collection? How to implement it.
2. What is hashing?
3. How to remove duplicate elements from a collection.
4. What is difference between ArrayList and LinkedList class?
5. What is use of ListIterator?
6. What is difference between HashMap and HashSet class?
7. Why does Vector is called Generic dynamic array?

Q.II Answer the following Questions:

1. Explain Collection framework in details.
2. Differentiate between List and Set interface.
3. Explain Hashtable with an example.
4. Differentiate between Vector and Array.
5. Differentiate between Iterator and ListIterator.
6. Write a java program for the implementation of Stack and Queue.
7. Explain Map interface with an example.
8. Explain different types of Collection in details.
9. Write a java program to accept the n employee names through command line, store them into the LinkedList and display them in reverse order.
10. Write a java program to accept n student names from user, store them into the Vector and display them in reverse order.

Q.III Write short note on:

1. Iterator
2. Random Class
3. TreeSet Class
4. Sequence
5. HashCode
6. HashMap

■■■

12... Database Programming Using JDBC

Objectives...

- To learn Java Database Connectivity (JDBC).
- To learn about JDBC Drivers and Architecture.
- To implement CURD operations using JDBC.
- To get knowledge about Connection to non-conventional databases.

12.1 INTRODUCTION TO JDBC

- Java Database Connectivity (JDBC) is an Application Programming Interface (API) used to connect Java application with Database.
- JDBC is used to interact with various types of Database such as Oracle, MS Access, My SQL and SQL Server.
- JDBC can also be defined as the platform-independent interface between a relational database and Java programming.
- It allows Java program to execute SQL statement and retrieve result from database.
- The JDBC API consists of classes and methods that are used to perform various operations like: connect, read, write, update, remove and store data in the database.
- It acts as a middle layer interface between Java applications and database.
- The JDBC classes are contained in the java package **java.sql** and **javax.sql.**
- JDBC helps you to write Java applications that manage three programming activities:
  - Connect to a data source, like a database.
  - Send queries and update statements to the database
  - Retrieve and process the results received from the database in answer to your query
- The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.
  - Making a connection to a database.
  - Creating SQL or MySQL statements.
  - Executing SQL or MySQL queries in the database.
  - Viewing & Modifying the resulting records

12.2 JDBC DRIVERS AND ARCHITECTURE

12.2.1 JDBC Architecture

- JDBC supports two types of processing models for accessing database i.e. two-tier and three-tier.

1. Two-tier Architecture:

- This architecture helps Java program or application to directly communicate with the database.
- It needs a JDBC driver to communicate with a specific database. Query or request is sent by the user to the database and results are received back by the user.
- The database may be present on the same machine or any remote machine connected via a network. This approach is called Client-Server architecture or configuration.

2. Three-tier Architecture:

- In three-tier, there is no direct communication. Requests are sent to the middle tier i.e. HTML browser sends a request to Java application which is then further sent to the database.
- Database processes the request and sends the result back to the middle tier which then communicates with the user. It increases the performance and simplifies the application deployment.

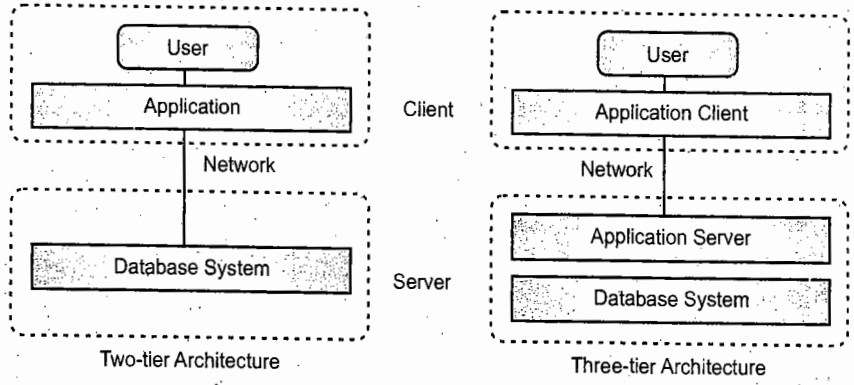


Fig. 12.1: Two-tier and Three-tier Architecture

JDBC architecture consists of two layers:

1. JDBC Application Layer:

- JDBC Application Layer consists of two components - Java application and JDBC API.
- In Application layer, a Java Application that wishes to communicate with a database uses the JDBC API to access the required JDBC driver.
- The JDBC API contains the JDBC driver manager, which connects Java application with the JDBC driver.

2. JDBC Driver Layer:

- JDBC Driver Layer consists of several database drivers that may be required to connect a Java application to any specific database as per user's requirement.
- For example - the JDBC Driver Layer may contain a MySQL database driver, an Oracle driver, a PostgreSQL Database driver and MariaDB Database driver to connect any Java application with any of these databases as per the requirement of this Java application.
- This JDBC driver is nothing but a program through which a Java application can easily communicate with a database.
- This JDBC driver translates the requests of a Java application such a database query like database access, update or storing procedures from Java language to Structured Query Language (SQL), which is then forwarded to a specific database which is used in Java application.
- These SQL statements are executed by the preferred database and the result is sent back to JDBC driver. The JDBC driver sends the result back to Java application (through JDBC API) in the form which it can easily understand.

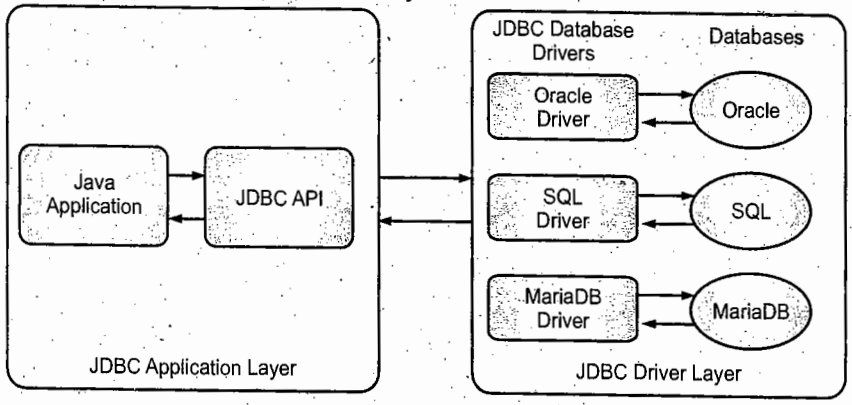


Fig. 12.2: JDBC Application and Driver Layer

12.2.2 JDBC Driver Type

- JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand.
- There are 4 types of JDBC drivers:
  1. Type-1 driver or JDBC-ODBC bridge driver.
  2. Type-2 driver or Native-API driver.
  3. Type-3 driver or Network Protocol driver.
  4. Type-4 driver or Thin driver.

1. Type-1 Driver or JDBC-ODBC Bridge:

- This driver acts as a bridge between JDBC and ODBC. It converts JDBC calls into ODBC calls and then sends the request to ODBC driver. It is easy to use but execution time is slow.

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver is not written in Java, that's why it is not a portable driver.
- This driver software is built-in with JDK so no need to install separately.
- It is a database independent driver.

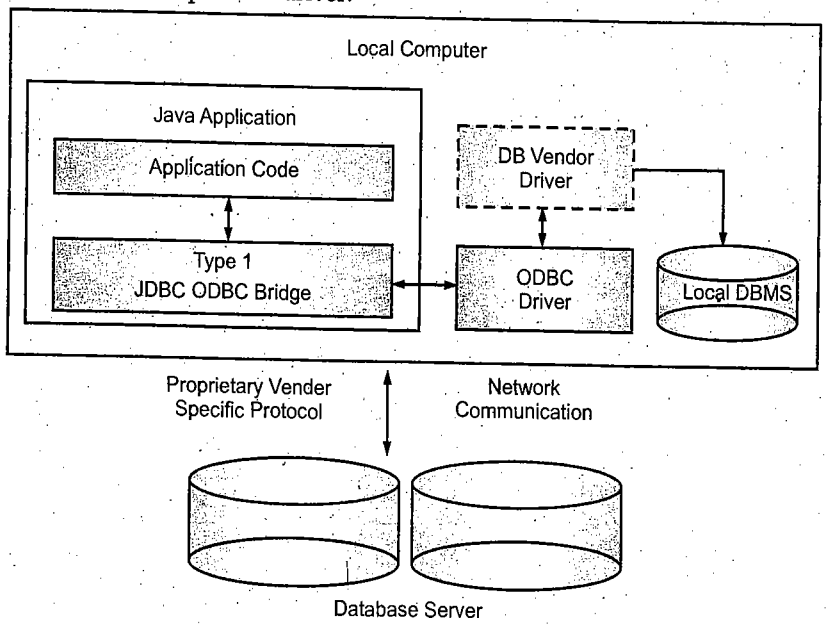


Fig. 12.3: Type-1 Driver or JDBC-ODBC Bridge

Advantages:

1. Easy to use.
2. Can be easily connected to any database.

Disadvantages:

1. Performance degraded because JDBC method call is converted into the ODBC function calls.
2. The ODBC driver needs to be installed on the client machine.

2. Type-2 Driver or Native API Partly Java Driver:

- This driver uses JNI (Java Native Interface) call on database specific native client API.
- It is comparatively faster than Type-1 driver but it requires native library and cost of application also increases.
- Driver needs to be installed separately in individual client machines.
- The Vendor client library needs to be installed on client machine.
- Type-2 driver is not written in Java, that's why it is not a portable driver
- It is a database dependent driver.

Advantage:

1. Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

1. The Native driver needs to be installed on the each client machine.
2. The Vendor client library needs to be installed on client machine.

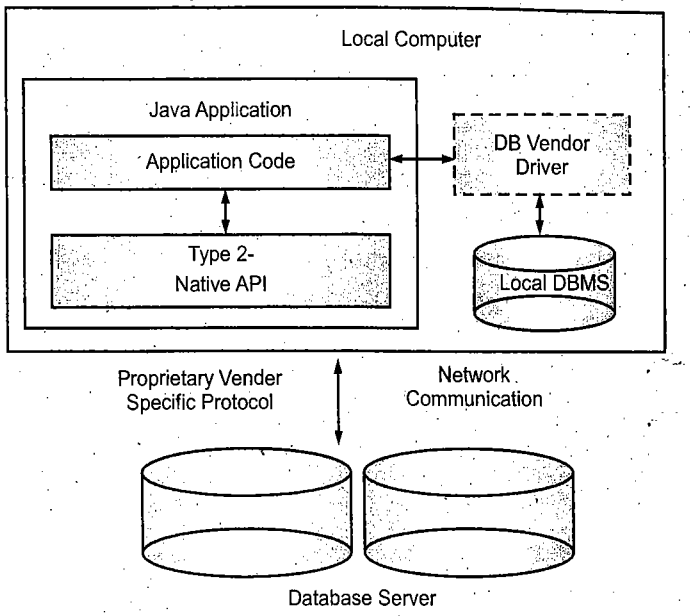


Fig. 12.4: Type-2 Driver or Native API Partly Java Driver

3. Type-3 Driver or Network Protocol Driver:

- These drivers communicate to JDBC middleware server using proprietary network protocol.
- This middleware translates the network protocol to database specific calls. They are database independent.
- They can switch from one database to another but are slow due to many network calls.
- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.
- Switching can be done from one database to another database.

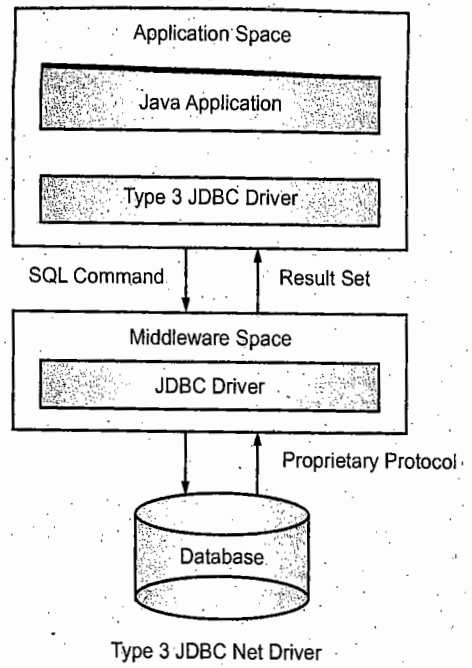


Fig. 12.5: Type-3 Driver or Network Protocol Driver

Advantage:

1. No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

1. Network support is required on client machine.
2. Requires database-specific coding to be done in the middle tier.
3. Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4. Type-4 or Thin Driver:

- This driver is also called pure Java driver because they directly interact with the database.
- It neither requires any native library nor middleware server.
- It has better performance than other drivers but comparatively slow due to an increase in a number of network calls.
- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.

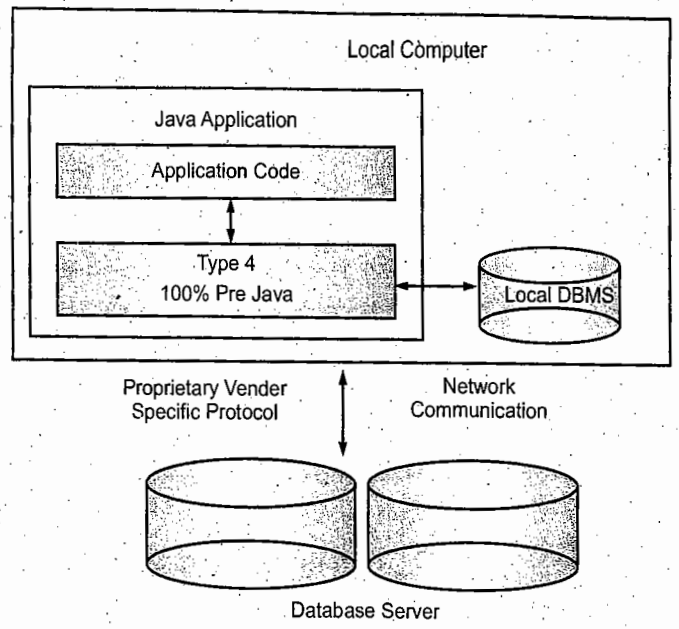


Fig. 12.6: Type-4 or Thin Driver

Advantages:

1. Better performance than all other drivers.
2. No software is required at client side or server side.

Disadvantage:

1. Drivers depend on the Database.

Selecting suitable Drivers for Database Connection:

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
- If your Java application is accessing multiple types of databases at the same time, type-3 is the preferred driver.
- Type-2 drivers are useful in situations, where a type-3 or type-4 driver is not available yet for your database.
- The type-1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

12.2.3 JDBC Components and Architecture

- The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers:
  - **JDBC API:** This provides the application-to-JDBC Manager connection.
  - **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.
- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

- The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.
- Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application –

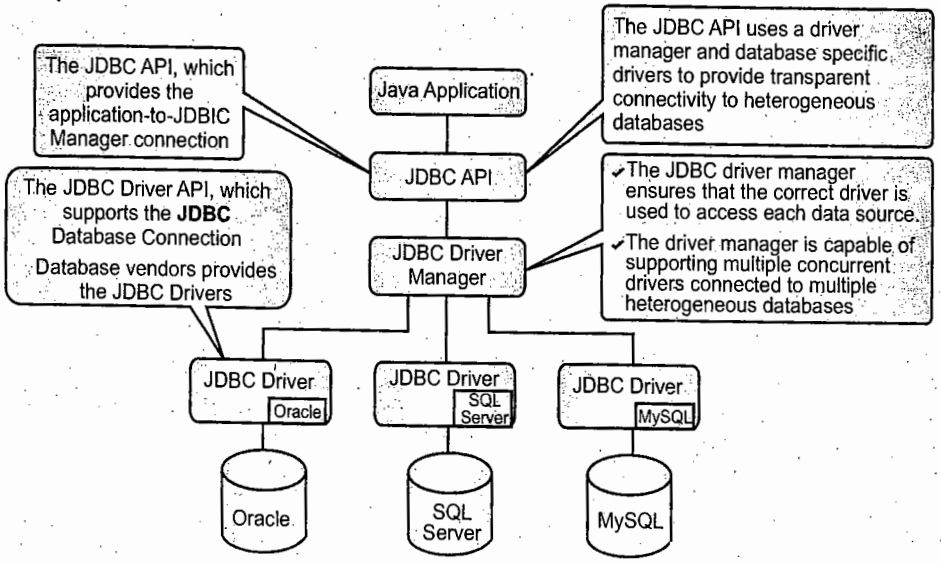


Fig. 12.7: JDBC Component Architecture

Common JDBC Components:

1. DriverManager Class:

- This class manages a list of database drivers. It matches the connection requests from the Java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain sub protocol under JDBC will be used to establish a database Connection.
- The DriverManager class acts as an interface between user and drivers.
- It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
- The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method `DriverManager.registerDriver()`.
- **Syntax:**
```
Connection cn= DriverManager.get
Connection  (ConnectionString,userID,Password);
```
- **Example:**
```
Connection cn=DriverManager.getConnection
("jdbc:postgresql:Demo","postgres","");
```


Methods used in DriverManager class:

Table 12.1: DriverManager Class Methods

Methods	Description
`public static void registerDriver(Driver driver)`	It is used to register the given driver with DriverManager.
`public static void deregisterDriver(Driver driver)`	It is used to deregister the given driver (drop the driver from the list) with DriverManager.
`public static Connection getConnection(String url)`	It is used to establish the connection with the specified url.
`public static Connection getConnection(String url,String userName,String password)`	It is used to establish the connection with the specified url, username and password.

2. Connection Interface:

- A Connection is the session between Java application and database.
- This interface consist various methods for contacting a database. The connection object represents communication context, i.e., all communication with database is done through connection object only.
- The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData.
- The connection interface provide many methods for transaction management like commit(), rollback() etc.
- By default, connection commits the changes after executing queries.
- **Syntax:** `Connection con;  //creating a Connection object`

Methods used in Connection Interface:

Table 12.2: Methods of Connection Interface

Methods	Description
`public Statement createStatement()`	Creates a statement object that can be used to execute SQL queries.
`public Statement createStatement(int resultSetType,int resultSetConcurrency)`	Creates a Statement object that will generate ResultSet objects with the given type and concurrency.
`public void setAutoCommit(boolean status)`	Is used to set the commit status. By default it is true.
`public void commit()`	Saves the changes made since the previous commit/rollback permanent.
`public void rollback()`	Drops all changes made since the previous commit/rollback.
`public void close()`	Closes the connection and Releases a JDBC resources immediately.

2. Statement Interface:

- This interface creates an object which is used to submit the SQL statements to the database. Some Statement interfaces accept parameters in addition to executing stored procedures.
- There are three types of Statements:
  (a) Statement
  (b) Prepared Statement
  (c) Callable Statement

(a) Statement Interface:

- The Statement interface provides methods to execute queries with the database.
- The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.
- **Syntax:** `Statement stmt = con.createStatement(); //creating Statement object`
- The important methods of Statement interface are as follows:

Table 12.3: Methods of Statement Interface

Methods	Description
`public ResultSet executeQuery(String sql)`	It is used to execute SELECT query. It returns the object of ResultSet.
`public int executeUpdate(String sql)`	It is used to execute specified query, it may be create, drop, insert, update, delete etc.
`public boolean execute(String sql)`	It is used to execute queries that may return multiple results.
`public int[] executeBatch()`	It is used to execute batch of commands.

(b) PreparedStatement:

- The PreparedStatement interface is a sub interface of Statement.
- It is used to execute parameterized precompiled query.
- Let's see the example of parameterized query:

```
String sql="insert into emp values(?,?,?)";
```

- Here, we are passing parameter (?) as a placeholder for the values. Its value will be set by calling the setter methods of PreparedStatement.
- The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.
- The prepareStatement() method of Connection interface is used to return the object of PreparedStatement.

- **Syntax:**

```
public PreparedStatement prepareStatement(String query)throws
SQLException{}
```

- The important methods of PreparedStatement interface are given below:

Table 12.4: Methods of PreparedStatement

Methods	Description
`public void setInt(int paramIndex, int value)`	Sets the integer value to the given parameter index.
`public void setString(int paramIndex, String value)`	Sets the String value to the given parameter index.
`public void setFloat(int paramIndex, float value)`	Sets the float value to the given parameter index.
`public void setDouble(int paramIndex, double value)`	Sets the double value to the given parameter index.
`public int executeUpdate()`	Executes the query. It is used for create, drop, insert, update, delete etc.
`public ResultSet executeQuery()`	Executes the select query. It returns an instance of ResultSet.

(c) CallableStatement Interface:

- CallableStatement interface is used to call the stored procedures and functions.
- For Example: Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.
- The prepareCall () method of Connection interface returns the instance of CallableStatement.
- **Syntax:**

```
public CallableStatement prepareCall("{ call procedurename(?,?...?)}");
```

- **Syntax:**

```
CallableStatement stmt=con.prepareCall("{call Disp(?,?)}");
```

- Here, It calls the procedure *Disp* that receives 2 arguments.

Table 12.5: Difference between Statement, PreparedStatement and CallableStatement In Java

Statement	PreparedStatement	CallableStatement
It is used to execute normal SQL queries.	It is used to execute parameterized or dynamic SQL queries.	It is used to call the stored procedures.

contd. ...

It is preferred when a particular SQL query is to be executed only once.	It is preferred when a particular query is to be executed multiple times.	It is preferred when the stored procedures are to be executed.
You cannot pass the parameters to SQL query using this interface.	You can pass the parameters to SQL query at run time using this interface.	You can pass 3 types of parameters using this interface. They are – IN, OUT and IN OUT.
This interface is mainly used for DDL statements like CREATE, ALTER, DROP etc.	It is used for any kind of SQL queries which are to be executed multiple times.	It is used to execute stored procedures and functions.
The performance of this interface is very low.	The performance of this interface is better than the Statement interface (when used for multiple execution of same query).	The performance of this interface is high.

3. ResultSet Interface:

- The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.
- ResultSet object can be moved forward only and it is not updatable.
- But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable.
- There are three Types of ResultSet:

Table 12.6: Types of ResultSet

Types	Description
`ResultSet.TYPE_FORWARD_ONLY (Default)`	The cursor can only move forward in the result set.
`ResultSet.TYPE_SCROLL_INSENSITIVE`	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
`ResultSet.TYPE_SCROLL_SENSITIVE.`	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

- **For Example**

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

- **Syntax:** Using Statement Interface,

```
Statement statement = connection.createStatement();
ResultSet result = statement.executeQuery("select * from people");
Using  PreparedStatement Interface
String sql = "select * from people";
PreparedStatement statement = connection.prepareStatement(sql);
ResultSet result = statement.executeQuery();
```

Methods used in ResultSet interface:

Table 12.7: ResultSet Methods

Methods	Description
`public boolean next()`	It is used to move the cursor to the one row next from the current position.
`public boolean previous()`	It is used to move the cursor to the one row previous from the current position.
`public boolean first()`	It is used to move the cursor to the first row in result set object.
`public boolean last()`	It is used to move the cursor to the last row in result set object.
`public boolean absolute(int row)`	It is used to move the cursor to the specified row number in the ResultSet object.
`public boolean relative(int row)`	It is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
`public int getInt(int columnIndex)`	It is used to return the data of specified column index of the current row as int.
`public int getInt(String columnName)`	It is used to return the data of specified column name of the current row as int.
`public String getString(int columnIndex)`	It is used to return the data of specified column index of the current row as String.
`public String getString(String columnName)`	It is used to return the data of specified column name of the current row as String.

4. Closing the resources:

- Once all the data is retrieved and all the operations are completed. We must close all the resources.

- For Example:

```
Stmt.close();  //Statement closed
Rs.close();    //ResultSet close
Con.close();   //Connection closed
```

5. DatabaseMetaData Interface:

- DatabaseMetaData interface provides methods to get metadata (data about data) of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.
- The getMetaData() method of Connection interface returns the object of DatabaseMetaData.
- **Syntax:** public DatabaseMetaData getMetaData()throws SQLException
- Commonly used methods of DatabaseMetaData interface.

Table 12.8: DatabaseMetaData Methods

Methods	Description
`public String getDriverName()throws SQLException`	It returns the name of the JDBC driver.
`public String getDriverVersion()throws SQLException`	It returns the version number of the JDBC driver.
`public String getUserName()throws SQLException`	It returns the username of the database.
`public String getDatabaseProductName()throws SQLException`	It returns the product name of the database.
`public String getDatabaseProductVersion()throws SQLException`	It returns the product version of the database.
`public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException`	It returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

6. ResultSetMetaData Interface:

- The metadata means data about data i.e. we can get further information from the data.
- If you have to get metadata of a table like total number of column, column name, column type etc.
- ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.
- The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData.

- **Syntax:**

```
public ResultSetMetaData getMetaData()throws SQLException
```

- Commonly used methods of ResultSetMetaData interface.

Table 12.9: ResultSetMetaData Methods

Methods	Description
`public int getColumnCount()throws SQLException`	It returns the total number of columns in the ResultSet object.
`public String getColumnName (int index)throws SQLException`	It returns the column name of the specified column index.
`public String getColumnTypeName (int index)throws SQLException`	It returns the column type name for the specified index.
`public String getTableName (int index)throws SQLException`	It returns the table name for the specified column index.

12.3 CONNECTING TO NON-CONVENTIONAL DATABASES

- JDBC is a technique used to connect Java application with database or DSN.
- The Steps for connecting JDBC application with database:
  1. Import JDBC-API
  2. Establishing a connection.
  3. Create a statement.
  4. Execute the query.
  5. Process the ResultSet object.
  6. Close the connection.

1. Import JDBC-API:

- For writing JDBC application, JDBC-API has to import because it consists of list of classes and interfaces which are needed for executing JDBC application.

2. Establishing Connections:

- First, establish a connection with the data source you want to use.
- A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver.
- This connection is represented by a Connection object.
- For Example:

```
//Load Driver into memory using Class.forName()
Class.forName("Driver Name");
Connection cn=DriverManager.getConnection("url", "userID", "Password");
```

Table 12.10: List of Database, driver and URL with example

Relational Database	Driver Name (qualified class name)	Database URL and Example
MySQL	com.mysql.jdbc.Driver org.gjt.mm.mysql.Driver	jdbc:mysql://<server>:<port>/<databaseName> **Example:** jdbc:mysql://localhost:3306/myDBName
Oracle	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@<server>:<port>:<databaseName> **Example:** jdbc:oracle:thin:@localhost:1521:xe
IBM DB2 App	com.ibm.db2.jdbc.app.DB2Driver	jdbc:db2:<databaseName> **Example:** jdbc:db2:myDBName
IBM DB2 Net	com.ibm.db2.jdbc.net.DB2Driver	jdbc:db2//<server>:<port>/<databasebName> **Example:** jdbc:db2://localhost:6789/myDBName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:<server>:<port>/<databaseName> **Example:** jdbc:sybase:Tds:localhost:4100/myDBName
Teradata	com.teradata.jdbc.TeraDriver	jdbc:teradata://<server>/database=<databaseName>, tmode=ANSI,charset=UTF8 **Example:** jdbc:teradata://localhost/database=myDBName, tmode=ANSI, charset=UTF8
Microsoft SQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:sqlserver://<server>:<port>;databaseName=<databaseName> **Example:** jdbc:sqlserver://localhost:1433;databaseName=myDBName

contd. ...

Postgres	org.postgresql.Driver	jdbc:postgresql://<server>:<port>/<databaseName> **Example:** jdbc:postgresql://localhost:5432/myDBName
MS Access (JDBC-ODBC Bridge)	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)}; DBQ=<myDBName.mdb>; **Example:** jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=myDBName.mdb;

3. Creating Statements:

- A Statement is an interface that represents a SQL statement.
- You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set.
- You need a Connection object to create a Statement object.
- For example, Creates a Statement object with the following code:
- `Statement stmt = con.createStatement();`
- There are three different kinds of statements:
  - **Statement:** Used to implement simple SQL statements with no parameters.
  - **PreparedStatement:** (Extends Statement.) Used for precompiling SQL statements that might contain input parameters. See Using PreparedStatements for more information
  - **CallableStatement:** (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters.

4. Executing Queries:

- To execute a query, call an execute method from Statement interface. They are as follows
  - **execute():** Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet().
  - **executeQuery():** Returns one ResultSet object.
  - **executeUpdate():** Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.
- For example, ResultSet `rs = stmt.executeQuery(query);`

5. Processing ResultSet Objects:

- You access the data in a ResultSet object through a cursor.
- Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet object.
- Initially, the cursor is positioned before the first row. You call various methods defined in the ResultSet object to move the cursor.
- ResultSet uses next() method to retrieve the result i.e. ResultSet.next();
- For example,

```
try
{
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select * from emp");
    while (rs.next())
    {
        String en = rs.getString("EName");
        int eno = rs.getInt("Emp_ID");
        float Sal = rs.getFloat("Salary");
        System.out.println(eno + "\t" + en +"\t" + sal)
    }
}
```

6. Closing Connections:

- When you are finished using a Statement, call the method Statement.close() to immediately release the resources it is using. When you call this method, its ResultSet objects are closed.
- Once all the operations are performed on JDBC then close the connection object using:

```
cn.close()
```

- We can directly use close() method to close the resource or can use finally block to close the resources:

```
finally
{
if (stmt != null) { stmt.close(); }
}
cn.close(); // Connection close
```

12.4 CURD OPERATION USING JDBC

- CURD is an acronym for CREATE, UPDATE, READ and DELETE which are basic functions of persistent storage.
- CURD operations can use forms or an interface view to retrieve and return data from a database.

Table 12.11: CURD Operation

Procedures	Function
CREATE	This is a form view to create a new record in the database.
UPDATE	Updates the content of the table based on the specified primary key for a record.
READ	Reads the table records based on the primary key within the input parameter.
DELETE	Deletes a specified row in the table.

- **For performing operations on Oracle database need to set following settings:**

1. Creating a user in Oracle Database and granting required permissions:
   Create a user-id protected by a password. This user-id is called child user.

```
create user USERNAME identified by PASSWORD;
```

2. Grant required permissions to child user. For simplicity we grant database administrator privilege to child user.

```
GRANT CONNECT, RESOURCE, DBA TO user_name;
connect username/password;
```

3. Once you are connected with oracle database then create a table.
   Create a sample table with blank fields:

```
CREATE TABLE userid
{
    id varchar2(30) NOT NULL PRIMARY KEY,
    pwd varchar2(30) NOT NULL,
    fullname varchar2(50),
    email varcha  r2(50)
);
```

Note: In the following code database name is login1 and password pwd1.

Program 12.1: Java program to illustrate Connecting to the Database.

```
import java.sql.*;
public class connect
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            // Establishing Connection
            Connection con = DriverManager.getConnection(
```

```
            "jdbc:oracle:thin:@localhost:1521:orcl", "login1", "pwd1");
            if (con != null)
                System.out.println("Database Connected");
            else
                System.out.println("Not Connected");
            con.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output: Database Connected

4. Implementing INSERT Statement:

- Once the database is connected with JDBC application the next step is using INSERT operation
- Using INSERT we can read the values and store it in to the table associated with database

Program 12.2: Java program to illustrate inserting to the Database.

```
import java.sql.*;
public class insert1
{
    public static void main(String args[])
    {
        String id = "1";
        String pwd = "dpu2020";
        String fullname = "DYP";
        String email = "acs.dypvp.edu.in";
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection("
            jdbc:oracle:thin:@localhost:1521:orcl","login1", "pwd1");
            Statement stmt = con.createStatement();
            // Inserting data in database
            String q1 = "insert into userid values('" +id+ "', '" +pwd+
                        "', '" +fullname+ "', '" +email+ "')";
            int x = stmt.executeUpdate(q1);
            if (x > 0)
                System.out.println("Record Successfully Inserted");
```



```
            else
                System.out.println("Insert Failed");
            con.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output: Record Successfully Inserted

5. Implementing UPDATE Statement

- We can update the records i.e. changing the existing values into new values with in database.
- For this we use UPDATE command

Program 12.3: Java program to illustrate updating the Database.

```
import java.sql.*;
public class update1
{
    public static void main(String args[])
    {
        String id = "id1";
        String pwd = "pwd1";
        String newPwd = "newpwd";
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection("
             jdbc:oracle:thin:@localhost:1521:orcl", "login1", "pwd1");
            Statement stmt = con.createStatement();
            // Updating database
            String q1 = "UPDATE userid set pwd = '" + newPwd +
                    "' WHERE id = '" +id+ "' AND pwd = '" + pwd + "'";
            int x = stmt.executeUpdate(q1);
            if (x > 0)
                System.out.println("Password Successfully Updated");
            else
                System.out.println("ERROR OCCURED:(");
            con.close();
        }
```

```
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
```

Output: Password Successfully Updated

6. Implementing DELETE Statement

- If you want to delete any specific record then that can be achieved using DELETE statement.

Program 12.4: Java program to illustrate deleting from Database.

```
import java.sql.*;
public class delete
{
    public static void main(String args[])
    {
        String id = "id2";
        String pwd = "pwd2";
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection("
            jdbc:oracle:thin:@localhost:1521:orcl", "login1", "pwd1");
            Statement stmt = con.createStatement();
            // Deleting from database
            String q1 = "DELETE from userid WHERE id = '" + id +
                    "' AND pwd = '" + pwd + "'";
            int x = stmt.executeUpdate(q1);
            if (x > 0)
                System.out.println("One User Successfully Deleted");
            else
                System.out.println("ERROR OCCURED:(");
            con.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output:

```
One User Successfully Deleted
```


7. Implementing SELECT Statement

- The records can be displayed/selected as per users need.
- This can be achieved using SELECT keyword

Program 12.5: Java program to illustrate selecting from Database.

```
import java.sql.*;
public class select
{
    public static void main(String args[])
    {
        String id = "id1";
        String pwd = "pwd1";
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection("
            jdbc:oracle:thin:@localhost:1521:orcl", "login1", "pwd1");
            Statement stmt = con.createStatement();
            // SELECT query
            String q1 = "select * from userid WHERE id = '" + id +
                                "' AND pwd = '" + pwd + "'";
            ResultSet rs = stmt.executeQuery(q1);
            if (rs.next())
            {
                System.out.println("User-Id: " + rs.getString(1));
                System.out.println("Full Name:" + rs.getString(3));
                System.out.println("E-mail:" + rs.getString(4));
            }
            else
            {
                System.out.println("No such user id is already
                registered");
            }
            con.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output:

```
User-Id:UI001
Full Name: Malati Tribhuwan
E-mail:tmv@gmail.com
```

PROGRAMS

Program 12.6: Write a Java Program for the implementation DML commands using Statement interface.

```
import java.sql.*;
class FetchRecord
{
   public static void main(String args[])throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection  con = DriverManager.getConnection
      ("jdbc:oracle:thin:@localhost:1521:xe", "system","oracle");
      Statement stmt=con.createStatement();
      stmt.executeUpdate("insert into emp values(1,'Malati',50000)");
      System.out.println("one record inserted…!");
      int  result=stmt.executeUpdate("update  emp  set  name='Satyavan',
      salary=10000 where id=1");
      int result=stmt.executeUpdate("delete from emp where id=1");
      System.out.println(result+" records affected");
      con.close();
   }
}
```

Output:

After insert command

id	Name	Salary
1	Malati	50000

one record inserted…!

After Update Command

id	Name	salary
1	Satyavan	10000

1 records affected

After Delete Command

1 records affected

Program 12.7: Consider Emp (Eno, Ename, sal) table. Write a Java program to insert the details of employee into the table using PreparedStatement interface .

```
import java.sql.*;
class InsertPrepared
{
   public static void main(String args[])
   {
```



```
      try
      {
         Class.forName ("oracle.jdbc.driver.OracleDriver");
         Connection con= DriverManager.getConnection
        ("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
         PreparedStatement  stmt=con.prepareStatement("insert  into  Emp
         values(?,?,?)");
         stmt.setInt(1,101);//1 specifies the first parameter in the query
         stmt.setString(2,"Advait");
         stmt.setInt(3,89899);
         int i=stmt.executeUpdate();
         System.out.println(i+" Record is Inserted");
         con.close();
      }catch(Exception e){ System.out.println(e);}
   }
}
```

Output:

1 Record Is Inserted

Program 12.8: Write a menu driven application in Java for the following.

(a) Insert
(b) Update
(c) Delete
(d) Search
(e) Display.
(f) Exit

Assume Emp (ENo, EName, Salary) is already created.

```
import java.sql.*;
import java.io.*;
class CRDImp
{
   public static void main(String args[])
   {
      int eno, sal,ch,k;
      String sql,en;
      try
      {
         Class.forName ("oracle.jdbc.driver.OracleDriver");
         Connection  con= DriverManager.getConnection
                ("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
         Statement st=con.createStatement ();
         BufferedReader br=new
                        BufferedReader(newInputStreamReader(System.in));
```

```
do
{
   System.out.println("1. Insert");
   System.out.println("1. Insert");
   System.out.println("2. Update");
   System.out.println("3. Delete");
   System.out.println("4. Search");
   System.out.println("5. Display");
   System.out.println("6. Exit");
   System.out.println("Enter Ur Choice");
   ch=Integer.parseInt(br.readLine());
   switch(ch)
   {
      case 1:
      System.out.println("ENo, EName Salary");
      eno=Integer.parseInt(br.readLine());
      en=br.readLine();
      sal=Integer.parseInt(br.readLine());
      sql="insert into Emp values(" + eno + " , '" + en
                                          + "',"+ sal + ")";
      k=st.executeUpdate(sql);
      if(k>0)
      System.out.println("Record Is Inserted….!");
      break;
      case 2:
      System.out.println("ENo, Salary");
      eno=Integer.parseInt(br.readLine());
      sal=Integer.parseInt(br.readLine());
      sql="update Emp set salary=" + sal + " where Eno=" + eno;
      k=st.executeUpdate(sql);
      if(k>0)
      System.out.println("Record Is Updated….!");
      break;
      case 3:
      System.out.println("ENo");
      eno=Integer.parseInt(br.readLine());
      sql="delete from Emp where Eno=" + eno;
      k=st.executeUpdate(sql);
      if(k>0)
      System.out.println("Record Is Deleted….!");
      break;
      case 4:
      System.out.println("Enter Eno for Searching");
      Eno=Integer.parseInt(br.readLine());
      ResultSet rs=st.executeQuery
      ("select * from Emp where Eno=" + eno);
      rs.next();
      System.out.println(rs.getString(1)
      + " " + rs.getString(2) + " " + rs.getString(3));
      break;
```



```
      case 5:
      System.out.println("Data From Emp Table Is");
      ResultSet rs1=st.executeQuery
      ("select * from Emp where Eno=" + eno);
      while(rs1.next())
      {
      System.out.println(rs1.getString(1)
      + " " + rs1.getString(2) + " " + rs1.getString(3));
      }
      break;
      case 6:
      cn.close();
      System.exit(0);
   }
  }
  while(ch!=6);
 }
 catch(Exception e)
 {
   System.out.println(e);
 }
 }
}
```

Program 12.9: Program Demonstrating CallableStatement using Stored Procedure.

Step 1: Create a stored procedure Accept()

```
create or replace procedure "Accept"  (id IN NUMBER,  name IN VARCHAR2) is
begin insert into Emp values(id,name);
end;
/
```

Step 2: Write a JDBC program to call the stored procedure

```
import java.sql.*;
public class Proc
{
   public static void main(String[] args) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection con=DriverManager.getConnection(
      "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

      CallableStatement stmt=con.prepareCall("{call Accept(?,?)}");
      stmt.setInt(1,101);
      stmt.setString(2,"Malati");
      stmt.execute();
      System.out.println("Record inserted successfully…!");
   }
}
```

Output:

```
Record inserted successfully…!
```

Program 12.10: Write a Java program to call the function using JDBC.

Solution:

In this example, we are calling the TotalSum() function that receives two input and returns the sum of the given number.

Step 1: Create a functionTotalSum ()

Create or replace function TotalSum(n1 in number,n2 in number)return number is temp number(8);

```
begin
temp:=n1+n2;
return temp;
end;
/
```

Step 2: Write a JDBC program to call the function

```
import java.sql.*;
public class FuncSum
{
   public static void main(String[] args) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection con=DriverManager.getConnection
                ("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
      CallableStatement stmt=con.prepareCall("{?= call TotalSum(?,?)}");
      stmt.setInt(2,10);
      stmt.setInt(3,43);
      stmt.registerOutParameter(1,Types.INTEGER);
      stmt.execute();
      System.out.println(" Sum="+stmt.getInt(1));
   }
}
```

Output: Sum=53

Program 12.11: Write a Java program using Swing for the following.

(a) Create

(b) Alter

(c) Drop

Solution:

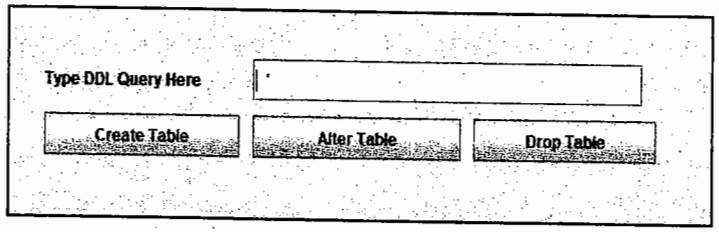
```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.sql.*;
class DDL extends JFrame implements ActionListener
{
   JLabel l1;
   JTextField t1;
```



```
   JButton b1,b2,b3;
   Connection cn;
   Statement st;
   String sql;
   public DDL()
   {
      setLayout(null);
      l1=new JLabel("Type DDL Query Here");
      t1=new JTextField();
      b1=new JButton("Create Table");
      b2=new JButton("Alter Table");
      b3=new JButton("Drop Table");
      l1.setBounds(100,100,150,30);
      t1.setBounds(260,100,300,30);
      b1.setBounds(100,140,150,30);
      b2.setBounds(260,140,160,30);
      b3.setBounds(430,140,140,30);
      add(l1);
      add(t1);
      add(b1);
      add(b2);
      add(b3);
      setSize(700,400);
      setVisible(true);
      b1.addActionListener(this);
      b2.addActionListener(this);
      b3.addActionListener(this);
      try
      {
         Class.forName("oracle.jdbc.driver.OracleDriver");
         con=DriverManager.getConnection
         ("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
         st=cn.createStatement();
      }
      catch(Exception obj)
      {
      }
   }
   public void actionPerformed(ActionEvent ae)
   {
      try
      {
         JButton b=(JButton)ae.getSource();
         sql=t1.getText();
```

```
            if(b==b1)
            {
               st.execute(sql);
               JOptionPane.showMessageDialog(null,"Table Is Created..");
            }
            if(b==b2)
            {
               st.execute(sql);
               JOptionPane.showMessageDialog(null,"Table Is Altered..");
            }
            if(b==b3)
            {
               st.execute(sql);
               JOptionPane.showMessageDialog(null,"Table Is Dropped..");
            }
         }
         catch(Exception objj)
         {
         }
      }
      public static void main(String args[])
      {
         new DDL();
      }
   }
```

Output:




```
Table Is Created
Table Is Altered
Table is Dropped
```

Program 12.12: Write a Java program for the implementation of Scrollable ResultSet.

Solution:

```
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Scroll extends JFrame implements ActionListener
{
   JLabel l1,l2,l3;
   JTextField t1,t2,t3;
```



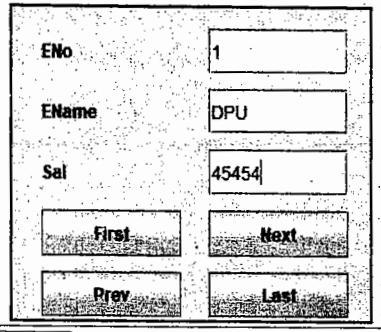
```
   JButton b1,b2,b3,b4;
   Connection cn;
   Statement st;
   ResultSet rs;
   public Scroll()
   {
      setLayout(null);
      l1=new JLabel("ENo");
      l2=new JLabel("EName");
      l3=new JLabel("Sal");

      t1=new JTextField();
      t2=new JTextField();
      t3=new JTextField();
      b1=new JButton("First");
      b2=new JButton("Next");
      b3=new JButton("Prev");
      b4=new JButton("Last");

      l1.setBounds(100,100,100,30);
      t1.setBounds(220,100,100,30);
      l2.setBounds(100,140,100,30);
      t2.setBounds(220,140,100,30);
      l3.setBounds(100,180,100,30);
      t3.setBounds(220,180,100,30);
      b1.setBounds(100,220,100,30);
      b2.setBounds(220,220,100,30);
      b3.setBounds(100,260,100,30);
      b4.setBounds(220,260,100,30);

      add(l1);
      add(t1);
      add(l2);
      add(t2);
      add(l3);
      add(t3);
      add(b1);
      add(b2);
      add(b3);
      add(b4);

      b1.addActionListener(this);
      b2.addActionListener(this);
      b3.addActionListener(this);
      b4.addActionListener(this);

      setSize(400,400);
      setVisible(true);
```

```
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con=DriverManager.getConnection
                    ("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
            st=cn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
            rs=st.executeQuery("select * from emp");
            rs.next();
            t1.setText(rs.getString(1));
            t2.setText(rs.getString(2));
            t3.setText(rs.getString(3));
        }
        catch(Exception obj)
        {
        }
    }
    public void actionPerformed(ActionEvent ae)
    {
        try
        {
            JButton b=(JButton)ae.getSource();
            if(b==b1)
            {
                rs.first();
                t1.setText(rs.getString(1));
                t2.setText(rs.getString(2));
                t3.setText(rs.getString(3));
            }
            if(b==b2)
            {
                rs.next();
                t1.setText(rs.getString(1));
                t2.setText(rs.getString(2));
                t3.setText(rs.getString(3));
            }
            if(b==b3)
            {
                rs.previous();
                t1.setText(rs.getString(1));
                t2.setText(rs.getString(2));
                t3.setText(rs.getString(3));
            }
```



```
            if(b==b4)
            {
                rs.last();
                t1.setText(rs.getString(1));
                t2.setText(rs.getString(2));
                t3.setText(rs.getString(3));
            }
        }
        catch(Exception oo)
        {
        }
    }
    public static void main(String args[])
    {
        new Scroll();
    }
}
```

Output:



Summary

- JDBC is a Java Database Connectivity which is used to connect a Java application with any type of database. After connecting to database we can perform all the database operations such as create, insert, update, delete and select using Java application.
- Every database supports Java compatible driver known as JDBC driver which will act as an interface between Java application and database.
- The JDBC uses many classes and interface. They are contained in the Java package java.sql and javax.sql.
- JDBC supports two types of processing models for accessing database.
  - **Two-tier:** Java program or application to directly communicate with the database.
  - **Three-tier:** there is no direct communication

- There are 4 types of JDBC drivers:
  - **Type-1 driver or JDBC-ODBC bridge driver :** Converts JDBC calls into ODBC calls
  - **Type-2 driver or Native-API driver:** Uses Java Native Interface call on database specific native client API.
  - **Type-3 driver or Network Protocol driver:** Communicate to JDBC middleware server using network protocol.
  - **Type-4 driver or Thin driver:** Directly interact with the database.
- The **DriverManager** provides a basic service for managing a set of JDBC drivers. Connection interface represents a database connection to a relational database. Statement interface is used to execute SQL statements against a relational database. **ResultSet** interface represents the result set of a database query.
- Class.forName() method is used to load the database driver in the memory.
- DriverManger.getConnection(connectionString,userID,password) method is used to set the connection with the database
- Statement, PreparedStatement is used to execute the SQL statement. It uses createStatement(), prepareStatement(sqlQuery) methods respectively
- CallableStatement are used to execute function and stored procedure. It uses prepareCall(Procedure) method.
- DatabaseMetaData interface provides methods to data about a database such as database product name, database product version, driver name etc.
- The ResultSetMetaData provides information about the ResultSet object like, the number of columns, names of the columns, datatypes of the columns, name of the table etc.
- getMetaData() method is used to retrieve the current metadata object.

Check Your Understanding

1. Which of the following is advantage of using JDBC connection pool?
   (a) Slow performance (b) Using more memory
   (c) Using less memory (d) Better performance
2. Which of the following is advantage of using PreparedStatement in Java?
   (a) Slow performance (b) Encourages SQL injection
   (c) Prevents SQL injection (d) More memory usage
3. Which of the following is used to call stored procedure?
   (a) Statement (b) PreparedStatement
   (c) CallableStatment (d) CalledStatement
4. Which of the following is used to limit the number of rows returned?
   (a) setMaxRows(int i) (b) setMinRows(int i)
   (c) getMaxrows(int i) (d) getMinRows(int i)
5. Which of the following is used to rollback a JDBC transaction?
   (a) rollback() (b) rollforward()
   (c) deleteTransaction() (d) RemoveTransaction()

6. What happens if you call deleteRow() on a ResultSet object?
   (a) The row you are positioned on is deleted from the ResultSet, but not from the database.
   (b) The row you are positioned on is deleted from the ResultSet and from the database.
   (c) The result depends on whether the property synchronizeWithDataSource is set to true or false.
   (d) You will get a compile error: the method does not exist because you cannot delete rows from a ResultSet.
7. Which of the following is an advantage of using PreparedStatement in Java?
   (a) Slow performance (b) Encourages SQL injection
   (c) Prevents SQL injection (d) More memory usage
8. Which type of Statement can execute parameterized queries?
   (a) PreparedStatement
   (b) ParameterizedStatement
   (c) ParameterizedStatement and CallableStatement
   (d) All kinds of Statements (i.e. which implement a sub interface of Statement)
9. Which of the following encapsulates an SQL statement which is passed to the database to be parsed, compiled, planned and executed?
   (a) DriverManager (b) JDBC driver
   (c) Connection (d) Statement
10. Which of the following is used to call a stored procedure?
    (a) Statement (b) PreparedStatement
    (c) CallableStatment (d) CalledStatement
11. What is, in terms of JDBC, a DataSource?
    (a) A DataSource is the basic service for managing a set of JDBC drivers.
    (b) A DataSource is the Java representation of a physical data source.
    (c) A DataSource is a registry point for JNDI-services.
    (d) A DataSource is a factory of connections to a physical data source.
12. Which driver is efficient and always preferable for using JDBC applications?
    (a) Type – 4 (b) Type – 1
    (c) Type – 3 (d) Type – 2

Answers

1. (d)	2. (c)	3. (c)	4. (d)	5. (a)	6. (b)	7. (c)	8. (a)	9. (d)	10. (c)
				11. (d)	12. (a)				

Practice Questions

Q.I Answer the following Questions in short:

1. What is JDBC? List any two JDBC components.
2. Explain JDBC Statement class.
3. Explain PreparedStatement of JDBC with example?
4. Explain the following terms with suitable syntax.
   (i) DriverManager
   (ii) Connection
5. Explain the statement types used to execute SQL Queries?
6. What is JDBC API and when do we use it?
7. What are different types of JDBC Drivers?
8. What is JDBC Connection? Explain steps to get Database connection in a simple Java program.
9. What is the use of JDBC DriverManager class?
10. When do we get Java.sql.SQLException: No suitable driver found?

Q.II Answer the following Questions:

1. How to get the Database server details in Java program?
2. What is JDBC Statement?
3. What is the difference between execute, executeQuery, executeUpdate?
4. What is JDBC PreparedStatement?
5. How to set NULL values in JDBC PreparedStatement?
6. What are the benefits of PreparedStatement over Statement?
7. What is the limitation of PreparedStatement and how to overcome it?
8. What is JDBC ResultSet?
9. What are different types of ResultSet?
10. How to use JDBC API to call Stored Procedures?
11. Write the steps for JDBC connectivity?
12. Explain ResultSetMetaData?
13. Explain DatabaseMetaData?

Q.III Write short note on:

1. DriverManagerComponent
2. Connection
3. Statement
4. PreparedStatement
5. ResultSet
6. Connection (repeat)
7. Connection String
8. SQL String
9. DatabaseMetaData
10. ResultSetMetaData

■■■

13... Java Server Technologies

Objectives...

- To get familiar with Web Application Basics and Architecture.
- To know challenges in Web Application Development.
- To study web based application development using Servlet and JSP.
- To study lifecycle of Servlet and JSP.
- To understand working of client and server communication using request and response functionality.
- To Exploring Deployment Descriptor using web.xml.

13.1 INTRODUCTION

- A Java web application is a collection of dynamic resources (such as Servlet, Java Server Pages, Java classes and jars) and static resources (HTML pages and pictures). A Java web application can be deployed as a WAR (Web Archive) file.
- Java Server technology provides a well-defined programming model and various tag libraries.
- The tag libraries contain tag handlers that implement the component tags. These features significantly ease the burden of building and maintaining web applications with server-side user interfaces (UIs). With minimal effort, you can complete the following tasks:
  - Create a web page.
  - Drop components onto a web page by adding component tags.
  - Bind components on a page to server-side data.
  - Wire component-generated events to server-side application code.
  - Save and restore application state beyond the life of server requests.
  - Reuse and extend components through customization.
- This chapter provides an overview of Java Server technology.

13.2 WEB APPLICATION BASICS

13.2.1 Web Application

- A Web application (Web app) is an application program that is stored on a remote server and delivered over the Internet through a browser interface. Web services are Web apps by definition and many, although not all, websites contain Web apps.
- A web application is a collection of web components. It provides features to end users through an interface typically presented in a web browser.
- A web application is based on a client-server model. In this model, the client is the web browser, and the web server is the feature set that runs remotely.
- Web consists of billions of clients and server connected through wires and wireless networks. The web clients make requests to web server.
- The web server receives the request, finds the resources and returns the response to the client. When a server answers a request, it usually sends some type of content to the client. The client uses web browser to send request to the server as shown in Fig. 13.1
- The server often sends response to the browser with a set of instructions written in HTML (HyperText Markup Language). All browsers know how to display HTML page to the client.
- Web applications do not need to be downloaded since they are accessed through a network. Users can access a Web application through a web browser such as Google Chrome, Mozilla Firefox or Safari.
- For a web app to operate, it needs a Web server, application server, and a database. Web servers manage the requests that come from a client, while the application server completes the requested task. A database can be used to store any needed information.
- A **website** is a collection of static files (webpages) such as HTML pages, images, graphics etc. A **Web application** is a web site with dynamic functionality on the server. Google, Facebook, Twitter are examples of web applications.
- A web application has many different uses, and with those uses, comes many potential benefits. Some common benefits of Web apps include:
  - Allowing multiple users access to the same version of an application.
  - Web apps don't need to be installed.
  - Web apps can be accessed through various platforms such as a desktop, laptop, or mobile.
  - Can be accessed through multiple browsers.
- **Working of Web technology:**
  1. The client sends the request to the web server.
  2. The web server receives the request.

3. The web server passes the request to the corresponding Servlet.
4. The Server processes the request and generates the response in the form of output.
5. The Servlet sends the response back to the web server.
6. The web server sends the response back to the client and the client browser displays it on the screen.

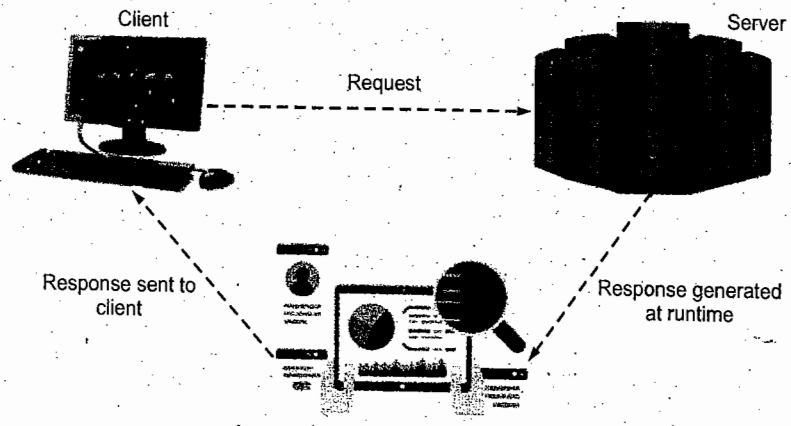


Fig. 13.1: Working of Web Technology

- In this Figure you can see, a client sends a request to the server and the server generates the response, analyses it and sends the response back to the client.

13.2.2 Servlet Basics

- Before Servlet, CGI (Common Gateway Interface) programming was used to create web applications but it has lot of limitations.
- Limitations of CGI:
  - High response time because CGI programs execute in their own OS shell.
  - CGI is not scalable.
  - CGI programs are not always secure or object-oriented.
  - It is Platform dependent.
- To overcome these limitations Servlet Technology is introduced for developing web applications. It uses Java language to create web applications.
- Web applications are helper applications that reside at web server and build dynamic web pages. A dynamic page could be anything like a page that randomly chooses picture to display or even a page that displays the current time.
- As Servlet Technology uses Java, web applications made using Servlet are Secured, Scalable and Robust.

Advantages of using Servlet:

1. Less response time because each request runs in a separate thread.
2. Servlets are scalable.
3. Servlets are robust and object oriented.
4. Servlets are platform independent.

Difference between Servlet and CGI:

Table 13.1: Difference between Servlet and CGI

Basis for Comparison	Common Gateway Interface	Servlets
Basic	Programs are written in the native OS.	Programs employed using Java.
Platform dependency	Platform dependent.	Does not rely on the platform.
Creation of process	Each client request creates its own process.	Processes are created depending on the type of the client request.
Conversion of the script	Present in the form of executables (native to the server OS).	Compiled to Java Byte Code.
Runs on	Separate process.	JVM.
Security	More vulnerable to attacks.	Can resist attacks.
Speed	Slower.	Faster.
Processing of script	Direct.	Before running the scripts it is translated and compiled.
Portability	Cannot be ported.	Portable.

13.3 ARCHITECTURE AND CHALLENGES OF WEB APPLICATION

13.3.1 Architecture of Web Application

- Web application architecture defines the interactions between applications, middleware systems and databases to ensure multiple applications can work together.
- When a user types in a URL and taps "Go/Search," the browser will find the Internet-facing computer the website lives on and requests that particular page.
- The server then responds by sending files over to the browser. After that action, the browser executes those files to show the requested page to the user. Now, the user gets to interact with the website. All of these actions are executed within a seconds.
- In web applications, there are two programs running concurrently the **server and the client side.**
  - The code which lives in the browser and responds to user input.
  - The code which lives on the server and responds to HTTP requests.

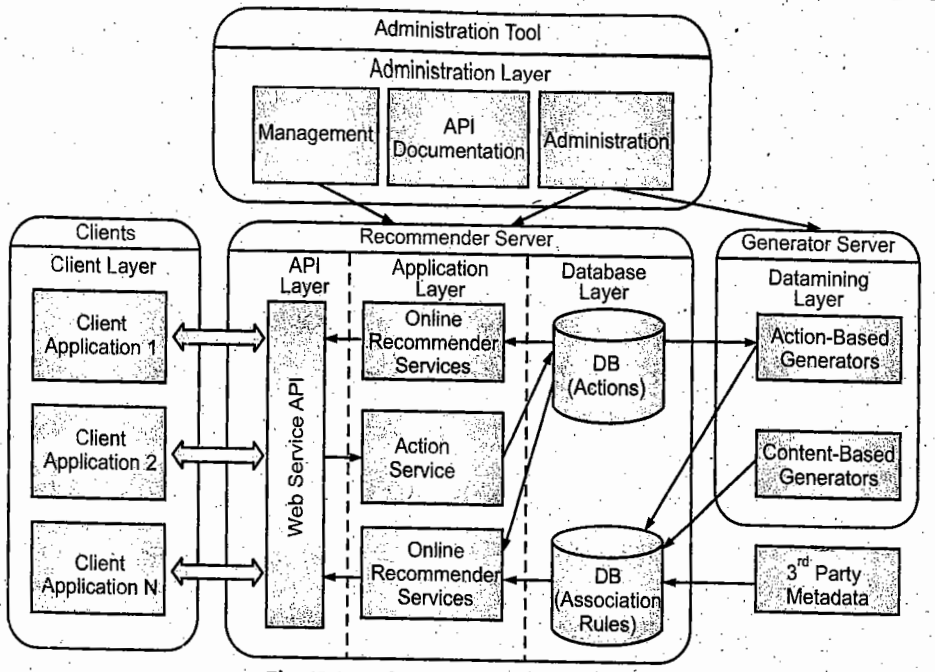


Fig. 13.2: Web Application Architecture

- These components can be categorized into two areas:
  1. User interface app components
  2. Structural components.

1. User interface app components:

- It refers to web pages displaying dashboards, logs, notifications, configuration settings and more.
- They are not relevant to the structural development of the application and are more user interface/experience oriented.

2. The structural components:

- It consist following app development process:

(a) The web browser or client:

- The web browser is execution of a web app functionality, with which the user interacts?
- The content delivered to the client can be developed using HTML, JavaScript, and CSS and does not require operating system related adaptations.
- The web browser or client manages how end users interact with the application.

(b) The web application server:

- It manages business logic and data persistence and can be built using PHP, Python, Java, Ruby, .NET, Node.js, among other languages.
- It is comprised of at least a centralized hub or control center to support multi-layer applications.

(c) **The database server:**

- It provides and stores relevant data for the application.
- It may also supply the business logic and other information that is managed by the web application server.

13.3.2 Challenges of Web Application

- Web development is an emerging technology widely used today. Better and user-friendly interfaces are in demand. When it comes to developing a successful web application, there are a number of factors defining that success.
- There are following challenges focused during web development:

1. **User Interface and User Experience:**

- Now a day's almost 90% people are using web application for their work this is the reason, simpler and customer oriented web application are highly expected now.
- Sometimes it's the small UI elements that make the biggest impact. In the era of Smartphone's, websites should be responsive enough on the smaller screens.
- If your web applications frustrate or confuse users, then it is difficult to maintain your customer's loyalty for your website.
- Website navigation is another part often neglected by developers. Intuitive navigation creates a better user experience for the website visitor. Intuitive navigation is leading your audience to the information they are looking without a learning curve. And when the navigation is intuitive, visitors can find out information without any pain, creating a flawless experience preventing them from visiting the competitors.

2. **Scalability:**

- Scalability is about load balancing between the servers, hence; when the load increases (i.e. more traffic on the page) additional servers can be added to balance it.
- You should not just throw the all load on a single server but you should design the software such that it can work on a cluster of servers.
- Service-Oriented Architecture (SOA) can help in improving scalability when more and more servers are added. SOA gives you the flexibility to change easily. Service oriented architecture is a design where application components provide services to other components through the communication protocol, basically over a network.

3. **Performance:**

- Generally website speed has the major importance for a successful website. Slow web applications are a failure.
- It is said that think about performance first before developing the web application.
- Some of the performance issues are poorly written code, Un-Optimized Databases, Unmanaged Growth of data, Traffic spikes, Poor load distribution, Default configuration, troublesome third party services, etc.

4. **Knowledge of Framework and Platforms:**

- Frameworks are the kick start for development languages: they boost performance, offer libraries of coding and extend capabilities, so developers need not do hand-coding web applications from the ground up.
- Frameworks offer features like models, APIs, snippets of code and other elements to develop dynamic web applications. Some of the frameworks have a rigid approach to development and some are flexible.
- Common examples of web frameworks are PHP, ASP.Net, Ruby on Rails and J2EE.
- Web platforms provide client libraries build on existing frameworks required to develop a web application or website. A new functionality can be added via external API.

5. **Security:**

- As compare to design and user experience, web app security is often neglected. But security should be considered throughout the software development life cycle, especially when the application is dealing with the vital information such as payment details, contact information, and confidential data.
- There are many things to consider when it comes to web application security such as denial of service attacks, the safety of user data, database malfunctioning, unauthorized access to restricted parts of the website, etc.
- Some of the security threats are Cross-Site Scripting, Phishing, Cross-Site Request Forgery, Shell Injection, Session Hijacking, SQL Injection, Buffer Overflow, etc. The website should be carefully coded to be safe against these security concerns.

13.4 INTRODUCTION TO SERVLET

- **Servlet** is a server-side Java program that executes on tomcat web server.
- Servlet can be described in many ways, depending on the context.
  - Servlet is a technology which is used to create a web application.
  - Servlet is an API that provides many interfaces and classes including documentation.
  - Servlet is an interface that must be implemented for creating any Servlet.
  - Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
  - Servlet is a web component that is deployed on the server to create a dynamic web page.
- Servlet is server side scripting language so it does not have main() method.
- Servlet application compiled by using javac compiler explicitly.
- For developing Servlet application in java following API's must be imported:

```
javax.servlet.*;
javax.servlet.http.*;
java.io.*
```

- There are two types Servlet:
  - GenericServlet
  - HttpServlet
- The GenericServlet and HttpServlet are the classes belongs to packages javax.servlet.* and javax.servlet.http.*.

13.4.1 Life Cycle of Servlet

- The Servlet is a java program executed on tomcat web server. It can be created either by inheriting GenericServlet or HttpServlet class.
- Servlet life cycle contains five steps:

Step 1: Loading of Servlet

- When the web server (e.g. Apache Tomcat) starts up, the servlet container deploys and loads all the Servlet.

Step 2: Creating instance of Servlet

- Once all the Servlet classes loaded, the servlet container creates instances of each servlet class.
- Servlet container creates only one instance per Servlet and all the requests to the servlet are executed on the same servlet instance.

Step 3: Invoke init() Method

- Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet.
- This method initializes the servlet. There are certain init parameters that you can specify in the deployment descriptor (web.xml) file.
- For example, if a servlet has value >=0 then its init() method is immediately invoked during web container startup.
- The init() method is called only once during the life cycle of servlet.

Step 4: Invoke service() Method

- Each time the web server receives a request for servlet, it creates a new thread that calls service() method.
- If the servlet is GenericServlet then the request is served by the service() method.
- If the servlet is HttpServlet then service() method receives the request and dispatches it to the correct handler method(POST or GET) based on the type of request.
- For example if it is a **GET Request** the service() method would dispatch the request to the doGet() method by calling the doGet() method with request parameters. Similarly the requests like **POST, HEAD, PUT** etc. are dispatched to the corresponding handlers doPost(), doHead(), doPut() etc. by service() method of servlet.

Step 5: Invoke destroy() Method

- When servlet container shuts down(this usually happens when we stop the web server), it unloads all the Servlet and calls destroy() method for each initialized Servlet.

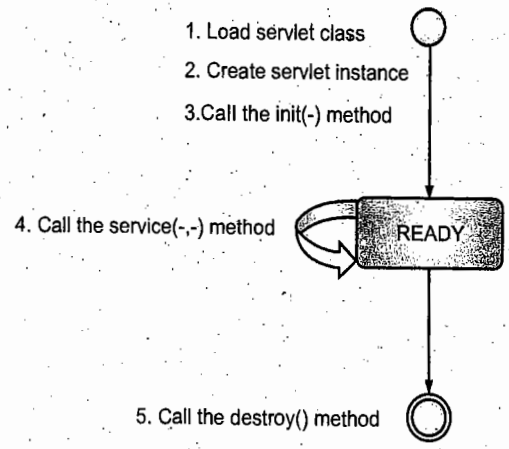


Fig. 13.3: Servlet Lifecycle

13.4.2 Methods of Servlet

- Servlet is an interface having following methods:

Table 13.2: Methods of Servlet

Method	Description
`public void init(ServletConfig config)`	Initializes the Servlet. It is the life cycle method of Servlet and invoked by the web container only once.
`public void service(ServletRequest request, ServletResponse response)`	Provides response for the incoming request. It is invoked at each request by the web container.
`public void destroy()`	Is invoked only once and indicates that Servlet is being destroyed.
`public ServletConfig getServletConfig()`	Returns the object of ServletConfig.
`public String getServletInfo()`	Returns information about Servlet such as writer, copyright, version etc.

13.4.3 How Servlet Works?

- There are three main terminology used in servlet:
  - **Web Server:** It can handle HTTP Requests send by clients and responds the request with an HTTP Response.

- **Web Application (webapp):** Webapp is an application developed by you to create a servlet. It consist HTML, java, class, web.xml files.
- **Web Container:** It is also known as Servlet Container and Servlet Engine. It is a part of Web Server that interacts with Servlets. This is the main component of Web Server that manages the life cycle of Servlets.

- **Servlet can be worked as follow:**

1. When the web server (e.g. Apache Tomcat) starts up, the servlet container deploy and loads all the Servlet. During this step Servlet container creates ServletContext object. ServletContext is an interface that defines the set of methods that a servlet can use to communicate with the servlet container.
2. Once the servlet is loaded, the servlet container creates the instance of servlet. For each instantiated servlet, its init () method is invoked.
3. Client (user browser) sends an Http request to web server on a certain port. Each time the web server receives a request, the servlet container creates HttpServletRequest and HttpServletResponse objects. The HttpServletRequest object provides the access to the request information and the HttpServletResponse object allows us to format and change the http response before sending it to the client.
4. The servlet container spawns a new thread that calls service () method for each client request. The service() method dispatches the request to the correct handler method based on the type of request. For example if server receives a Get Request the service() method would dispatch the request to the doGet() method by calling the doGet() method with request parameters. Similarly the requests like Post, Head, Put etc. are dispatched to the corresponding handlers doPost(), doHead(), doPut() etc. by service() method of servlet.
5. When servlet container shuts down, it unloads all the Servlet and calls destroy() method for each initialized Servlet.

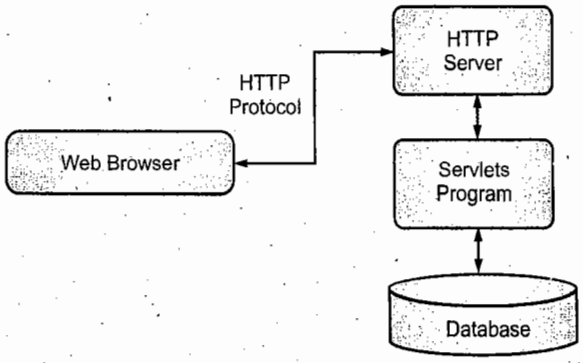


Fig. 13.4: Working with Servlet

13.5 DEVELOPING AND DEPLOYING SERVLET

13.5.1 Developing Servlet

1. **Select C:\Program Files\Apache Software Foundation\Apache Tomcat 8.0.27\webapps**
   Create your own directory structure as mentioned below:

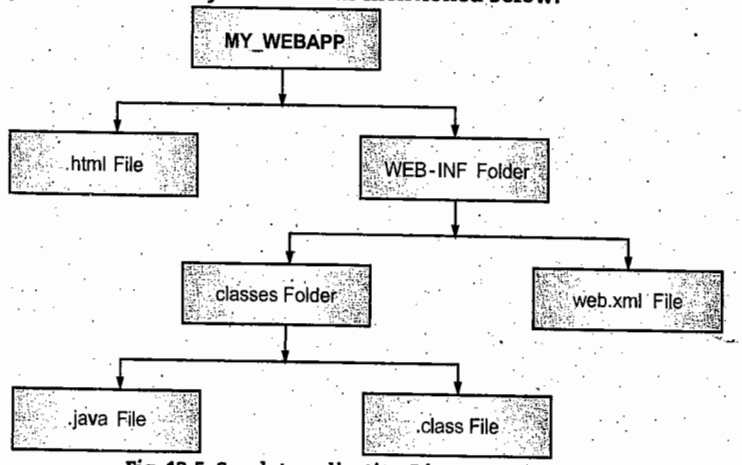


Fig. 13.5: Servlet application Directory structure

2. **Write a client side request application and save it into your web application folder using .html extension.**
   For Example: Accepting name from user.
   Save this file as "index.html"
   Note: in form tag the action parameter should be the servlet ".class" file name.

```
<html>
<body>
<form method=post action="Demo">
Enter your Name<input type=text name="nm">
<input type=submit name=submit value=submit>
</form>
</body>
</html>
```

3. **The WEB-INF folder is a web information folder which holds the information about your web application such as the .class file and deployment descriptor i.e. web.xml**

```
<web-app>
 <servlet>
 <servlet-name> My_WEBAPP </servlet-name>
         <servlet-class>Demo</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name> My_WEBAPP </servlet-name>
      <url-pattern>/Demo</url-pattern>
 </servlet-mapping>
</web-app>
```

4. **Now write the java program and save it in classes folder. Compile it and a ".class file" will be generated.**

Note: Classes folder must be represented in a small case letter.

The ".class file" very important here it is used in form tag of HTML file as well as in web.xml for mapping

For Example: In the ".html" file we used "Demo" as class file name then we must create a "Demo.java" servlet file.

Example to accept name from user and display on screen.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Demo extends HttpServlet
{
   public void doPost(HttpServletRequest request, HttpServletResponse
   response)
   {
      try
      {
         response.setContentType("text/html");
         // set the type of reponse given to client
         PrintWriter out = response.getWriter();
         /Y For using println() to display output
         String name = request.getParameter("nm");
         //accepting input request from client (Note: nm is a variable
         from .html file)
         out.print("Hello "+name);
         out.close();
      }
      catch(Exception exp)
      {
         System.out.println(exp);
      }
   }
}
```

Save the above program in C:\Program Files\Apache Software Foundation\Apache Tomcat 8.0.27\webapps\My_WEBAPP\WEB-INF\classes\. Compile and a class file "Demo.class:" is generated.

13.5.2 Deployment of Servlet

- Open apache tomcat server application and start service.

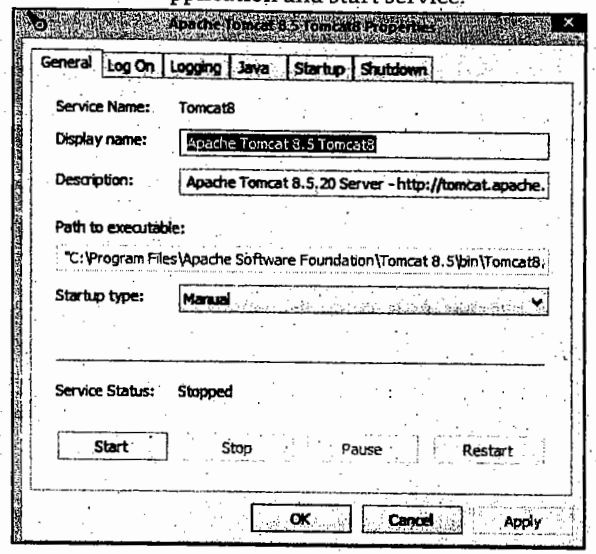


Fig. 13.6: Apache tomcat server

- Once the server services started open a web browser and start Apache tomcat server using http://localhost:8080.
- The home page of Apache tomcat will be open.

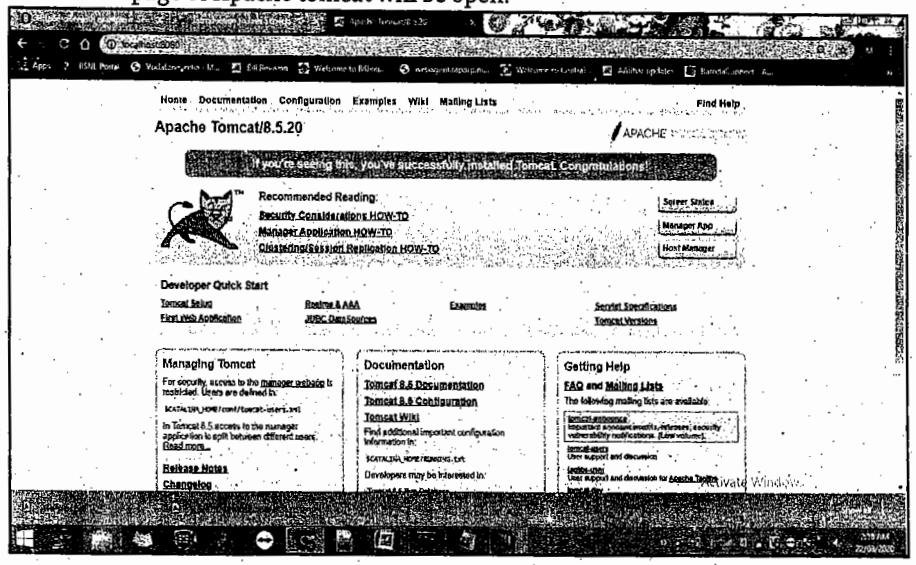


Fig. 13.7: Home page of Apache tomcat 8.5.20

- Now type your web application name in URL address bar and your application will be executed.

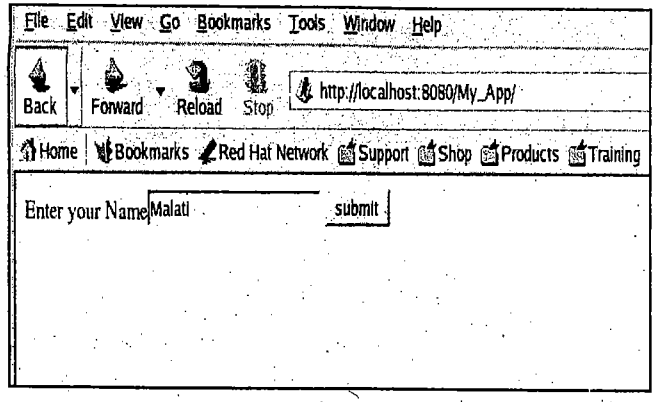


Fig. 13.8: Executing client request

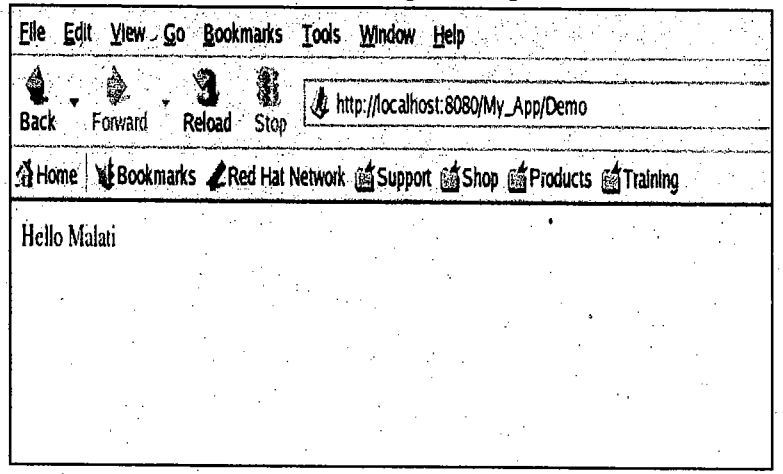


Fig. 13.9: Servlet response "Demo.java"

13.5.3 Exploring Deployment Descriptor (web.xml)

- A web application's deployment descriptor describes the classes, resources and configuration of the application and how the web server uses them to serve web requests.
- When the web server receives a request for the application, it uses the deployment descriptor to map the URL of the request to the code that ought to handle the request.
- The deployment descriptor is a file named **web.xml.** It resides in the app's WAR under the **WEB-INF/** directory.
- The file is an XML file whose root element is **<web-app>**.
- The web.xml file provides configuration and deployment information for the web components that comprise a web application.

- **For example:**

```
<web-app>
<servlet>
 <servlet-name> My_WEBAPP </servlet-name>
   <servlet-class>Demo</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name> My_WEBAPP </servlet-name>
   <url-pattern>/Demo</url-pattern>
 </servlet-mapping>
</web-app>
```

- The web.xml defines mappings between URL paths and the Servlet that handle requests with those paths.
- The web server uses this configuration to identify the servlet to handle a given request and call the class method that corresponds to the request method (e.g. the doGet() method for HTTP GET requests).
- To map a URL to a servlet, you declare the servlet with the <servlet> element, then define a mapping from a URL path to a servlet declaration with the <servlet-mapping> element.
- The <servlet-mapping> element specifies a URL pattern and the name of a declared servlet to use for requests whose URL matches the pattern.
- The<url-pattern>represents the URL of the web application which will be visible in the address bar of web browser

13.6 HANDLING REQUEST AND RESPONSE

- In the client and server technology: client sends the request to server and server sends the response to client as a result of request
- The request methods are used to accept the request from client i.e. input from html file in to servlet file.
- In Servlet ServletRequest and HttpServletRequest interface supports many request methods.
- ServletResponse and HttpServletResponse interface supports many response methods.

Table 13.3: Methods of ServletRequest

ServletRequest Methods	Description
`String getParameter(String name)`	Obtains the value of a parameter sent to the servlet as part of a get or post request. The name argument represents the parameter name.

contd. ...

`Enumeration getParameterNames()`	Returns the names of all the parameters sent to the servlet as part of a post request.
`String[] getParameterValues(String name)`	For a parameter with multiple values, this method Returns an array of strings containing the values for a specified servlet parameter.
`String getProtocol()`	Returns the name and version of the protocol the request uses in the form *protocol/majorVersion.minorVersion*, for example, HTTP/1.
`String getRemoteAddr()`	Returns the Internet Protocol (IP) address of the client that sent the request.
`String getRemoteHost()`	Returns the fully qualified name of the client that sent the request.
`int getServerPort()`	Returns the port number on which request was received.
`String getServerName()`	Returns the host name of the server that received the request.
HttpServletRequest Methods	
`Cookie[] getCookies()`	Returns an array of Cookie objects stored on the client by the server.
`HttpSession getSession(boolean create)`	Returns an HttpSession object associated with the client's current browsing session. This method can create an HttpSession object (True argument)if one does not already exist for the client.
`String getServletPath()`	Returns the part of this request's URL that calls the servlet.
`String getMethod()`	Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.
`String getQueryString()`	Returns the query string that is contained in the request URL after the path.
`String getPathInfo()`	Returns any extra path information associated with the URL the client sent when it made this request.
`String getRemoteUser()`	Returns the login of the user making this request, ifm the user has been authenticated, or null if the user has not been authenticated.

contd. ...

The Response methods are used to give response(output) to client from server i.e. servlet	
ServletResponse Methods	
`OutputStream getOutputStream()`	Obtains a byte-based output stream for sending binary data to the client.
`PrintWriter getWriter()`	Obtains a character-based output stream for sending text data (usually HTML formatted text) to the client.
`void setContentType(String type)`	Specifies the content type of the response to the browser. The content type is also known as MIME (Multipurpose Internet Mail Extension) type of the data. For examples, "text/html", "image/gif"etc.
`String setContentLength(int len)`	Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
HttpServletResponse Methods	
`void addCookie(Cookie cookie)`	Used to add a Cookie to the header of the response to the client.
`void sendError(int ec)`	Sends an error response to the client using the specified status.
`void sendError(int ec, String msg)`	Sends an error response to the client using the specified status code and descriptive message.
`void sendRedirect (String url)`	Sends a temporary redirect response to the client using the specified redirect location URL.
`void setHeader(String name, String value)`	Sets a response header with the given name and value.

To handle request and response in servlet java uses following methods:

1. **service(ServletRequest obj,ServletResponse obj):**
   - This method is supported by GenericServlet class. This is a lifecycle method which get automatically override by HttpServlet
2. **public void doPost(HttpServletRequest request, HttpServletResponse response):**
   - This method is supported by HttpServlet class.
   - This method throws IOException and ServletException.
   - The doPost(request, response) method is executed when a web page is activated using a click event(Example: button click, hyperlink).
3. **public void doGet(HttpServletRequest request, HttpServletResponse response):**
   - This method is supported by HttpServlet class.
   - This method throws IOException and ServletException
   - The doGet(request, response) method is executed when a web page is activated or called by specifying address (URL) in web browser's address bar.

Difference between GET and POST requests:

Table 13.4: Difference between GET and POST request

GET Request	POST Request
Data is sent in header to the server	Data is sent in the request body
Get request can send only limited amount of data	Large amount of data can be sent.
Get request is not secured because data is exposed in URL	Post request is secured because data is not exposed in URL.
Get request can be bookmarked and is more efficient.	Post request cannot be bookmarked.

13.7 INTRODUCTION TO JSP

- JSP is Java Server Page, which is a dynamic web page and used to build dynamic websites.
- To run JSP, we need web server which can be tomcat provided by apache.
- JSP is dynamic file where as Html file is static. HTML cannot get data from database or dynamic data.
- JSP can be interactive and communicate with database and controllable by programmer.
- It is saved by extension of **.jsp**. Each Java server page is compiled into a servlet before it can be used. This is normally done when the first request to the JSP page is made.
- There is no need to compile jsp page it gets compiled implicitly by JSP Engine.
- For writing JSP application its delimiters are used: Starting Delimiter (<%) and ending Delimiter(%>)

13.7.1 Lifecycle of JSP

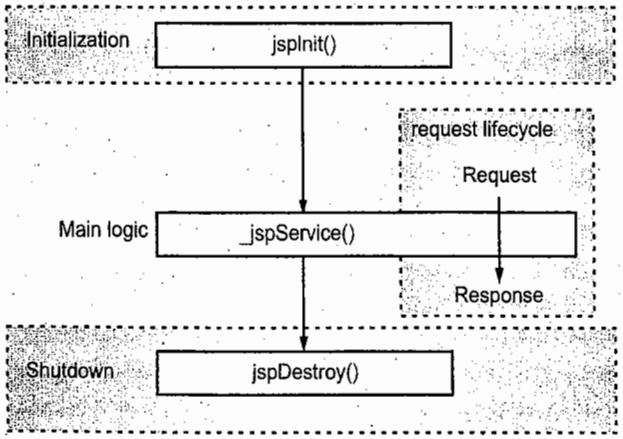


Fig. 13.10: JSP lifecycle

- A Java Server Page life cycle is defined as the process started with its creation which later translated to a servlet and follows servlet lifecycle. This is how the process goes on until its destruction.
- JSP Life Cycle is defined as translation of JSP Page into servlet as a JSP Page needs to be converted into servlet first in order to process the service requests. The Life Cycle starts with the creation of JSP and ends with the disintegration of that.
- When the browser asks for a JSP, JSP engine first checks whether it needs to compile the page. If the JSP is last compiled or the recent modification is done in JSP, then the JSP engine compiles the page.
- Compilation process of JSP page involves three steps:
  - Parsing of JSP
  - Turning JSP into servlet
  - Compiling the servlet

Following steps explain the JSP life cycle:

1. Translation of JSP page
2. Compilation of JSP page(Compilation of JSP page into _jsp.java)
3. Class loading (_jsp.java is converted to class file _jsp.class)
4. Instantiation(Object of generated servlet is created)
5. Initialization(_jspinit() method is invoked by container)
6. Request Processing(_jspservice() method is invoked by the container)
7. Destroy (_jspDestroy() method invoked by the container)

JSP Lifecycle is depicted in the below diagram:

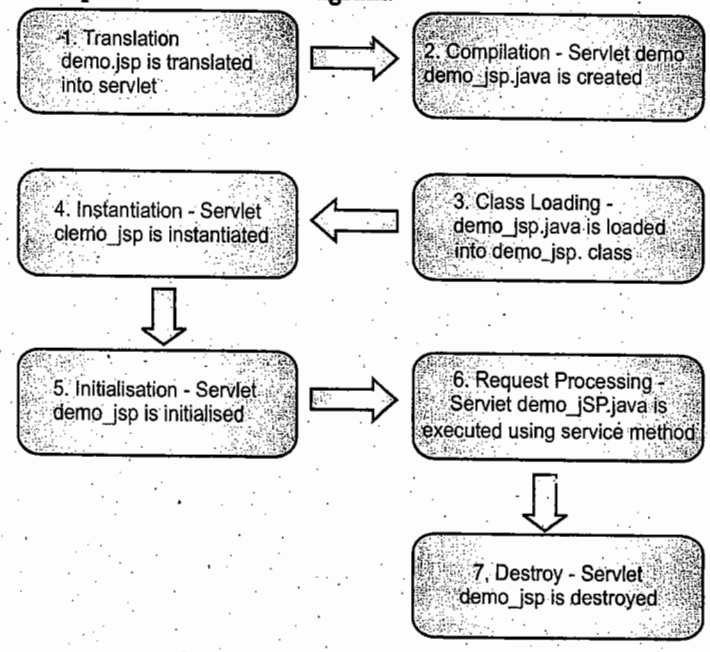


Fig. 13.11: JSP lifecycle and translation

13.7.2 A JSP Elements

- A JSP contains three important types of elements:

1. JSP Directives:

- Directives are message to the JSP container that enable the programmer to specify page setting to include content from other resources & to specify custom tag libraries for use in a JSP.
- **Syntax:** `<%@ name attribute1="….", attribute2="…"…%>`
- **There are three types of Directives:**

(a) Page Directive:

- The page directives specify global settings for the JSP in the JSP container. There can be many page directives, provided that there is only one occurrence of each attribute.
- **Syntax:** `<%@ page attribute="value" %>`
- **For example:**

```
<%@ page
[ language="java" ]
[ extends="package.class" ]
[ import="{package.class | package.*}, ..." ] [ session="true|false" ]
[ buffer="none|8kb|sizekb" ] [ autoFlush="true|false" ]
[ isThreadSafe="true|false" ] [ info="text" ]
[ errorPage="relativeURL" ]
[ contentType="mimeType [ ; charset=characterSet ]" | "text/html ;
charset=ISO-8859-1" ]
[ isErrorPage="true|false" ]
[ pageEncoding="characterSet | ISO-8859-1" ]
%>
```

Program 13.1: Program to display current Date.

```
<html>
<body>
<%@ page import="java.util.Date" %>
Today is: <%= new Date() %>
</body>
</html>
```

Output:

```
Tue Sep 01 21:20:28 IST 2020
```

(b) Include Directive:

- An include directive is used to include the contents of any resource it may be JSP file, HTML file or text file.
- If we want to use any file in JSP file then we can use include directive.

- An include directive includes the original content of the included resource at page translation time (the JSP page is translated only once so it will be better to include static resource).
- **Syntax:**<%@ include file="resourceName" %>
- **Note:** `resourseName is any file which we want to use in current JSP file`

Program 13.2: Program to include welcome.html

```
<html>
<body>
<%@ include file="welcome.html" %>
Today is: <%= java.util.Calendar.getInstance ().getTime() %>
</body>
</html>
```

Output:

```
Tue Sep 01 21:22:21 IST 2020
```

(c) Taglib Directive:

- The JSP taglib directive is used to define a tag library that defines many tags.
- We use the TLD (Tag Library Descriptor) file to define the tags. In the custom tag section we will use this tag so it will be better to learn it in custom tag.
- **Syntax:** `<%@ taglib uri="urioftheteglibrary" prefix="prefixoftaglibrary" %>`
- **For Example:** In this example, we are using our tag named currentDate. To use this tag we must specify the taglib directive so the container may get information about the tag.

Program 13.3: Program to demonstrate taglib directive.

```
<html>
<body>
<%@ tagliburi="http://www.javatpoint.com/tags" prefix="mytag" %>
<mytag:currentDate/>
</body>
</html>
```

Output:

```
It Gives description about given website
```

2. Scripting Elements:

- In JSP, java code can be written inside the jsp page using the scriptlet tag.
- The scripting elements provides the ability to insert java code inside the jsp.
- There are three types of scripting elements:

(a) Declaration Tag:

- A declaration declares one or more variables or methods that you can use in Java code later in the JSP file.
- **Syntax:** `<%! Java declaration statements %>`

- **Example:**

```
<%! private int count = 0; %>
<%! int i = 0; %>
```

(b) Expressions Tag:

- An expression element contains a java expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.
- **Syntax:** `<%= expression %>`
- **Example:** `Your name is <%= request.getParameter("name") %>`

(c) Scriptlet Tag:

- A scriptlet contains a set of java statements which is executed.
- A scriptlet can have java variable and method declarations, expressions, use implicit objects and contain any other statement valid in java.
- **Syntax:** `<% statements %>`

Program 13.4: Write a JSP script to say Hello message to the User.

```
<html>
<body>
<form method="Get" action="Name.jsp">
User Name <input type="text" name="username">
<input type="submit" value="Submit">
<%
String name = request.getParameter("userName");
out.println("Hello " + name);%>
```

Output:

```
User Name : DPU
Hello DPU
```

3. **Actions:** Actions encapsulates functionally in predefined tags that programmers can embedded in a JSP.

13.7.3 Implicit objects used in JSP

- The implicit objects are system defined and can be used directly without declaring and initializing.

Table 13.5: Implicit Objects used in JSP

Implicit object	Description
Application	A javax.servlet.ServletContext object that represents the container in which the JSP executes. It allows sharing information between the jsp page's Servlet and any web components within the same object.
Config	A javax.servlet.ServletConfig object that represents the JSP configuration options. As with Servlet, configuration options can be specified in a Web application descriptor (web.xml). The method getInitParameter() is used to access the initialization parameters.

contd. ...

Exception	A java.lang.Throwable object that represents an exception that is passed to a JSP error page. This object is available only in a JSP error page.
Out	A javax.servlet.jsp.JspWriter object that writes text as part of the response to a request. This object is used implicitly with JSP expressions and actions that insert string content in a response.
Page	An Object that represents the current JSP instance.
pageContext	A javax.servlet.jsp.PageContext object that provides JSP programmers with access to the implicit objects discussed in this table.
Request	An object that represents the client request and is normally an instance of a class that implements HttpServletRequest. If a protocol other than HTTP is used, this object is an instance of a subclass of javax.servlet.Servlet-Request. It uses the getParameter() method to access the request parameter.
Response	An object that represents the response to the client and is normally an instance of a class that implements HttpServletResponse (package javax.servlet.http). If a protocol other than HTTP is used, this object is an instance of a class that implements javax.servlet.ServletResponse.
Session	A javax.servlet.http.HttpSession object that represents the client session information. This object is available only in pages that participate in a session.

13.7.4 Advantages of JSP over Servlet

- There are many advantages of JSP over the Servlet. They are as follows:
  1. **Extension to Servlet:** JSP technology is the extension to Servlet technology. We can use all the features of the Servlet in JSP. In addition to, we can use implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP development easy.
  2. **Easy to maintain:** JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.
  3. **Fast Development: No need to recompile and redeploy.** If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.
  4. **Less code than Servlet:** In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

13.7.5 Steps for writing JSP web application in Java

1. **Select C:\Program Files\Apache Software Foundation\Apache Tomcat 8.0.27\webapps\ create your own directory structure as mentioned below:**

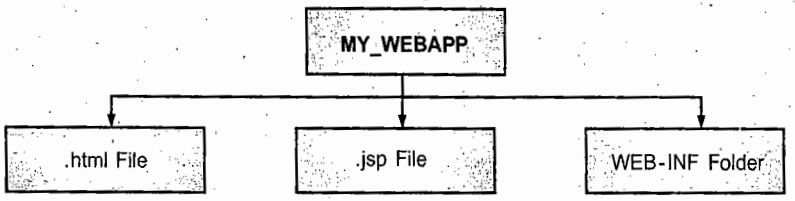


Fig. 13.12: JSP application Directory structure

2. **Write a client side request application and save it into your web application folder using .html extension.**

   For Example: Accepting name from user

   Save this file as "index.html"

   Note: in form tag the action parameter should be the name of .jsp file.

```
<html>
<body>
<form method=post action="welcome.jsp">
Enter your Name<input type=text name="nm">
<input type=submit name=submit value=submit>
</form>
</body>
</html>
```

3. **Write a server side request application and save it into your web application folder using .jsp extension.**

   For Example: Display name of user

   Save this file as "welcome.jsp"

```
<html>
<body>
<%
   String name=request.getParameter("nm");
   out.println("Welcome...!"+name);
%>
</body>
</html>
```

   Output:

```
Enter your Name : DYP
Welcome....!DYP
```


PROGRAMS

Program 13.5: Write a JSP Application to accept nick name and Name from an user, If visit count is even then display name otherwise display nickname.

Solution:

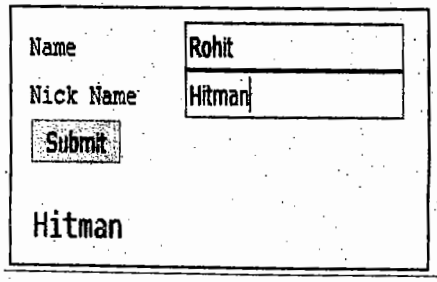
index.html

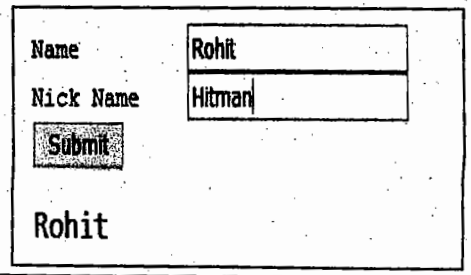
```
<html>
<body>
<form method="Get" action="NickName.jsp">
<pre>
 Name <input type="text" name="t1">
 Nick Name <input type="text" name="t2">
 <input type="submit" value="Submit">
</pre>
</form>
</body>
</html>
```

NickName.jsp

```
<%! static int cnt;%>
<%
 String nm,nk;

 nm=request.getParameter("t1");
 nk=request.getParameter("t2");
 cnt++;
 if(cnt%2==0)
 out.println(nk);
 else
 out.println(nm);
%>
```

Output:





Program 13.6: Write a JSP application to check whether given mail ID is valid or not.

Solution:

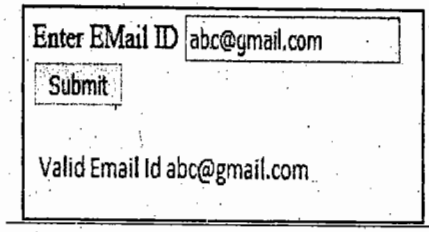
index.html

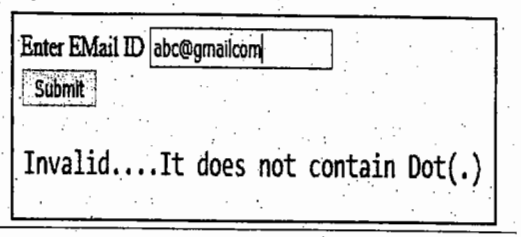
```
<html>
<body>
<form method="Get" action="AssVal.jsp">
Enter EMail ID <input type="text" name="t1"> <br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

AssVal.jsp

```
<%
String em;
int i,l,scnt=0,dcnt=0;
char ch;
em=request.getParameter("t1");
l=em.length();
for(i=0;i<l;i++)
{
ch=em.charAt(i);
if(ch=='.')
{
dcnt++;
}
if(ch=='@')
{
scnt++;
}
}
if(dcnt==0)
{
out.println("Invalid....It does not contain Dot(.)");
}
else if(scnt>1 || scnt<1)
{
out.println("It does not have @ symbol or more than one @ symbol");
}
else
out.println("Valid Email Id " + em);
%>
```


Output:





Program 13.7: Write a servlet application to display marks on to the browser.

Solution:

index.html

```
<html>
<body>
<form method="Get" action="Mark">
<pre>
 Seat No      <input type="text" name="t1">
 Stud Name    <input type="text" name="t2">
 Class        <input type="text" name="t3">
 Total Marks  <input type="text" name="t4">
 <input type="submit" value="Submit">
</pre>
</form>
</body>
</html>
```

web.xml

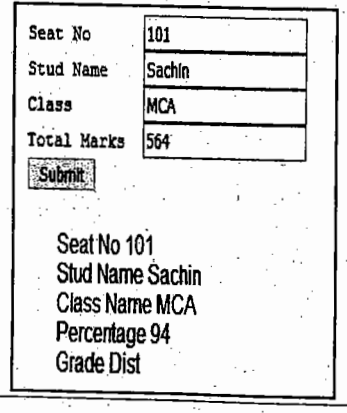
```
<web-app>
   <servlet>
      <servlet-name>Ass15B</servlet-name>
      <servlet-class>Mark</servlet-class>
 </servlet>
   <servlet-mapping>
      <servlet-name>Ass15B</servlet-name>
 <url-pattern>/Mark</url-pattern>
   </servlet-mapping>
</web-app>
```

Mark.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Mark extends HttpServlet
{
   public void doGet(HttpServletRequest req,HttpServletResponse res) throws
   IOException,ServletException
   {
      res.setContentType("text/html");
      PrintWriter pw=res.getWriter();
```

```
        String sno=req.getParameter("t1");
        String sn=req.getParameter("t2");
        String cn=req.getParameter("t3");
        int tm=Integer.parseInt(req.getParameter("t4"));

        int p=tm/6;
        String gd;

        if(p>=70)
        {
           gd="Dist";
        }
        else if(p>=60 && p<70)
        {
           gd="First";
        }
        else if(p>=50 && p<60)
        {
           gd="Second";
        }
        else if(p>=40 && p<50)
        {
           gd="Pass";
        }
        else
        gd="Fail";

        pw.println("Seat No " + sno);
        pw.println("Stud Name " + sn);
        pw.println("Class Name " + cn);
        pw.println("Percentage " + p);
        pw.println("Grade " + gd);
    }
}
```

Output:



Summary

- CGI stands for common gateway interface used to process the requests of the clients sequentially.
- If number of clients increases then it takes more time for the processing the requests. To overcome this Servlet is used.
- Servlet is a java program executed on tomcat web server.
- There are two types of Servlet: GenericServlet and HttpServlet, the main difference between them is GenericServlet is protocol independent and HttpServlet is protocol dependent.
- For the creation of user defined Servlet either GenericServlet or HttpServlet class has to inherit into it.
- Servlet processes the request of the clients randomly.
- It first loads into the memory, creates the threads for the processing the requests of the clients.
- If number of clients increases then it creates number of threads and all that threads are residing into the memory so it takes more memory space and more time for the execution.
- To overcome this JSP is introduced. It a java server pages used to developed dynamic as well as static website.
- There is no need to compile JSP application it gets compiled implicitly by the JSP engine.
- JSP application always written inside its delimiters (<% ..... %>)
- JSP has lot of built in components by using it we can develop different types of dynamic websites.

Check Your Understanding

1. Which option is true about session scope?
   (a) Objects are accessible only from the page in which they are created
   (b) Objects are accessible only from the pages which are in same session
   (c) Objects are accessible only from the pages which are processing the same request
   (d) Objects are accessible only from the pages which reside in same application
2. Which of the scripting of JSP not putting content into service method of the converted servlet?
   (a) Declarations (b) Scriptlets
   (c) Expressions (d) None of the above

3. Why is XML a good way to transfer text-based data from one program or tool to another?
   (a) XML imposes important limitations on the receiving program or tool
   (b) The receiving program or tool can use the XML tagging information to determine how to best handle the incoming data
   (c) XML tags offer an extra level of security
   (d) XML tags specify to the receiving program or tool exactly how to format and display the data
4. The difference between Servlets and JSP is the _____.
   (a) translation (b) compilation
   (c) syntax (d) Both A and B
5. Which of the following are the valid scopes in JSP?
   (a) request, page, session, application
   (b) request, page, session, global
   (c) response, page, session, application
   (d) request, page, context, application
6. JSP includes a mechanism for defining _____ or custom tags.
   (a) static attributes (b) local attributes
   (c) dynamic attributes (d) global attributes
7. How many jsp implicit objects are there and these objects are created by the web container that are available to all the jsp pages?
   (a) 8 (b) 9
   (c) 10 (d) 7
8. Why use Request Dispatcher to forward a request to another resource, instead of using a sendRedirect?
   (a) Redirects are no longer supported in the current servlet API
   (b) Redirects are not a cross-platform portable mechanism
   (c) TheRequestDispatcher does not use the reflection API
   (d) TheRequestDispatcher does not require a round trip to the client, and thus is more efficient and allows the server to maintain request state
9. Which is not a directive?
   (a) include (b) page
   (c) export (d) useBean
10. The Java _____ specification defines an application programming interface for communication between the Web server and the application program.
    (a) Servlet (b) Server
    (c) Program (d) Randomize

11. The doGet() method in the example extracts values of the parameter's type and number by using _____.
    (a) request.getParameter() (b) request.setParameter()
    (c) responce.getParameter() (d) responce.getAttribute()

Answers

1. (b)	2. (c)	3. (b)	4. (c)	5. (a)	6. (c)	7. (b)	8. (d)	9. (c)	10. (a)	11. (a)

Practice Questions

Q.I Answer the following Questions in short:

1. State the role of service () method?
2. List the methods used to store and obtain information from aHttpSession Object?
3. Explain use of scriptlets?
4. "Http is a stateless protocol" comment.
5. What is the purpose of getsession() method?
6. Write the syntax of comments in JSP.
7. Explain Servletconfig() method.
8. What is the use of web.xml

Q.II Answer the following Questions:

1. What is the web application and what is the difference between Get and Post request?
2. What information is received by the web server if we request for a Servlet?
3. What are the ways for servlet collaboration and what is the difference between RequestDispatcher and sendRedirect() method?
4. What is the difference between ServletConfig and ServletContext interface?
5. How many ways can we maintain the state of a user? Which approach is mostly used in web development?
6. How to count the total number of visitors and whole response time for a request?
7. How to run servlet?
8. Explain JSP features.
9. Explain Servlet Life cycle with suitable example?
10. Explain the Different types of tags in JSP.
11. Write a note on JSP Directives.
12. Explain JSP Architecture in detail.

Q.III Write short note on:

1. Web application
2. Servlet
3. Session
4. JSP Life Cycle
5. JSP Directives

■■■