T.Y.B.C.A. SCIENCE : SEMESTER-V

NEW SYLLABUS
CBCS PATTERN

PROGRAMMING IN JAVA

Dr. Ms. MANISHA BHARAMBE
MANISHA SURYAWANSHI
KAMIL AJMAL KHAN

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

SPPU New Syllabus

A Book Of

Programming in Java

For B.C.A. (Science) : Semester - V

[Course Code BCA - 351 : Credits - 04]

CBCS Pattern

As Per New Syllabus, Effective from June 2021

Dr. Ms. Manisha Bharambe
M.Sc. (Comp. Sci.), M.Phil, Ph.D. (Comp. Sci.)
Vice Principal and BCA Coordinator
MES Abasaheb Garware College
Pune 4

Mrs. Manisha Suryawanshi
M.C.S., M.Phil. (CS),
M.B.A.(HR)
Asst. Prof., Placement Co-ordinator
Modern College of Arts, Science and Commerce
(Autonomous)
Pune-5

Mr. Kamil Ajmal Khan
M.Sc. Computer Science
Asst. Prof. Computer Science Department
Abeda Inamdar Senior College
Pune

Price ₹ 350/-

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N5871

Programming in Java **ISBN 978-93-5451-113-4**

First Edition : August 2021

© : Authors

The text of this publication, or any part thereof, should not be reproduced or transmitted in any form or stored in any computer storage system or device for distribution including photocopy, recording, taping or information retrieval system or reproduced on any disc, tape, perforated media or other information storage device etc., without the written permission of Authors with whom the rights are reserved. Breach of this condition is liable for legal action.

Every effort has been made to avoid errors or omissions in this publication. In spite of this, errors may have crept in. Any mistake, error or discrepancy so noted and shall be brought to our notice shall be taken care of in the next edition. It is notified that neither the publisher nor the authors or seller shall be responsible for any damage or loss of action to any one, of any kind, in any manner, therefrom.

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

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

MUMBAI

Nirali Prakashan : 385, S.V.P. Road, Rasdhara Co-op. Hsg. Society Ltd.
Girgaum, Mumbai 400004, Maharashtra; Mobile : 9320129587
Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

➢ DISTRIBUTION BRANCHES

JALGAON

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

KOLHAPUR

Nirali Prakashan : New Mahadvar Road, Kedar Plaza, 1st Floor Opp. IDBI Bank, Kolhapur 416 012
Maharashtra. Mob : 9850046155; Email : niralikolhapur@pragationline.com

NAGPUR

Nirali Prakashan : Above Maratha Mandir, Shop No. 3, First Floor
Rani Jhanshi Square, Sitabuldi, Nagpur 440012, Maharashtra
Tel : (0712) 254 7129; Email : niralinagpur@pragationline.com

DELHI

Nirali Prakashan : 4593/15, Basement, Agarwal Lane, Ansari Road, Daryaganj
Near Times of India Building, New Delhi 110002 Mob : 08505972553
Email : niralidelhi@pragationline.com

BENGALURU

Nirali Prakashan : Maitri Ground Floor, Jaya Apartments, No. 99, 6th Cross, 6th Main
Malleswaram, Bengaluru 560003, Karnataka; Mob : 9449043034
Email: niralibangalore@pragationline.com

Other Branches : Hyderabad, Chennai

Note : Every possible effort has been made to avoid errors or omissions in this book. In spite this, errors may have crept in. Any type of error or mistake so noted, and shall be brought to our notice, shall be taken care of in the next edition. It is notified that neither the publisher, nor the author or book seller shall be responsible for any damage or loss of action to any one of any kind, in any manner, therefrom. The reader must cross check all the facts and contents with original Government notification or publications.

niralipune@pragationline.com | www.pragationline.com

Also find us on www.facebook.com/niralibooks

Preface ...

We take an opportunity to present this book entitled as **"Programming in Java"** to the students of **B.C.A.(Science), Semester - V** as per the New Syllabus, (CBCS Pattern 2019).

The book covers theory of Introducing Java, Classes and Objects, Inheritance and Interface, Collections, Exception Handling and I/O, Swing, Database Programming, Servlets & JSP.

A special words of thank to Shri. Dineshbhai Furia, Mr. Jignesh Furia for showing full faith in us to write this text book. We also thank to Mr. Akbar Shaikh of M/s Nirali Prakashan for their excellent co-operation.

We also thank Ms. Chaitali Takale, Mr. Ravindra Walodare, Mr. Sachin Shinde, Mr. Ashok Bodke, Mr. Moshin Sayyed and Mr. Nitin Thorat.

Although every care has been taken to check mistakes and misprints, any errors, omission and suggestions from teachers and students for the improvement of this text book shall be most welcome.

Authors

■■■

Syllabus ...

1. Introducing Java, Classes and Objects **[10 Hrs.]**

Introducing Java

1.1 A Short History of Java
1.2 Features of Java
1.3 Java Environment - Compiler, Interpreter, JVM
1.4 Simple Java Program
1.5 Types of Comments
1.6 Declaring Single and Multi-Dimensional Arrays
1.7 Accepting Input using Command Line Arguments
1.8 Accepting Input from console (Using BufferedReader and Scanner class)

Classes and Objects

1.9 Defining Your Own Classes
1.10 Access Specifiers (public, protected, private, default)
1.11 Array of Objects
1.12 Constructor, Overloading Constructors and use of "this" Keyword
1.13 Static Blocks, Static Fields and Static Methods
1.14 Predefined Classes – Object Class Methods (equals(), toString(),hashcode())
1.15 Garbage Collection (finalize() Method)

2. Inheritance and Interface **[10 Hrs.]**

Inheritance

2.1 Inheritance Basics (extends Keyword) and Types of Inheritance
2.2 Superclass, Subclass and use of Super Keyword
2.3 Method Overriding and Runtime Polymorphism
2.4 Use of Final Keyword related to Variable, Method and Class
2.5 Use of Abstract Class and Abstract Methods

Interface

2.6 Defining and Implementing Interfaces
2.7 Runtime Polymorphism Using Interface

Packages

2.8 Creating, Accessing and using Packages

3. Collections, Exception Handling and I/O **[10 Hrs.]**

Collections

3.1 Wrapper Classes
3.2 Introduction to the Collection framework
3.3 List - ArrayList, LinkedList and Vector
3.4 Set - HashSet, TreeSet, and LinkedHashSet
3.5 Map - HashMap, LinkedHashMap, Hashtable and TreeMap
3.6 Interfaces such as Iterators, ListIterators, Enumerations

Syllabus ...

1. Introducing Java, Classes and Objects **[10 Hrs.]**

Introducing Java

1.1 A Short History of Java
1.2 Features of Java
1.3 Java Environment - Compiler, Interpreter, JVM
1.4 Simple Java Program
1.5 Types of Comments
1.6 Declaring Single and Multi-Dimensional Arrays
1.7 Accepting Input using Command Line Arguments
1.8 Accepting Input from console (Using BufferedReader and Scanner class)

Classes and Objects

1.9 Defining Your Own Classes
1.10 Access Specifiers (public, protected, private, default)
1.11 Array of Objects
1.12 Constructor, Overloading Constructors and use of "this" Keyword
1.13 Static Blocks, Static Fields and Static Methods
1.14 Predefined Classes – Object Class Methods (equals(), toString(),hashcode())
1.15 Garbage Collection (finalize() Method)

2. Inheritance and Interface **[10 Hrs.]**

Inheritance

2.1 Inheritance Basics (extends Keyword) and Types of Inheritance
2.2 Superclass, Subclass and use of Super Keyword
2.3 Method Overriding and Runtime Polymorphism
2.4 Use of Final Keyword related to Variable, Method and Class
2.5 Use of Abstract Class and Abstract Methods

Interface

2.6 Defining and Implementing Interfaces
2.7 Runtime Polymorphism Using Interface

Packages

2.8 Creating, Accessing and using Packages

3. Collections, Exception Handling and I/O **[10 Hrs.]**

Collections

3.1 Wrapper Classes
3.2 Introduction to the Collection framework
3.3 List - ArrayList, LinkedList and Vector
3.4 Set - HashSet, TreeSet, and LinkedHashSet
3.5 Map - HashMap, LinkedHashMap, Hashtable and TreeMap
3.6 Interfaces such as Iterators, ListIterators, Enumerations

Exception Handling

3.7 Exception Class, Checked and Unchecked Exception

3.8 Catching Exception and Exception Handling – try, catch, finally, throw and throws, multiple catch block

3.9 Creating user Defined Exception

I/O

3.10 String class(basic methods), String Buffer Class

3.11 File Class

3.12 DataInputStream and DataOutputStream Class

4. Swing **[10 Hrs.]**

4.1 What is Swing?

4.2 The MVC Architecture and Swing

4.3 Layout Manager and Layouts, The JComponent Class

4.4 Components - JLabel, JButton, JText, JTextArea, JCheckBox, JRadioButton, JList, JComboBox, JMenu and JPopupMenu Class, JMenuItem

4.5 Dialogs (Message, confirmation, input), JFileChooser

4.6 Event Handling: Event sources, Listeners - ActionListener, ItemListener

4.7 Mouse and Keyboard Event Handling

4.8 Adapters - MouseAdapter, KeyAdapter

4.9 Anonymous Inner Class

5. Database Programming **[10 Hrs.]**

5.1 The Role of jdbc, jdbc Configuration

5.2 Types of Drivers

5.3 Connectivity with Database

5.4 JDBC Statements - Statement, PreparedStatement, CallableStatement

5.5 Scrollable and Updatable Result Sets

- TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, TYPE_SCROLL_SENSITIVE
- CONCUR_READ_ONLY, CONCUR_UPDATABLE

5.6 Metadata - DatabaseMetadata, ResultSetMetadata(Database : PostgreSQL)

6. Servlets & JSP **[10 Hrs.]**

Servlets

6.1 Introduction to Servlets and Hierarchy of Servlets

6.2 Life Cycle of a Servlet

6.3 Tomcat Configuration (Note: Only for Lab Demonstration)

6.4 Handing Get and Post Request (HTTP)

6.5 Handling a Data from HTML to a Servlet

6.6 Session Tracking - Cookies and Http Session

JSP

6.7 Simple JSP Program

6.8 Life Cycle of a JSP

6.9 Implicit Objects

6.10 Scripting Elements – Declarations, Expressions, Scriplets, Comments

6.11 JSP Directives – Page Directive, Include Directive

6.12 Mixing Scriplets and HTML

■■■

Contents ...

1.	Introducing Java, Classes and Objects	1.1 – 1.58
2.	Inheritance and Interface	2.1 – 2.50
3.	Collections, Exception Handling and I/O	3.1 – 3.58
4.	Swing	4.1 – 4.58
5.	Database Programming	5.1 – 5.42
6.	Servlets & JSP	6.1 – 6.64
	Bibliography	B.1 – B.1

■■■

1... Introducing Java, Classes and Objects

Learning Objectives...

- To understand basic concepts of Java.
- To learn Java Environment.
- To study Java 1D, 2D array.
- To understand basic concepts of Objects and Classes.
- To study Constructors and Access Specifiers.
- To learn Predefined Classes in Java.
- To learn Garbage Collection.

1.1 INTRODUCING JAVA

- Java is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs.
- Java is used for Mobile applications, Desktop applications, Web applications, Web servers and Application servers, Games, Database connection, etc.
- Java works on different platforms like Windows, Mac, Linux, Raspberry Pi, etc.
- Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission.
- Although Java has become inseparably linked with the online environment of the internet, it is important to remember that Java is first and foremost a programming language.
- Computer language innovation and development occurs for two fundamental reasons:
  (i) To adapt to changing environments and uses.
  (ii) To implement refinements and improvements in the art of programming.
- Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++.

- Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades.
- In an OOP environment, data is considered as a critical element and is tied to functions that operate on it. A set of data that is encapsulated in an object is known as the member data and the functions that operates on these data are known as the member functions.
- Java is a general-purpose computer programming language which enables computer programmers to write computer instructions using English based commands, instead of having to write in numeric codes.

1.2 A SHORT HISTORY OF JAVA

- Java was developed at Sun Microsystems (which has since been acquired by Oracle) in 1991, by a team of James Gosling, Patrick Naughton, Chris Wrath, Ed Frank and Mike Sheridan as its members. The language was initially called Oak. It was later termed as Java.
- Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lind holm were key contributors to the maturing of the original prototype.
- Java is a general-purpose, Object-Oriented Programming (OOP) language. James Gosling is called the father of Java Programming.
- Java is an Object-Oriented Programming language means Java enables us not only to organize our program code into logical unit called objects but also to take advantage of encapsulation, inheritance, polymorphism and so on.
- As of March 2021, the latest version is Java 16, with Java 11, a currently supported long-term support (LTS) version, released on September 25, 2018. Oracle released the last zero-cost public update for the legacy version Java 8 LTS in January 2019 for commercial use, although it will otherwise still support Java 8 with public updates for personal use indefinitely. Other vendors have begun to offer zero-cost builds of OpenJDK 8 and 11 that are still receiving security and other upgrades.
- Oracle (and others) highly recommends uninstalling outdated versions of Java because of serious risks due to unresolved security issues. Since Java 9, 10, 12, 13, 14, and 15 are no longer supported, Oracle advises its users to immediately transition to the latest version (currently Java 16) or an LTS release.
- An application is a type of software that allows user to perform specific task. Sun Microsystems Inc. has divided Java into following parts:
  1. **Java SE (Java Standard Edition):** It contains basic core Java classes and used to develop standard applets and applications.

2. **J2EE (Java Enterprise Edition):** It contains classes that are beyond Java SE and used to providing business solutions on a network.
3. **J2ME (Java Micro Edition):** Java ME is for developers who develop code for portable devices like a PDA, a cellular phone and so on.

1.3 FEATURES OF JAVA [W-18]

- The fundamental forces that needed the invention of Java are portability and security, other factors also played an important role in molding the final form of the language.
- The key considerations were summed up by the Java team in the following list of features:

1. **Simple:** Java is a simple language. There are various concepts that make Java a simple language. Java is designed to be easy to learn. Programs in Java are easy to write and debug because Java does not use the pointers, preprocessor header files, operator overloading etc.
2. **Secure:** Java is aimed to be used in networked or distributed environments. Java does not allocate direct pointer to memory, this makes it impossible to accidentally reference memory that belongs to other program.
3. **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Java code is portable. It was an important design goal of Java that it be portable so that as new architectures (due to hardware, operating system, or both) are developed, the Java environment could be ported to them.
4. **Object-oriented:** Java is a true object oriented language. Almost everything in Java is an object. Java has a wide variety of classes. These classes are organized in various packages. We can use them in our programs while implementing inheritance. The object model in Java is simple and easy to extend.
5. **Robust:** Java is a robust language because it provides many safeguards to ensure reliable code. Java does not allow pointer which avoids security problem. There is no concept of reference variable. This eliminates the possibility of overwriting memory and corrupting data. Automatic garbage collection eliminates memory leaks.
6. **Multithreaded:** Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multi-threaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multi-process synchronization that enables you to construct smoothly running interactive systems.
7. **Architecture-neutral:** A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that

no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. Their goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished.

8. **Interpreted:** As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. Most previous attempts at cross-platform solutions have done so at the expense of performance.
9. **High Performance:** As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a Just-in-Time compiler.
10. **Distributed:** Java facilitates the building of distributed application by a collection of classes for use in networked applications. Java is designed for the distributed environment of the Internet.
11. **Dynamic:** Java language is capable of dynamically linking in new class libraries, methods and objects. Because Java is interpreted, it is an extremely dynamic language. At runtime, the Java environment can extend itself by linking in classes that may be located on remote servers on a network, (for example; the Internet).

1.4 JAVA ENVIRONMENT - COMPILER, INTERPRETER, JVM

- Java environment includes a large number of development tools [part of system known as JDK (Java Development Kit)].

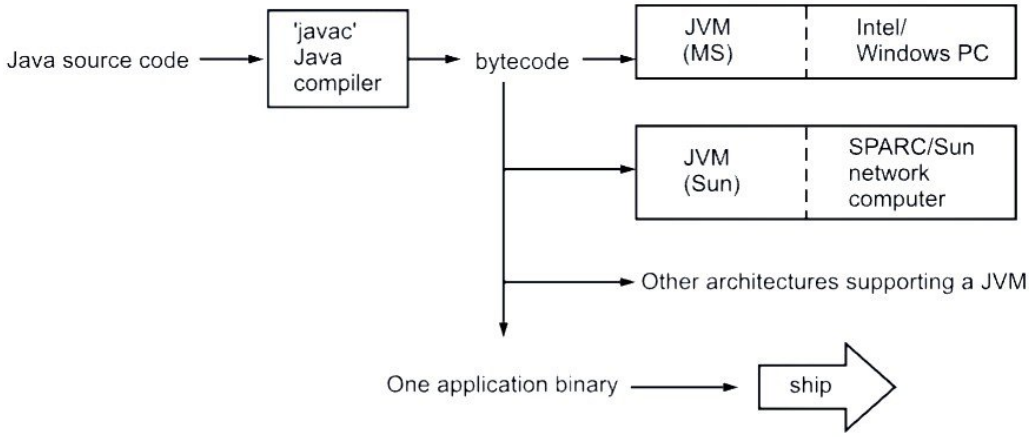


Fig. 1.1: Java Environment

- The Java source file is stored as a '.java' file and is compiled to get an executable '.class' file.
- A Class Loader is used to download the executable file.
- The so-called 'native code' that the processor understands is never stored, and has to be generated from the .class file each time. Using a 'Just-In-Time (JIT)' compiler with the interpreter.

- The JIT compiler enables all the frequently used class files to be compiled into native code just once, rather than each time they are used. This improves the speed of execution of the commands to an acceptable level. The stored native code last only for the lifetime of the application.
- The API provided by the language is constantly available to the JVM, and the entire package, excepting the compiler that generates the .class file from the .java file, is referred to as the Java Runtime Environment (JRE).

1.4.1 Compiler(JIT)

- JIT (Just-in-Time) compiler is a module inside the JVM which helps in compiling certain parts of byte code into the machine code for higher performance.
- JVM translates bytecode into machine language as they are called Just-in-Time compilation (JIT). The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time.
- Java programs consist of classes, which contain platform-neutral bytecode that can be interpreted by a JVM on many different computer architectures.
- At run-time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application.
- The JIT compiler helps improve the performance of Java programs by compiling bytecode into native machine code at run time.

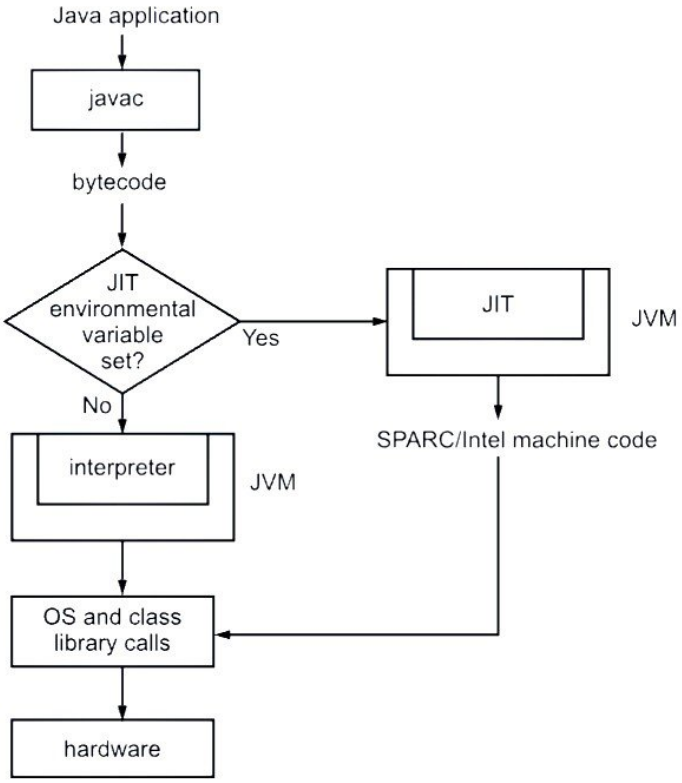


Fig. 1.2: JIT Compile Process

- When the JIT compiler environment variable is on (the default), the JVM reads the .class file for interpretation and passes it to the JIT compiler. The JIT compiler then compiles the bytecodes into native code for the platform on which it is running.
- The JIT compiler is enabled by default and is activated when a Java method is called. The JIT compiler compiles the bytecode of that method into native machine code, compiling it "just in time" to run.

1.4.2 Interpreter (JRE)

- Java Runtime Environment (JRE) facilitates the execution of programs in Java. The Java Runtime Environment is a set of software tools which are used for developing java applications. It is used to provide runtime environment.
- It mainly includes the following:
  1. Java Virtual Machine.
  2. Runtime Class Libraries.
  3. User Interface Toolkits.
  4. Deployment Technologies.
- Java language has been designed to work in a platform-independent manner, means that the result is independent of the hardware used, the CPU executing the command or the operating system supporting the system.
- In order to achieve this dream concept of Write Once and Run Anywhere (WORA), we have a Java Runtime Environment (JRE) running in the machine on top of the operating system.

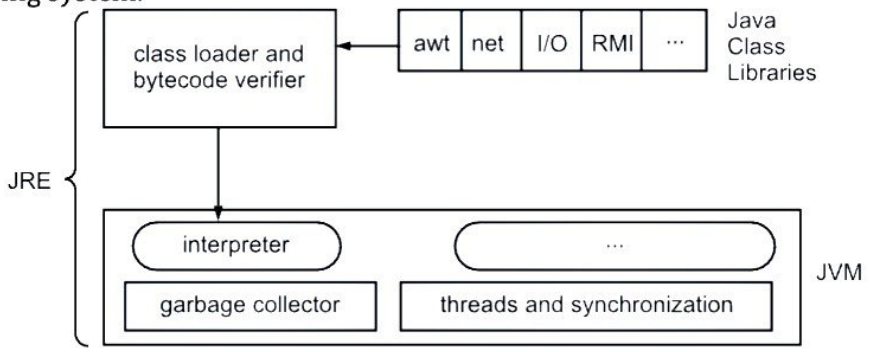


Fig. 1.3: Java Runtime Environment (JRE)

- As illustrated in Fig. 1.3 the JRE has following two major components:
  1. Java Base Class is the implementation of the Java APIs used in the program. JRE effectively uses these implements to achieve the results.
  2. The Java Virtual Machine (JVM) provides a platform-independent way of executing code by abstracting the differences between operating systems and CPU architectures.

1.4.3 JVM

- Java Virtual Machine (JVM) is a virtual machine that provides runtime environment to execute java bytecode.

- When writing Java applications, the developer does not need to directly call Windows (or other operating systems) library functions. The Java compiler does not write native instructions, instead it generates bytecodes for a virtual machine called Java Virtual Machine (JVM).
- In other words, it is a virtual machine that works like a CPU of a computer.

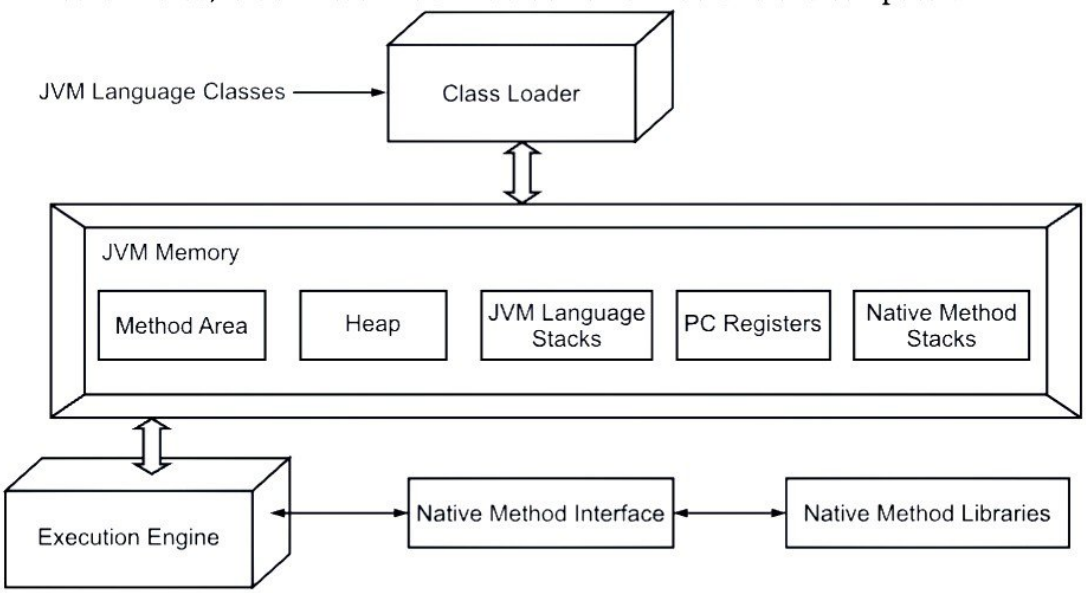


Fig. 1.4: Architecture of JVM

- JVM architecture contains the following components:
  1. **Class Loader:** Class loader loads the Class for execution.
  2. **Method Area:** It stores pre-class structure as constant pool.
  3. **Heap:** Heap is in which objects are allocated.
  4. **Stack:** Local variables and partial results are store here. Each thread has a private JVM stack created when the thread is created.
  5. **Program Register:** It holds the address of JVM instruction currently being executed.
  6. **Native Method Stack:** It contains all native used in application.
  7. **Execution Engine:** Execution engine controls the execution of instructions contained in the methods of the classes.
  8. **Native Method Interface:** Native method interface gives an interface between Java code and native code during execution.
  9. **Native Method Libraries:** Native Libraries consist of files required for the execution of native code.

1.5 SIMPLE JAVA PROGRAM [W-18]

Java Program Structure:

- A Java language program may contain many classes of which main() is defined in only one class.

- Classes in Java program contain data members and methods that operate on the data members of the class. Methods in Java Program may contain datatype declarations and executable statements.
- To write a Java program, we first define classes and then put them together.

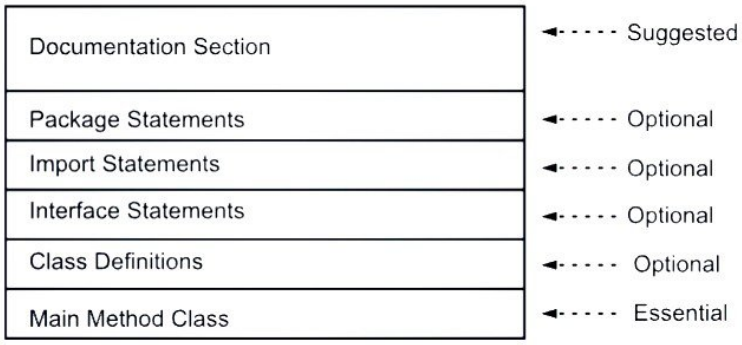


Fig. 1.5: Structure of Java Program

- A Java program structure contains following sections:
  1. **Documentation Section:** This section is a set of comment lines giving the name of the program, the author and the other details, which the programmer would like to refer.
  2. **Package Statement:** In Java files, the first statement allowed is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong to this package.

     For example: `package employee;`

     It is optional because no need and our class to be part of packages.
  3. **Import Statement:** Similar to #include statement in C.
     - For example, `import employee.test;`
     - In above example, import statement instructs the interpreter to load the test class contained in package employee.
     - Using import statement; we can access classes that part of other named packages.
  4. **Interface Statement:** An interface is like class but includes group of methods declaration. Interfaces are used when we want to implement multiple inheritance features.
  5. **Class Definition:** Java program may contain many/multiple class definitions. Classes are primary feature of Java program. The classes in Java are used to map real-world problems.
  6. **Main Method Class:** Java stand-alone program requires main method as starting point and this is essential part of program. Main method in Java program creates object of various classes and on reaching, end of the main, the program terminate and the control passes back to the operating system.

Program 1.1: First Java Program.

```
class SimpleJava
{
   public static void main(String args[])
   {
      System.out.print("My First Java Program with Kamil");
   }
}
```

Output:

```
My First Java Program with Kamil
```

Compiling the Program:

- To compile the above program, execute the compiler, `javac` specifying the name of the source file on the command line. This is shown below:

```
C:\> javac SimpleJava.java
```

- Javac compiler creates a file named 'SimpleJava.class' which contains the byte code version of the program. It is an intermediate representation of the program.

Executing the Program:

- To actually run the program, you must use the Java application launcher called '`java`'. This is shown below:

```
C:\> java SimpleJava
```

Explanation:

- **Public:** It is an access specifier. We should use a public keyword before the main() method so that, JVM can identify the execution point of the program. If we use private, protected, and default before the main() method, it will not be visible to JVM.
- **Static:** You can make a method static by using the keyword static. We should call the main() method without creating an object. Static methods are the method which invokes without creating the objects, so we do not need any object to call the main() method.
- **Void:** In Java, every method has the return type. Void keyword acknowledges the compiler that main() method does not return any value.
- **Main():** It is a default signature which is predefined in the JVM. It is called by JVM to execute a program line by line and end the execution after completion of this method. We can also overload the main() method.
- **String args[]:** The main() method also accepts some data from the user. It accepts a group of strings, which is called a string array. It is used to hold the command line arguments in the form of string values.

1.6 TYPES OF COMMENTS

- Comments are non-executable statements and are ignored by the Java compiler. The comments increases the readability of the programs.

- Java language provides three styles of comments as given below:

1. **Single Line Comments:**
   - It starts with two forward slashes (//) and continue to the end of the current line. Line comments do not require an ending symbol.
   - Java's single line comments are proved useful for supplying short explanations for variables, function declarations, and expressions.

     Example: `//This is Single Line Comment`
2. **Block/Multiline Comments:**
   - Multiline comments start with /* and ends with */. Any text between /* and */ will be ignored by Java compiler.
   - Java's multiline comments are useful when the comment text does not fit into one line; therefore needs to span across lines.

     Example: `/*This is Multi Line Comment */`
3. **Javadoc/Documentation Comments:**
   - Javadoc comments are used to document the new classes you create as a programmer i.e. provides information about the class for users of the class to read.
   - Java documentation comment is given by /**.....*/.

     Example: `/** Welcome to Nirali Prakashan */`

Program 1.2: Program for Java comment.

```
/**
   This is a Documentation Comment.
*/
public class CommentsDemoExample
{
   public static void main(String args[])
   {
      /* This is a Multi-line Comment */
      System.out.println("Hello World");// This is a Single Live Comment.
   }
}
```

Output:

```
Hello World
```

1.7 DECLARING SINGLE AND MULTI-DIMENSIONAL ARRAYS

- An array is a group of similar type variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

- Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
- Array is a contiguous fixed-length structure for storing multiple values of the same type. Array in Java is index based, first element of the array is stored at 0 index.

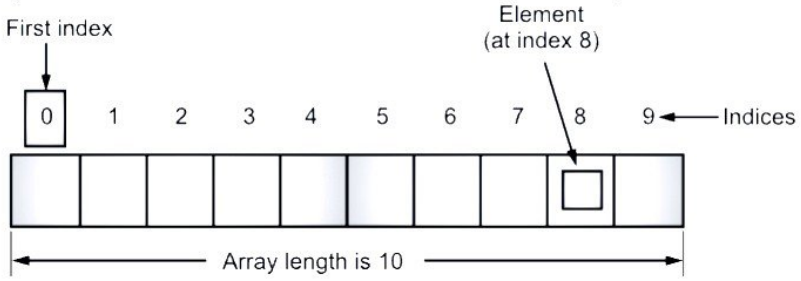


Fig. 1.6: Representation of Array

1.7.1 Single-Dimensional (1D) Array

- Array having only one subscript variable is called One-Dimensional array.
- It is also called as Single-Dimensional Array or Linear Array.
- A one-dimensional array is, essentially, a list of similar type variables. To create an array, you first must create an array variable of the desired type.
- To declare an array, you specify the name of the array and the data type, as you would for any other variable. Adding an empty set of brackets ([ ]) indicates that the variable is an array.
- The general **form/syntax** of a one-dimensional array declaration is given below:

```
datatype array_name [];
```

- Here, datatype declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold.
- **For example**, the following declares an array named month_days with the type "array of int":

```
int month_days[];
```

Instantiating Arrays:

- As we mentioned earlier, Java arrays are objects, so to allocate memory for an array, you need to instantiate the array using the new keyword. Here, is the **syntax** for instantiating an array:

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

Where, size is an expression that evaluates to an integer and specifies the number of elements in the array.

For example: `month_days = new int[12];`

Assigning Initial Values to Arrays:

- Java allows you to instantiate an array by assigning initial values when the array is declared. To do this, you specify the initial values using a comma-separated list within curly braces as given below:

  ```
  datatype array_name[] = {value0, value1, value2, ... valueN};
  ```

  Where, valueN is an expression that evaluates to the data type of the array and is the value to assign to the element at index N.
- Note that, we do not use the new keyword and we do not specify a size for the array. The number of elements in the array is determined by the number of values in the initialization list.
- For example, following statement declares and instantiates an array of Numbers:

  ```
  int nine = 9;
  int[ ] Numbers = {2, 4, 3, 8, one, one + 2, 13, 14, 17, 18};
  ```
- Because 10 values are given in the initialization list, this array has 10 elements, Notice that the values can be an expression, for example, one and one + 2.

Program 1.3: Program for one-dimensional (1D) array.

```
import java.util.*;
class arraydemo
{
   public static void main(String args[])
   {
      int a[]={2,4,6,3,1};
      System.out.println("Number of elements in array a:"+a.length);
      System.out.println("Elements in array a:" + a.length);
      for(int i=0;i <a.length;i++)
      System.out.print (a[i] + "\t");
   }
}
```

Output:

```
Number of elements in array a: 5
Elements in array a: 2      4      6      3      1
```

Program 1.4: Program for how to create, initialize, and process arrays.

```
public class TestArray
{
   public static void main(String[] args)
   {
      double[] myList = {1.9, 2.9, 3.4, 3.5};
      // Print all the array elements
      for (int i = 0; i < myList.length; i++)
      {
         System.out.println(myList[i] + " ");
      }
```

```
        // Summing all elements
        double total = 0;
        for (int i = 0; i < myList.length; i++)
        {
           total += myList[i];
        }
        System.out.println("Total is " + total);
        // Finding the largest element
        double max = myList[0];
        for (int i = 1; i < myList.length; i++)
        {
           if (myList[i] > max) max = myList[i];
        }
        System.out.println("Max is " + max);
     }
  }
```

Output:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Program 1.5: Program to print Odd and Even Numbers from an Array.

```
public class Exoddevenarray
{
   public static void main(String args[])
   {
      int arr[]={10,23,54,61,32,22};  //Declaration of Array
      System.out.println("Even Array Elements:");
      for(int i=0;i<arr.length;i++)
      {
         if(arr[i]%2==0)
         {
            System.out.println(arr[i]);
         }
      }
      System.out.println("Odd Array Elements:");
```

```
        for(int i=0;i<arr.length;i++)
        {
            if(arr[i]%2!=0)
            {
                System.out.println(arr[i]);
            }
        }
    }
}  //end of class
```

Output:

```
Even Array Elements:
10
54
32
22
Odd Array Elements:
23
61
```

1.7.2 Multi-Dimensional (2D) Array

- In Java, multi-dimensional arrays are actually arrays of arrays. These arrays look and act like regular multi-dimensional arrays. However, as you will see, there are a couple of subtle differences.
- To declare a multi-dimensional array variable, specify each additional index using another set of square brackets. **For example,** the following declares a two-dimensional array variable called twoD.

```
int twoD[][] = new int[4][5];
```

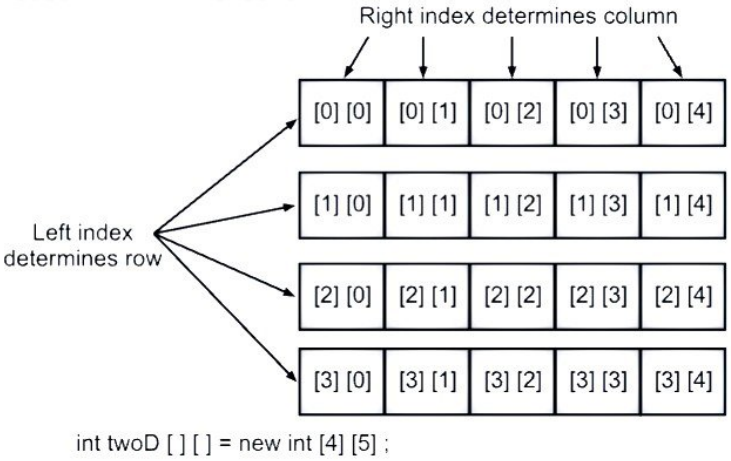


Fig. 1.7: 2D Array Declaration

- To declare a multi-dimensional array, we use the same syntax as for an array, except that we include an empty set of brackets for each dimension.

- Here is the general **syntax** for declaring a Two-Dimensional (2D) array:

```
datatype array_Name[] [];
```

- Here is the general **syntax** for declaring a Three-Dimensional (3D) array:

```
datatype array_name [] [] [];
```

Instantiating Multi-dimensional Arrays:

- Just like instantiating 1D arrays, you instantiate a multi-dimensional array using the new keyword. Here, is the **syntax** for instantiating a 2D array:

```
array_name = new datatype [expl][exp2];
```

  Where, exp1 and exp2 are expressions that evaluate to integers and specify the number of rows and the number of columns in the array respectively.
- Above statement allocates memory for the array. The number of elements in a two-dimensional array is equal to the sum of the number of elements in each row.
- When all the rows have the same number of columns, the number of elements in the array is equal to the number of rows multiplied by the number of columns.

Program 1.6: Program for Two-Dimensional (2D) array.

```
class TwoDArray
{
public static void main(String args[])
{
   int twoD[][]= new int[4][5];
   int i, j, k = 0;
   for(i=0; i<4; i++)
   for(j=0; j<5; j++)
   {
      twoD[i][j] = k;
      k++;
   }
   for(i=0; i<4; i++)
   {
      for(j=0; j<5; j++)
      System.out.print(twoD[i][j] + " ");
      System.out.println();
   }
}
}
```

Output:

```
0  1  2  3  4
5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
```

Program 1.7: Program to read and display 2D Array.

```
class Example2d
{
  public static void main(String[] args)
  {
    int[][] arr = { { 1, 2, 3 }, { 4, 5, 6 },{7, 8, 9 } };
    for (int i = 0; i < 3; i++)
    {
      for (int j = 0; j < 3; j++)
      {
          System.out.println("arr[" + i + "][" + j + "] = " + arr[i][j]);
      }//j
    }//i
    System.out.println("Array Elements :");
    for (int i = 0; i < 3; i++)
    {
      for (int j = 0; j < 3; j++)
      {
        System.out.print(" " + arr[i][j]);
      }//j
      System.out.println();
     }//i
   }
}
```

Output:

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 4
arr[1][1] = 5
arr[1][2] = 6
arr[2][0] = 7
arr[2][1] = 8
arr[2][2] = 9
Array Elements:
1 2 3
4 5 6
7 8 9
```

Program 1.8: Program Demonstrate a three-dimensional array.

```
class ThreeDMatrix
{
   public static void main(String args[])
   {
      int threeD[][][] = new int[3][4][5];
      int i, j, k;
      for(i=0; i<3; i++)
      for(j=0; j<4; j++)
      for(k=0; k<5; k++)
      threeD[i][j][k] = i * j * k;
      for(i=0; i<3; i++)
      {
         for(j=0; j<4; j++)
         {
            for(k=0; k<5; k++)
            System.out.print(threeD[i][j][k] + " ");
            System.out.println();
         }
         System.out.println();
      }
   }
}
```

Output:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Program 1.9: Program for 2D array.

```
class twodim
{
   final static int r = 3;
   final static int c = 3;
   public static void main(String[] args)
   {
      int mult[] [] = new int[r][c];
      int row, column,
      int i, j;
```

```
        for (i=1; i<=r; i++)
        {
            for (j=1; j<=c; j++)
            {
                mult [i][j] = i * j;
                System.out.println(" " + mult [i][j]);
            }
        }
        System.out.println("  ");
    }
}
```

Output:

```
The given matrix:
1 2 3
2 4 6
3 6 9
```

Program 1.10: Program addition of two matrices.

```
class Testarray
{
    public static void main(String args[])
    {
        //creating two matrices
        int a[][]={{1,3,4},{3,4,5}};
        int b[][]={{1,3,4},{3,4,5}};
        //creating another matrix to store the sum of two matrices
        int c[][]=new int[2][3];
        //adding and printing addition of Two matrices
        for(int i=0;i<2;i++)
        {
            for(int j=0;j<3;j++)
            {
                c[i][j]=a[i][j]+b[i][j];
                System.out.print(c[i][j]+" ");
            }
        System.out.println();//new line
        }
    }
}
```

Output:

```
2   6   8
6   8   10
```

Program 1.11: Program for sorting elements in array.

```
import java.util.Arrays;
class SortIntArrayExample
{
   public static void main(String[] args)
   {
      //create an int array
      int[] i1 = new int[]{3,2,5,4,1};
      //print original int array
      System.out.print("Original Array:");
      for(int index=0; index < i1.length ; index++)
      System.out.print(" " + i1[index]);
      Arrays.sort(i1);
      //print sorted int array
      System.out.print("Sorted int array:");
      for(int index=0; index < i1.length ; index++)
      System.out.print(" " + i1[index]);
      int[] i2 = new int[]{5,2,3,1,4};
       Arrays.sort(i2,1,4);
      //print sorted int array
      System.out.print("Partially Sorted int array:");
      for(int index=0; index < i2.length ; index++)
      System.out.print(" " + i2[index]);
   }
}
```

Output:

```
Original Array: 3 2 5 4 1
Sorted int array: 1 2 3 4 5
Partially Sorted int array: 5 1 2 3 4
```

Program 1.12: Program to reverse elements in array.

```
import java.util.*;
class ReverseArray
{
   public static void main(String args[])
   {
      // Create java.util.Scanner object for taking input
      Scanner s=new Scanner(System.in);
```

```
        // Take no.of elements and store it in n
        System.out.println("Enter the no.of elements:");
        int n=s.nextInt();
        // Create array of size n
        int a[]=new int[n];
        // Read elements into the array
        System.out.println("Enter the elements into the array:");
        for(int i=0;i<n;i++)
        {
            a[i]=s.nextInt();
        }
        // Reverse elements in the array
        reverse(a);
        // Print the array
        for(int i=0;i<n;i++)
        {
            System.out.printf("a[%d]=%d\n",i,a[i]);
        }
    }
    public static void reverse(int[] a)
    {
        // Loop for length/2 times only else re-swapping takes place
        for(int i=0;i<a.length/2;i++)
        {
            int temp=a[i];
            a[i]=a[(a.length-1)-i];
            a[(a.length-1)-i]=temp;
        }
    }
}
```

Output:

```
Enter the no.of elements: 4
Enter the elements into the array: 1 2 3 4
a[0]=4
a[1]=3
a[2]=2
a[3]=1
```

Program 1.13: Program for transpose matrix.

```
import java.util.Scanner;
class TransposeaMatrix
{
   public static void main(String args[])
   {
      int m, n, c, d;
      Scanner in = new Scanner(System.in);
      System.out.println("Enter the number of rows and columns of matrix:");
      m = in.nextInt();
      n = in.nextInt();
      int matrix[][] = new int[m][n];
      System.out.println("Enter the elements of matrix:");
      for (c = 0 ; c < m ; c++)
      for (d = 0 ; d < n ; d++)
      matrix[c][d] = in.nextInt();
      int transpose[][] = new int[n][m];
      for (c = 0 ; c < m ; c++)
      {
         for (d = 0 ; d < n ; d++)
         transpose[d][c] = matrix[c][d];
      }
      System.out.println("Transpose of entered matrix:-");
      for (c = 0 ; c < n ; c++)
      {
         for (d = 0 ; d < m ; d++)System.out.print(transpose[c][d]+"\t");
         System.out.print("\n");
      }
   }
}
```

Output:

```
Enter the number of rows and columns of matrix:2  3
Enter the elements of matrix:
1  2  3
4  5  5
Transpose of entered matrix:
1    4
2    5
3    5
```

1.8 ACCEPTING INPUT USING COMMAND LINE ARGUMENTS [S-19]

- Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command line arguments to main( ).
- A command line argument is the information that directly follows the program name on the command line when it is executed.
- A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.
- To access the command line arguments inside a Java program is quite easy – they are stored as strings in a string array passed to the args parameter of main( ). The first command line argument is stored at args[0], the second at args[1], and so on.
- The following program displays all of the command line arguments that it is called with:

```
public class CommandLine
{
    public static void main(String args[])
    {
        for(int i = 0; i<args.length; i++)
        {
            System.out.println("args[" + i + "]:" + args[i]);
        }
    }
}
```

- Try executing this program as shown here:

```
$java CommandLine this is a command line 200 -100
```

- This will produce the following **output**:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
args[4]: line
args[5]: 200
args[6]: -100
```

1.9 ACCEPTING INPUT FROM CONSOLE (USING BUFFEREDREADER AND SCANNER CLASS) [W-18]

- There are many ways to get information from the user. In many cases, the user must be told that they should enter some information. This is known as prompting the user. The java.io package contains most of I/O classes.

- These are given below:
  1. **java.io.InputStream:** Stores information about the connection between an input device and the computer or program.
  2. **java.io.InputStreamReader:** Used to translate data bytes received from InputStream objects into a stream of characters.
  3. **java.io.BufferedReader:** Used to buffer (store) the input received from an inputStreamReader object.

1.9.1 BufferedReader

- BufferedReader improves performance by buffering input. The BufferedReader class does have a method called readLine that does return a line of text as type by the user.
- The contents of the stream can be read into a buffer to speed up the execution. The size of the buffer can be set explicitly, if required.
- In addition to the methods defined by the Reader class, this class has a method readLine() to read a line from the stream. It has two forms of constructors.
  1. **BufferedReader(Reader inStream):** This form creates a buffered character stream with a default buffer size.
  2. **BufferedReader(Reader inStream, int buffSize):** The size of the buffer is passed in buffSize.

Program 1.14: Program for BufferedReader.

```
import java.io.*;
public class BufferedReaderExample
{
    public static void main(String args[])throws Exception
    {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        // Accepting String value
        System.out.println("Enter your name");
        String name=br.readLine();
        System.out.println("Welcome "+name);
        // Accepting integer value
        System.out.println("Enter your Roll Number");
        int roll_no=Integer.parseInt(br.readLine());
        System.out.println("Roll number is "+roll_no);
        // Accepting Float value
        System.out.println("Enter your Percentage");
        float per=Float.parse Float(br.readLine());
        System.out.println("Percentage is "+per);
    }
}
```

Output:

```
Enter your name
kamil khan
Welcome kamil khan
Enter your Roll Number
101
Roll number is 101
Enter your Percentage
78
Percentage is 78.0
```

1.9.2 Scanner

- We have already learnt about Java BufferedReader to input data from user. In Java, the java.util.Scanner or Scanner class is one of them which allows the user to read values of various types in java.
- A Java Scanner is the fastest and the easiest way to get input from a user in Java than java.io.InputStream and java.io.InputStrearnReader.
- Scanner is the complement of Formatter. Added by JDK 5, Scanner reads formatted input and converts it into its binary form. Although it has always been possible to read formatted input, it required more effort than most programmers would prefer.
- Because of the addition of Scanner, it is now easy to read all types of numeric values, strings, and other types of data, whether it comes from a disk file, the keyboard, or another source.
- This next line creates a Scanner that reads from standard input, which is the keyboard by default:

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

- In general, to use Scanner, follow this procedure:
  1. Determine if a specific type of input is available by calling one of Scanner's hasNextX methods, where X is the type of data desired.
  2. If input is available, read it by calling one of Scanner's nextX methods.
  3. Repeat the process until input is exhausted.

Program 1.15: Use Scanner to compute an average of the values.

```
import java.util.*;
class AvgNums
{
   public static void main(String args[])
   {
      Scanner conin = new Scanner(System.in);
```

```
        int count = 0;
        double sum = 0.0;
        System.out.println("Enter numbers to calculate average.");
        // Read and sum numbers.
        while(conin.hasNext())
        {
            if(conin.hasNextDouble())
            {
                sum += conin.nextDouble();
                count++;
            }
            else
            {
                String str = conin.next();
                if(str.equals("done")) break;
                else
                {
                    System.out.println("Data format error.");
                    return;
                }
            }
        }
        System.out.println("Average is " + sum / count);
    }
}
```

Output:

```
Enter numbers to calculate average.
1
2
3
4
5
done
Average is 3.25
```

- Java programming language provides if, if...else, switch decision making statements. Java programming language provides for, while, do...while loops to handle looping requirements. Java supports break and continue loop control statements.

1.10 CLASSES AND OBJECTS

- In Object-Oriented Programming (OOP) technique, we design a program using objects and classes. Object is the physical as well as logical entity whereas class is the logical entity only.

1.10.1 Overview of Classes

- Classes in Java provide a convenient method for packing together a group of logically related data items and functions that work on them.
- In Java language, the data items are called fields and the functions are called methods. Calling a specific method in an object is described as sending the object of message.
- A class can be defined as, "a template/blueprint that describes the behaviors/ states that object of its type support".
- A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex.
- A simplified general **form/syntax** of a class definition is shown here:

```
class classname
{
   type instance-variable1;
   type instance-variable2;
   type instance-variableN;
   type methodname1(parameter-list)
   {
      // body of method
   }
   type methodname2(parameter-list)
   {
      // body of method
   }
   // ...
   type methodnameN(parameter-list)
   {
      // body of method
   }
}
```

- **Example of Class:**

```
Class Employee
{
   int EmpID:
   float salary;
}
```

Methods in Java:

- A Java method is a collection of statements that are grouped together to perform a certain task.
- Method describes behaviour of an object. A method is a collection of statements that are group together to perform an operation.
- **Syntax:**

```
return_type methodName(parameter_list)
{
    //body of method ...
}
```

- **Example of a Method:**

```
public String getName(String st)
{
    String name="Kamil Ajmal Khan ";
    name=name+st;
    return name;
}
```

Program 1.16: Program for class.

```
public class JavaClassExample
{
    public static void main(String[] arg)
    {
        for (int i = 0; i<4; i++)
        {
            System.out.println("Number is:"+i);
        }
    }
}
```

Output:

```
Number is: 0
Number is: 1
Number is: 2
Number is: 3
```

1.10.2 Overview of Objects

- A real-world entity that has state and behavior is known as an object.
- These real-world objects share two characteristics i.e. state and behavior.

- **Example:** A cat has states - color, name, etc. as well as behaviors like walking and eating.
- In simple words, object is an instance of a class.
- Fig. 1.8 shows a real world example of class and objects. In this figure class Car has objects like Audi, Nissan, Volvo.

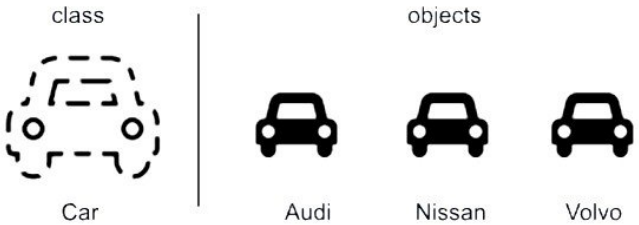


Fig. 1.8: Example of Class and Object

Declaring Objects:

- When you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process.

(i) Declaration:

- We have to specify what type (i.e. class) the object will be. A variable declaration with a variable name with an object type.

  Syntax: `<class name> object name;`

  Where, class name is the name of the already defined class and the object name is a valid identifier.

(ii) Instantiation (Creating Objects):

- Objects are created using the 'new' keyword. This 'new' keyword creates an object of the specified class and returns the reference of that object.
- The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

  Syntax: `<objectname>=new classname([arguments]);`
- Let us create an object for the already defined Box class in the above program segment.

```
Box mybox = new Box();
```

- This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

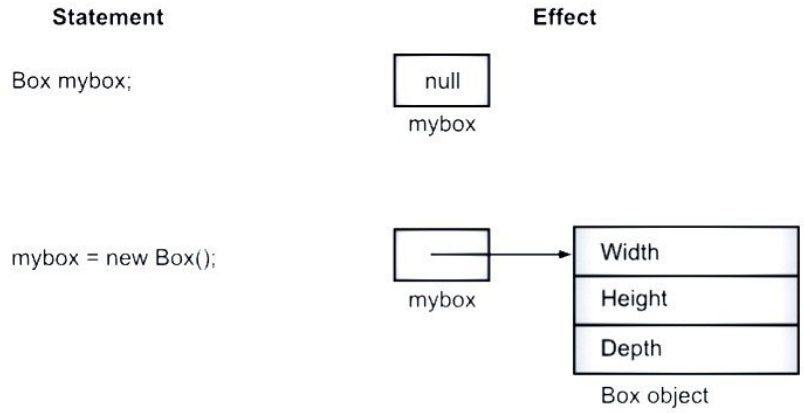


Fig. 1.9

(iii) Accessing Class Members:

- The dot operator (.) or member selection operator is used to access the instance variables of the class.
- We cannot access the instance variables outside of the class without referring to the object.
- Member access follows the following **syntax**:

  ```
  objectname.variablename
  ```

  Here, the 'objectname' is the name of the object and 'variablename' is name of the instance variable.
- So, the instance variables of the class Employee can be accessed as,

  ```
  Employye.E_id=1234;
  Employee.salary=30000;
  ```

Program 1.17: Program for declaration and instantiation of object.

```
import java.io.*;
class Circle
{
   int radius;
   float perimeter;
   float area;
}
class MyCircle
{
   public static void main(String args[])
   {
      final float pi = 3.14f;
      Circle c = new Circle();
      BufferedReadaer in = new BufferedReader
                                    (new InputStreamReader(System.in));
      System.out.print("Enter radius:");
```

```
        try
        {
            c.radius = Integer.parseInt(in.readLine());
            c.perimeter = 2.0f * pi * (float) c.radius;
        }
        catch (IOException ioe)
        {
        ioe.printstackTrace();
        }
        c.area = pi * (float) (c.radius * c.radius);
        System.out.println("Perimeter:"+c.perimeter);
        System.out.println("Area:"+c.area);
    }
}
```

Output:

```
Enter radius: 23
Perimeter: 144.44
Area: 1661.06
```

Difference between Class and Object:

- There are many differences between object and class. A list of differences between object and class are given below:

Table 1.1: Difference between Class and Object

Sr. No.	Class	Object
1.	Class is a blueprint or template from which objects are created.	Object is an instance of a class.
2.	Class is a group of similar objects.	Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.
3.	Class is a logical entity.	Object is a physical entity.
4.	Class is declared using class keyword like class Student{ … }	Mainly Object is created through new keyword like Student s1=new Student();
5.	Class is declared once.	Object is created many times as per requirement.
6.	Class doesn't allocated memory when it is created.	Object allocates memory when it is created.

1.11 DEFINING YOUR OWN CLASSES

- In above section, you started writing simple classes. However, all those classes had just a single main method. Now the time has come to show you how to write the kind of "workhorse classes" that are needed for more sophisticated applications. These classes typically do not have a main method. Instead, they have their own instance fields and methods. To build a complete program, you combine several classes, one of which has a main method.
- To use Java effectively, you want to create and use your own classes. This is one of the great powers of Object-Oriented Languages, the ability to construct programs out of independent building-blocks that cut large problems down into small, easily solvable ones.
- We are going to create a very simple object class that will print a "Hello" message to the screen when called.
- This class will have two different methods that will do the same thing, though we will vary the messages they print a bit so that we can see which one is doing the work.

```
public class HelloClass
{
   // sayHello() prints "Hello!" to the Java console output
   public void sayHello()
   {
      System.out.println("Hello!\n");
   }
   /**
   * doHello() prints "Hello, hello!" to the Java console output
   * It's static, so you don't need to instantiate a HelloClass
   * object to use it
   */
   public static void doHello()
   {
      System.out.println("Hello, hello!\n");
   }
} // End of HelloClass
```

- The two methods we have to print messages are sayHello() and doHello(). To use sayHello(), we need to have a HelloClass object created, then call that object's sayHello()method.
- doHello(), however, is a static method. This means is belongs to the class, not to any object of the class. So we can use it without any HelloClass objects being created first.

1.12 ACCESS SPECIFIERS (PUBLIC, PROTECTED, PRIVATE, DEFAULT)

- In Java, it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. For example, we may not like the objects of a class directly alter the value of a variable or access a method. We can achieve this in Java by applying visibility modifiers. The visibility modifiers are also known as access modifiers.
- Java provides a number of access modifiers to set access (scope) levels for classes, variables, methods and constructors. The four access levels are:
  1. Visible to the package (default/friendly).
  2. Visible to the class only (private).
  3. Visible to the world (public).
  4. Visible to the package and all subclasses (protected).
- Let us see above access specifiers in detail.

1. Default / Friendly Access Modifier (No Keyword):

- If we give no specifier at all, the access is referred as friendly. It means that all the other classes in the current package have access to the friendly member but to all the classes outside the package the members appear to be private.
- Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
- A variable or method declared without any access control modifier is available to any other class in the same package.
- The fields in an interface are implicitly public static final and the methods in an interface are by default public.
- Variables and methods can be declared without any modifiers, as in the following examples:

```
string version = "1.5.1";
boolean processOrder()
{
   return true;
}
```

2. Private Access Modifier (private):

- Methods, variables and constructors that are declared private can only be accessed within the declared class itself.
- Private access modifier is the highest degree restrictive access level. Class and interfaces cannot be private.
- Variables that are declared private can be accessed outside the class if public getter methods are present in the class.
- Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

- The following class uses private access control:

```
public class PLog
{
   private String format;
   public String getFormat()
   {
      return this.format;
   }
   public void setFormat(String format)
   {
      this.format = format;
   }
}
```

- Here, the format variable of the PLog class is private, so there's no way for other classes to retrieve or set its value directly.
- So to make this variable available to the outside world, we defined two public methods i.e., getFormat(), which returns the value of format, and setFormat(String), which sets its value.

3. Public Access Modifier (public):

- A class, method, constructor, interface etc. declared public has the widest possible visibility and accessible everywhere. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.
- However, if the public class we are trying to access is in a different package, then the public class still needs to be imported.
- Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.
- The following function uses public access control:

```
public static void main(String[] arguments)
{
   // ...
}
```

- The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

4. Protected Access Modifier (protected):

- In Java, the visibility level of a "protected" field lies in between the public access and friendly access.
- Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

- The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in an interface cannot be declared protected.
- Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.
- The following parent class uses protected access control, to allow its child class override openSpeaker() method:

```
class AudioPlayer
{
   protected boolean openSpeaker(Speaker sp)
   {
      // implementation details ...
   }
}
class StreamingAudioPlayer
{
   boolean openSpeaker(Speaker sp)
   {
      // implementation details
   }
}
```

- In above code, if we define openSpeaker() method as private, then it would not be accessible from any other class other than AudioPlayer. If we define it as public, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, that's why we used protected modifier.

Table 1.2: Visibility of field in a class of Java.

Access Location	Public modifier	Protected modifier	Friendly (default) modifier	Private protected modifier	Private modifier
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

Program 1.18: Program for access modifier.

```
class Rectangle
{
   public int length;            //public Access
   public int breadth;           //public Access
   static int rectCount = 0;     //default Access
   //Constructor to initialize length and breadth Rectangle()
   {
      rectCount++;
   }  //method to calculate area of rectangle
   public int area()
   {
      int rectArea;
      rectArea = length * breadth;
      return rectArea;
   }
}
class AccessSpecifer
{
   public static void main(String[] args)
   {
      //create first rectangle object
      Rectangle firstRect = new Rectangle();
      //accessing public members outside that Rectangle class
      firstRect.length = 10;
      firstRect.breadth = -30;
      System.out.println("Area of Rectangle is:"+firstRect.area());
      //Rectangle.rectCount = 5;
      //accessing member with default access
      System.out.println("Number of Object Created"+Rectangle.rectCount);
   }
}
```

Output:

```
Area of Rectangle is : - 300
Number of Object Created : 1
```

1.13 ARRAY OF OBJECTS

- In Java, rather than creating object of a particular or class, we can also create array of objects.
- In Java, an array of objects can also be created to store and process the objects. We can also create array of object. The process to declare and create an array of objects is similar to the process of declaring and creating an array of simple data types.
- It is possible to create array of objects of user created/defined class.
- When we create an array of objects then the data type gets replaced with the class name. The following is the **syntax** to declare and create an array of objects:

```
class_name arr_name [ ] = new class_name [size];
```

  Where,
  - class_name is the name of the class whose objects are used to create an array of objects.
  - arr_name is the name of the array, which is to be created.
  - new is the Java operator, which is used to allocate memory to an array.
  - size is the number of objects that an array can hold.
- Let's now suppose that we have a class, ArrayDemo. We have to create an array of objects of this class. To create an array of objects to ArrayDemo class see the following example:

```
ArrayDemo a_obj[ ] = new ArrayDemo [2]:
```

- The a_obj array can hold 2 objects of ArrayDemo class. To use the a_obj array, we have to create objects of every array index.
- See the following code snippet to create objects of every index of a_obj array:

```
a_obj [0] = new ArrayDemo();
a_obj [1] = new ArrayDemo();
```

- We can use loop to create array of objects as follows:

```
ArrayDemo[] arr = new ArrayDemo[10];
for(int i = 0; i = 10; i++)
arr[i] = new ArrayDemo();
```

Program 1.19: Program for array of object. **[W-18]**

```
class Employee
{
    int id;
    String name;
    public void setData(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
```

```
    public void displayData()
    {
        System.out.println("Employee ID :"+this.id);
        System.out.println("Employee Name :"+this.name);
    }
}
class Main
{
    public static void main(String args[])    //OR
    {      //Employee[] emp=new Employee[2];
        Employee[] emp = new Employee[2];     //for(int i = 0; i < 2; i++)
        emp[0].setData(1,"Amar");             //emp[i] = new Employee()
        emp[1].setData(2,"Akbar");            //emp[i].displayData();
        emp[0].displayData();
        emp[1].displayData();
    }
}
```

Output:

```
Employee ID : 1
Employee Name : Amar
Employee ID : 2
Employee Name : Akbar
```

1.14 CONSTRUCTOR [S-19]

- A constructor is a special method of a class in Java programming that initializes an object of that type.
- Constructors have the same name as the class itself. A constructor is automatically called when an object is created.
- Constructor constructs the values i.e. provides data for the object that is why it is known as constructor.

Syntax:

```
class_Name
{
    Constructor_Name (same as class_Name); // Constructor
    {
        //constructor body…..
    }// end of Constructor
}// end of Class
```

- There are basically following rules defined for the constructor:
  1. Constructor name must be same as its class name.
  2. Constructor must have no explicit return type.
  3. Constructors may be private, protected or public.
  4. Multiple constructors may exist, but they must have different signatures, i.e., different numbers and/or types of input parameters.

Program 1.20: Program for constructors.

```
class Rectangle
{
   int length;
   int breadth;
   //constructor to initialize length and breadth of rectangle
   Rectangle()
   {
      length = 5;
      breadth= 6;
   }
   //method to calculate area of rectangle
   int area()
   {
      int rectArea = length * breadth;
      return rectArea ;
   }
}
//class to create rectangle objects and calculates area
class ConstructorExample
{
   public static void main(String[] args)
   {
      Rectangle firstRect = new Rectangle();
      System.out.println("Area of Rectantangle:"+ firstRect.area());
   }
}
```

Output:

```
Area of rectangle: 30
```

Types of Java Constructor:

- Constructors can be classified into two types, Default Constructors and Parameterized Constructors as shown in Fig. 1.10.

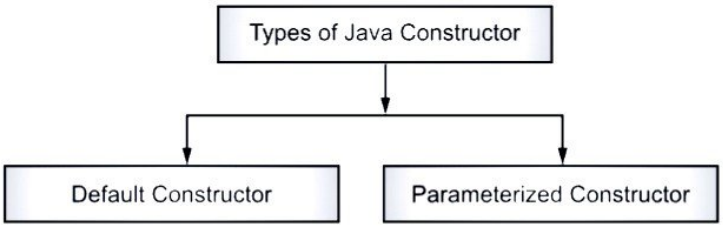


Fig. 1.10: Types of Constructor

1. Default Constructor:

- The constructors which does not accept any argument is called Default Constructor. The argument list is void.
- In other word, when the object is created Java creates a no-argument constructor automatically known as Default Constructor.
- It does not contain any parameters nor does it contain any statements in its body. Its only purpose is to enable you to create object of class type.
- Default constructor provides the default values to the object like 0, null etc. depending on the type.

Syntax: `<class_name>(){}`

Program 1.21: Program for default constructor.

```
class StudentInfo
{
    int rollno;
    int marks1;
    int marks2;
    int total;
    StudentInfo()   //no-arg default constructor
    {
        rollno=2;
        marks1=30;
        marks2=40;
    }
    //method to calculate total
    int Total()
    {
        total = marks1 + marks2;
        return total;
    }
```

```
    //method to display the result
    void displayResult()
    {
        System.out.println("Roll no of student is" + rollno);
        System.out.println("marks1 are" + marks1);
        System.out.println("marks2 are" + marks2);
        System.out.println("total is" + total);
    }
}
class StudentResult
{
    public static void main(String args[])
    {
        int total1,total2,grandtotal;
        StudentInfo student1 =new StudentInfo();    //creates first object
        StudentInfo student2 =new StudentInfo();    //creates second object
        total1=student1.Total();
        total2=student2.Total();
        grandtotal=total1+total2;
        student1.displayResult();
        student2.displayResult();
        System.out.println("Grand Total is" + grandtotal);
    }
}
```

Output:

```
Roll no of student is 2
marks1 are 30
marks2 are 40
total is 70
Roll no of student is 2
marks1 are 30
marks2 are 40
total is 70
Grand Total is 140
```

2. Parameterized Constructor:

- Constructor can take value, value is called as argument. A constructor that has parameters is known as Parameterized Constructor. Parameterized constructor is used to provide different values to the distinct objects.
- Using parameterized constructor, it is possible to initialize objects with different set of values at the time of their creation. These different set of values initialized to objects must be passed as arguments when constructor is invoked.
- The parameter list can be specified in the parentheses in the same way as parameter list is specified in the method.
- The **syntax** for constructor is as follows:

```
ConstructorName([parameterList])
{
    //constructor body ...
}
```

- Here, the ConstructorName is same as the class name it belongs to. The parameterList is the list of optional zero or more parameter(s) that is specified after the classname in parentheses. Each parameter specification, if any, consists of a type and a name and is separated from each other by commas.

Program 1.22: Program for parametrized constructor.

```
class StudentInfo
{
    int rollno;
    int marks1;
    int marks2;
    int total;
    StudentInfo(int roll_no,int m1,int m2)  //Parameterized constructor
    {
        rollno=roll_no;
        marks1=m1;
        marks2=m2;
    }
    //method to calculate total
    int Total()  //Method declaration
    {
        total = marks1 + marks2;
        return total;
    }
```

```
    //method to display the result
    void displayResult()
    {
        System.out.println("Roll no of student is" + rollno );
        System.out.println("marks1 are" + marks1 );
        System.out.println("marks2 are" + marks2);
        System.out.println("total is" + total);
    }
}
class StudentResult
{
    public static void main(String args[])
    {
        int total1,total2,grandtotal;
        StudentInfo student1 = new StudentInfo(1,50,80);
        StudentInfo student2 = new StudentInfo(2,40,60);
        total1=student1.Total();
        total2=student2.Total();
        grandtotal=total1+total2;
        student1.displayResult();
        student2.displayResult();
        System.out.println("Grand Total is" + grandtotal);
    }
}
```

Output:

```
Roll no of student is 1
marks1 are 50
marks2 are 80
total is 130
Roll no of student is 2
marks1 are 40
marks2 are 60
total is 100
Grand Total is 230
```

1.14.1 Overloading Constructors

- Constructors having the same name with different parameter list is called as Constructor overloading.
- The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Program 1.23: Program for constructor overloading.

```
class Student
{
    int id;
    String name;
    int age;
    Student(int i,String n)
    {
        id = i;
        name = n;
    }
    Student(int i,String n,int a)
    {
        id = i;
        name = n;
        age=a;
    }
    void display()
    {
        System.out.println(id+" "+name+" "+age);
    }
public static void main(String args[])
{
    Student s1 = new Student(1,"Kamil", 22);
    Student s2 = new Student(2,"Prajakta",20);
    s1.display();
    s2.display();
}
```

Output:

```
1 Kamil 22
2 Prajakta 20
```

Methods Overloading:

- In Java, we can define number of methods in a class with the same name. Defining two or more methods with the same name in a class is called Method Overloading.
- The overloaded constructors are called here as method overloading because we are allowed to create methods with same name, but different parameters can be passed with different definitions. This is referred as **Polymorphism**.
- When we call a method with its object, Java first checks for the method name and then for the number of parameters and its types. Then it decides which method should be executed.

Program 1.24: Program for method overloading.

```
class overload
{
   int m, n;
   overload()                  //default constructor
   {
      m = 5;
      n = 28;
   }
   overload (int p, int q)     //parameterized constructor
   {
      m = p;
      n = q;
   }
   overload (double x, double y)
   {
      m = x;
      n = y;
   }
   void display()
   {
      System.out.println(m);
      System.out.println(n);
   }
}
class mainover
{
   public static void main(String args[])
   {
      overload ob1 = new overload();
      overload ob2 = new overload(10, 5);
      overload ob3 = new overload(25.6, 80.5);
```

```
        ob1.display();
        ob2.display();
        ob3.display();
    }
}
```

Output:

```
5   }
28  }  for ob1
10  }
5   }  for ob2
25  }
80  }  for ob3
```

- Here, for object ob1, the default constructor is called. For object ob2 the parameterized constructor "overload (int p, int q)" is called, whereas for object ob3, the parameterized constructor "overload (double x, double y)" is called. But because of the data type of m and n are int, the value of x and y i.e. 25.6 and 80.5 get truncated to integer part.

Table 1.3: Difference between Constructor and Method

Constructor	Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor name must be same as the class name.	Method name may or may not be same as class name.
Constructor is invoked implicitly.	Method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor must not have return type.	Method must have return type.

1.14.2 Use of 'this' Keyword

- Many times it is necessary to refer to its own object in a Method or a Constructor. To allow this Java defines the 'this' keyword.
- As we know in C++, sometimes a method will need to refer to the object that invoked it. To do so, Java also has a 'this' keyword. 'this' can be used inside any method to refer to the current object. It means that 'this' is always a reference to the object on which the method was invoked.
- **Syntax:** `this.field`

Usage of this Keyword:

1. 'this' keyword can be used to refer current class instance variable.
2. 'this' keyword can be used to invoke current class constructor.
3. 'this' keyword can be used to invoke current class method (implicitly).
4. 'this' can be passed as an argument in the method call.

- Fig. 1.11 shows use of 'this' keyword.

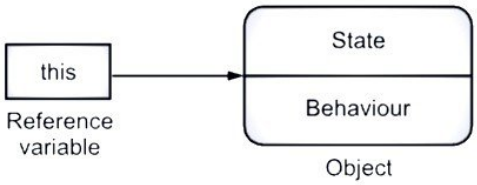


Fig. 1.11: Use of this keyword

- **Example:**

```
class BoxDim
{
   int height;
   int depth;
   int length;
   BoxDim(int height, int depth, int length)
   {
      this.height = height;
      this.depth = depth;
      this.length = length;
   }
}
```

- The class 'BoxDim' contains three instance variables i.e. height, depth and length. The constructor of the class also contains three different local variables with the same names of instance variables.
- Compiler will not show any error here. Then how differentiate among these variables? The 'this' keyword does this job. 'this' will refer to the variables of its own class. It is acting as the object of the current class.
- In the method, the 'this' keyword is used to differentiate between instance and local variables. This concept is also referred as 'Instance variable hiding'. This resolves the name-space collisions.

Program 1.25: Program for 'this' keyword.

```
class Rectangle
{
   int length,breadth;
   void show(int length,int breadth)
```

```
    //Formal and instance variable name are same
    {
        this.length=length;     //Use of this keyword
        this.breadth=breadth;
    }
    int calculate()
    {
    return(length*breadth);
    }
}
/* Main class */
public class UseOfThisOperator
{
    public static void main(String[] args)
    {
        Rectangle rectangle=new Rectangle();
        rectangle.show(8,6);
        int area = rectangle.calculate();
        System.out.println("The area of a Rectangle is:" + area);
    }
}
```

Output:

```
The area of a Rectangle is: 48
```

1.15 STATIC BLOCK, STATIC FIELDS AND STATIC METHODS

1.15.1 Static Block

- Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. This code inside static block is executed only once: the first time the class is loaded into memory. For example, check output of following Java program.

Program 1.26: Program using of static block.

```
class Test
{
    static int i;
    int j;    // start of static block
    static
    {
        i = 10;
        System.out.println("static block called ");
    }
}
```

```
class Main
{
    public static void main(String args[])
    {
        System.out.println(Test.i);
    }
}
```

Output:

```
static block called
10
```

1.15.2 Static Fields & Static Methods

- The static keyword is used to create methods that will exist independently of any instances created for the class.
- Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Program 1.27: Java program to using static field and methods.

```
public class CountInstance
{
    private static int numInstances = 0;
    protected static int getCount()
    {
        return numInstances;
    }
    private static void addInstance()
    {
        numInstances++;
    }
    CountInstance()
    {
        CountInstance.addInstance();
    }
    public static void main(String[] arguments)
    {
        System.out.println("Starting with" + CountInstance.getCount()
                                                        + "instances");
```

```
        for (int i = 0; i < 500; ++i)
        {
            new CountInstance();
        }
        System.out.println("Created" + CountInstance.getCount()
                                                        + "instances");
    }
}
```

Output:

```
Starting with 0 instances
Created 500 instances
```

Program 1.28: Program using static method which maintains bank account information about various customers.

```
class Account
{
     int accountNo; //account ID
     double balance;
     static double rate = 0.05;
     void setData(int n,double bai)
     {
        accountNo = n;
        balance = bai;
     }
      void quarterRatecal()
     {
        double interest = balance * rate * 0.25;
        balance += interest;
     }
     static void modifyRate(double incr)
     {
        rate += incr;
        System.out.println("Modified Rate of Interest:" + rate);
     }
     void show()
     {
        System.out.println("Account Number:" + accountNo);
        System.out.println("Rate of Interest:" +rate)
        System.out.println("Balance:"+ balance);
     }
}
```

```
public class StaticMethod
{
     public static void main(String[] args)
     {
        Account acc1 = new Account();
        Account acc2 = new Account();
        Account.modifyRate(0.01);
        System.out.println("Customer1 Information......");
        acc1.setData(201,1000);
        acc1.quarterRatecal(); //Calculate interest
        acc1.show(); //display account interest
        System.out.println("Customer2 Information......");
        acc1.setData(202,1000);
        acc2.quarterRatecal(); //Calcuate interest
        acc2.show(); //display account information
     }
}
```

Output:

```
Modified Rate of Interest : 0.060000000000000005
Customer1 Information ...
Account Number : 201
Rate of Interest : 0.06
Balance : 1000.0
Customer2 Information ...
Account Number : 202
Rate of Interest : 0.06
Balance of 1000.0
```

- The Account class in program contains a static method modifyRate(). It modifies the rate of interest by the value specified as argument (0.01). Only static method can access a static field.

1.16 PREDEFINED CLASSES - OBJECT CLASS METHODS (equals(), toString(), hashcode())

- Java provides different types of predefined classes that programmers can use to build sophisticated programs. If you have a choice of using a predefined standard class or devising your own custom class that provides the same functionality, choose the standard class.
- The advantages of using a standard class include:
  1. The effort to produce the class is eliminated entirely; you can devote more effort to other parts of the application's development.
  2. Java compiler automatically imports all the classes in the java.lang package into every source file.

- Some of the most important classes of the java.lang package:
  - Object
  - Math
  - The wrapper classes
  - String
  - StringBuffer

Object Class:

- The object class is the parent class of all the classes in Java by default. In other words, it is the topmost class of java.
- The java.lang.Object class is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
- Following is the declaration for java.lang.Object class:

```
public class Object
```

- The Object class provides some common behaviour to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Methods of Object Class:

- We already known object class is the superclass of all classes. Some methods of the object class are:
  - **protected Object clone():** Creates and returns a copy of this object.
  - **boolean equals(Object obj):** Indicates whether some other object is "equal to" this one.
  - **protected void finalize():** This method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
  - **Class getClass():** Returns the runtime class of this Object.
  - **int hashCode():** Returns a hash code value for the object.
  - **void notify():** Wakes up a single thread that is waiting on this object's monitor.
  - **void notifyAll():** Wakes up all threads that are waiting on this object's monitor.
  - **String toString():** Returns a string representation of the object.
  - **void wait():** Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
  - **void wait(long timeout):** Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
  - **void wait(long timeout, int nanos):** Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Program 1.29: Program for object class.

```
class Rectangle extends Object
{
   private double length,breadth;
   Rectangle(double x,double y)
   {
      length = x ;
      breadth = y ;
   }
   public void area()
   {
      System.out.println("Area of Rectangle is = " + breadth);
   }
   public void circumference()
   {
      System.out.println("Circumference of
                                          Rectangle is="+2*(length+breadth));
   }
}
class ObjectClass
{
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(10,20);
      Rectangle rl = new Rectangle(10,20);
      System.out.println("String Representation = " + r.toString());
      System.out.println("Class Name = " + r.getClass());
      System.out.println("Hash Code = " + r.hashCode());
      System.out.println("r.equals(rl) = " + r.equals(rl));
   }
}
```

Output:

```
String Representation = Rectangle@76ed5528
Class Name = class Rectangle
Hash Code = 1995265320
r.equals(rl) = false
```

1.17 GARBAGE COLLECTION (FINALIZE( ) METHOD)

- We know that constructors are used to initialise an object when it is declared. This process is referred as initialisation. Java runtime is an automatic garbage collecting system.
- In C++, if we create objects dynamically with new operator then the memory allocated for these objects gets deallocated manually; just by using delete operator.
- In Java, the deallocation is handled automatically. The technique which deallocates memory [assigned to objects] automatically is called garbage collection.
- In Java, garbage means unreferenced objects. Garbage collector is automatically invoked when the program is being run.
- Garbage collector can be called by calling gc() method of runtime class. There are many ways, in which an object can be unreferenced:
  1. By nulling the reference,
  2. By assigning a reference to another.
  3. By anonymous object etc.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- In Java, it is performed automatically. So, Java provides better memory management.

Advantage of Garbage Collection:

1. It is automatically done by the garbage collector(a part of JVM), so we don't need to make extra efforts.
2. It makes Java memory efficient because garbage collector removes the unreferenced objects from heap memory.

finalize() Method:

- When the reference of an object is not exists then it is assumed that object is no longer needed. And the memory which is occupied by the object is reclaimed. So garbage collector automatically frees the memory resources used by the objects.
- But sometimes the object may hold some non-java resources such as file handle or window character font. The garbage collector cannot free these resources.
- In order to free these resources, we need finalize() method. This is called finalization. By using the finalization, we can define specific action that will occur when an object being destroyed by the garbage collector.
- To add a finalizer to our class, we have to define it as a finalize() method.
- The finalize() method has following general **form/Syntax**:

```
protected void finalize()
{
    // finalization code here ...
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.

- The garbage collector runs periodically. It checks for objects which are no longer referenced by any running state or indirectly through any running object. Before it gets free, the java run-time calls the finalize() method.
- Protected keyword prevents the access of finalize() code from outside class. Java does not support destructors.

Program 1.30: Following example shows that the objects allocated external resources should provide a finalize method that cleans them up or class will create a resource leak.

```
import java.util.*;
public class ObjectfinDemo extends GregorianCalendar
{
   public static void main(String[] args)
   {
      try
      {
         // create a new ObjectfinDemo object
         ObjectfinDemo cal = new ObjectfinDemo();
         // print current time
         System.out.println("" + cal.getTime());
         // finalize cal
         System.out.println("Finalizing...");
         cal.finalize();
         System.out.println("Finalized.");
      } catch (Throwable ex)
      {
         ex.printStackTrace();
      }
   }
}
```

Output:

```
Sat Sep 22 00:27:21 EEST 2014
Finalizing...
Finalized.
```

Summary

- James Gosling is called Father of Java Programming. Java is an Object Oriented Programming language means Java enables us not only to organize our program code into logical unit called objects but also to take advantage of encapsulation, inheritance and polymorphism and so on.

- Various features of Java includes Simple, Secure, Portable, Object-oriented, Robust, Multithreaded, Architecture-neutral, Interpreted, High Performance, Distributed, Dynamic.
- Java Virtual Machine (JVM) is a virtual machine that provides runtime environment to execute Java byte code.
- Comments are non-executable statements and are ignored by the Java compiler.
- The comments increase the readability of the programs.
- An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions.
- Array is a data structure where we store similar elements. We can store only fixed set of elements in a Java array. Types of array-1D array, 2D array.
- A one-dimensional array having only one subscript. In Java 2D array is a matrix of rows and columns.
- A Java program is basically a collection of classes. A class is defined by a set of declaration statements and methods containing executable statements.
- Object is the physical as well as logical entity whereas class is the logical entity only.
- A constructor is a special method of a class in Java programming that initializes an object of that type. Constructors have the same name as the class itself. A constructor is automatically called when an object is created.
- We know that constructors are used to initialise an object when it is declared. This process is referred as initialisation. Java runtime is an automatic garbage collecting system.

Check Your Understanding

1. What allows the programmer to destroy an object x?
   (a) x.delete()
   (b) x.finalize()
   (c) Runtime.getRuntime().gc()
   (d) Only the garbage collection system can destroy an object.
2. What is the stored in the object obj in following lines of code?
   box obj;
   (a) Memory address of allocated memory of object
   (b) NULL
   (c) Any arbitrary pointer
   (d) Garbage
3. Which of these operators is used to allocate memory for an object?
   (a) malloc (b) alloc
   (c) new (d) give

4. What is the output of this program?

```
class main_class
{
   public static void main(String args[])
   {
      int x = 9;
      if (x == 9)
      {
         int x = 8;
         System.out.println(x);
      }
   }
}
```

(a) 9 (b) 8
(c) Compilation error (d) Runtime error

5. Which of these is necessary to specify at time of array initialization?
(a) Row (b) Column
(c) Both Row and Column (d) None of the mentioned

6. Which three are legal array declarations?
1. int [] myScores [];
2. char [] myChars;
3. int [6] myScores;
4. Dog myDogs [];
5. Dog myDogs [7];
(a) 1, 2, 4 (b) 2, 4, 5
(c) 2, 3, 4 (d) All are correct

7. Which one of the following will declare an array and initialize it with five numbers?
(a) Array a = new Array(5); (b) int [] a = {23,22,21,20,19};
(c) int a [] = new int[5]; (d) int [5] array;

Answers

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

Practice Questions

Q.I Answer the following questions in short:

1. What is Java?
2. Why Java is platform-neutral language?

3. Explain the secure feature of Java.
4. Why Java needs compiler and interpreter?
5. Why Java is called portable?
6. Which are the tasks performed by Java environment?
7. How to compile Java program?
8. How to run Java program?
9. How Garbage collection is done in java?
10. How many types of access specifiers are provided by java?
11. What is array?
12. How a comment is added in java program?
13. What are the types of array?

Q.II Answer the following questions:

1. Explain the significance of each of the following:
   (i) finalize() (ii) public
   (iii) protected (iv) private
2. Discuss garbage collection in Java.
3. What are objects? How to create objects?
4. What is a constructor? How many types of constructors are present in Java?
5. Explain static block.
6. Explain static fields and methods.
7. Explain various types of access modifiers in detail.
8. Create a class circle, consider pi and radius. Calculate the area of a circle and display it.
9. Write a program to find the prime numbers.
10. Explain at least five features of Java.
11. Explain Java environment in detail.
12. Write short note on: JVM, Compiler, Interpreter, constructor, garbage collections, predefined classes, array of objects, static fields and methods.
13. Explain accepting input from console (Using BufferedReader and Scanner class).
14. Explain accepting input using Command line argument.
15. Predict the output:

```
Public class Test
{
   Public static void main (string[ ] args)
   {
      boolean [ ] [ ] x = new boolean [3] [ ];
      x [0] = new boolean [1];
      x [1] = new boolean [2];
      x [2] = new boolean [3];
      System.out.println ("x[2] [2] is" + x [2] [2]) ;
   }
}
```

Ans. `x[2] [2] is false`

16.
```
Class Test
{
    Public static void main (string [ ] args)
    {
        int [ ] x = { 1, 2, 3, 4};
        int [ ] y = x ;
        x = new int [2];
        for (int i = 0; i < y.length; i ++)
        System.out.println(y [i]);
    }
}
```
Ans. 1 2 3 4

Q.III Define the terms:

1. this keyword
2. Compiler
3. Interpreter
4. Array
5. Class
6. Object
7. Constructor
8. Garbage collections

Previous Exam Questions

Winter 2018

1. "Import statement is not essential in Java." Justify True/False. **[1 M]**

Ans. Refer to Section 1.5.

2. Create a class Employee with data member as name and salary. Accept the information of 5 employees and display it (use arrayof objects). **[4 M]**

Ans. Refer to Program 1.19.

3. Explain any three features of Java. **[4 M]**

Ans. Refer to Section 1.3.

4. Explain the various ways used to accept input from console in Java with proper syntax and example. **[3 M]**

Ans. Refer to Section 1.9.

Summer 2019

1. When does method overloading determine? **[1 M]**

(i) At a run time (ii) At a compile time

(iii) At a coding time (iv) At an execution time

Ans. (ii)

2. What is constructor? **[1 M]**

Ans. Refer to Section 1.14.

3. Create student class having data member (roll_no, name, percentage) accept values and display details (use command line argument). **[3 M]**

Ans. Refer to Section 1.8.

■■■

2... Inheritance and Interface

Learning Objectives...

- To understand basic concepts of Inheritance.
- To study Types of inheritance.
- To learn Superclass, Subclass and use of Super Keyword.
- To study use of final keyword related to variable, method and class.
- To study the use of abstract class and abstract methods.
- To study about Interfaces.
- To learn Runtime Polymorphism using Interface.
- To study creating, accessing and using packages.

2.1 INHERITANCE BASICS

- Object Oriented Programming (OOP) allows you to create new class based on a class that has been already defined using inheritance.
- Inheritance is one of the most dominant and vital feature of the OOP's because it supports the hierarchical classifications, reusability of class and defined to specialization.
- Inheritance allows us to create hierarchical classification. Using inheritance, a class can be inherited by another class. This base class is called superclass and the inherited class is called as subclass.
- Inheritance represents the 'IS-A relationship', also known as 'Parent-Child relationship'.
- To support the concept of multiple inheritance, Java provides the alternative approach known as Interface.

2.1.1 Overview of Inheritance [W-18, S-19]

- Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality.
- Inheritance can be defined as, "the process where one object acquires the properties of another". In simple words, the mechanism of deriving a new class from an old class is called inheritance.

- A class that is derived from any another class is called a subclass or derived class or extended class or child class. The class from which a subclass is derived is called super class or base class or parent class.
- Fig. 2.1 illustrating a class called Vehicle, which has two subclasses called Car and Truck.

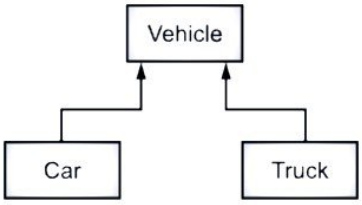


Fig. 2.1: Example of Inheritance

- The class Car and Truck inherits from the class Vehicle. The Vehicle class is the superclass of Car and Truck. Car and Truck are subclasses of Vehicle.

Advantages of Inheritance:

1. Code reusability through inheritance is increased.
2. Inheritance provides a clear model structure which is easy and simple to understand.
3. Code is easy to manage which can be divided into parent and child classes.
4. No need to write code from scratch. You can start coding with existing class.
5. Inheritance can be used easily to convert smaller systems into larger systems.
6. Good at representing the objects.

2.1.2 extends Keyword

- To inherit a class, we have to simply incorporate the definition of one class into another by using the 'extends' keyword.
- The **syntax** is given below:

```
class subclassname extends superclassname
{
    //Body of the class ...
}
```

- The keyword 'extends' specifies that the properties of superclassname are extended to subclassname. After this the subclass will contain all the methods of super class and it will add the members of its own. Remember, we can not extend a subclass from more than one super class. Direct multiple inheritance is not allowed in Java.

Program 2.1: A simple example of inheritance with extends.

```
class Super
{
   int i,j;
   void show()
   {
      System.out.print("i and j: ");
      System.out.println( + i + " " + j);
   }
}
class Sub extends Super
{
   int k;
   void display()
   {
      System.out.println("k: " + k);
   }
   void sum()
   {
      System.out.println("i+j+k: " + (i+j+k));
   }
}
 public class SimpleInheritance
{
   public static void main(String args[])
   {
      Super a = new Super();
      Sub b = new Sub();
      a.i = 5;
      a.j = 12;
      System.out.println("Contents of super: ");
      a.show();
      System.out.println();
      b.i = 11;
      b.j = 13;
      b.k = 17;
      System.out.println("Contents of sub: ");
```

```
        b.show();
        b.display();
        System.out.println();
        System.out.println("Sum of i, j and k in sub:");
        b.sum();
    }
}
```

Output:

```
Contents of super: i and j: 5 12
Contents of sub: i and j: 11 13 k: 17
Sum of i, j and k in sub: i+j+k: 41
```

- Above program shows an example of simple inheritance in which the 'Sub' class is inherited from 'Super' class. So the members of 'Super' class that is, 'i', 'j' and method 'show ( )' are available in 'Sub' class. By creating the object of 'Sub' class we can access all these members. But the methods 'display ( )', 'sum ( )' and variable 'k' cannot accessed in 'Super' class. So 'Super' class object cannot use these members.

2.1.3 Types of Inheritance [W-18, S-19]

- Fig. 2.2 shows various types of inheritance in Java.

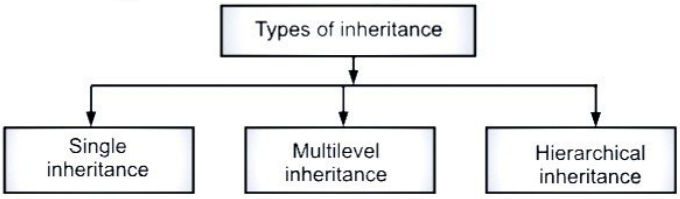


Fig. 2.2: Types of Inheritance

1. Single Inheritance:

- When a class extends another one class only then we call it a single inheritance.
- In case of single inheritance there is only a sub class and its parent class. It is also called simple inheritance or one level inheritance.
- Fig. 2.3 shows that class B extends only one class which is A. Here, A is a **parent class** of B and B would be a **child class** of A.

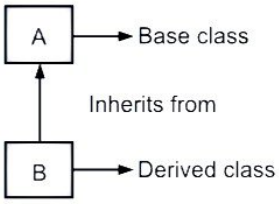


Fig. 2.3: Single Inheritance

- The Fig. 2.3 shows that class A is super class and class B is the sub class. Let us consider that we have class 'vehicle' which is inherited by class 'Taxi', (See Fig. 2.4).

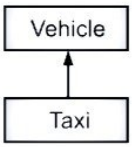


Fig. 2.4: Example of Single Inheritance

Program 2.2: Program of single inheritance.

```
class First
{
   int val;
   void init()
   {
      BufferedReadaer  in = new BufferedReader
      (new InputStreamReader(System.in));
      System.out.print("Enter a number: ");
      try
      {
         val =Integer.parseInt(in.readLine());
      }
      catch(IOException ioe)
      {
         ioe.printStackTrace();
      }
      int square()
      {
         return(val*val);
      }
   }
}
class Second extends First
{
   int mem;
   int cube()
   {
      mem = square() * val;
      return mem;
   }
}
```

```
public class SingleInheritance
{
    public static void main(String args[])
    {
        Second s = new Second();
        s.init();
        System.out.println("Cube: "+s.cube());
    }
}
```

Output:

```
Enter a number: 5
Cube: 125
```

- Here, 'Second' class is inherited from 'First' class. In 'Second' class, the member method 'Square( )' has been called in an expression. After creating the object of the 'Second' class, we called the method 'init ( )' of the 'First' class.

2. Multilevel Inheritance:

- We can create any layers in the inheritance as we want, this is called Multilevel Inheritance. This hierarchy can be created at any level of inheritance.
- When a subclass is derived from a derived class then this mechanism is known as the Multilevel Inheritance.
- The derived class is called the subclass or child class for its parent class and this parent class works as the child class for it's just above (parent) class.
- The general **form/syntax** is,

```
class A
{
......
......
......
}
class B extends A
{
......
......
......
}
class C extends B
{
......
......
......
}
```

- Class A may be the superclass for Class B and Class B may be the superclass for Class C and Class C may be the superclass for Class D. In above case, Class B inherits all properties of Class A, Class C inherits all properties of Class B and Class D inherits all properties of Class C.

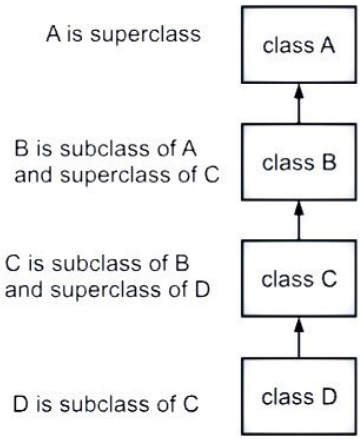


Fig. 2.5: Representation of Multilevel Inheritance

Program 2.3: Program illustrates the concept of multilevel inheritance.

```
class White
{
   int num;
   White(int x)   //statement1
   {
      num = x;
   }
}
class Red extends White
{
   Red(int y)
   {
      super(y);   //statement2
   }
   void print()
   {
      System.out.println("Value of num: "+num);
   }
}
```

```
class Magenta extends Red
{
   int maggy;
   Magenta(int z)
   {
      super(z);   //statement3
      maggy = z;
   }
}
public class MultiLevel
{
   public static void main(String args[])
   {
      Magenta m = new Magenta(96); //statement4
      m.print();
      System.out.println("Value of maggy: "+m.maggy);
   }
}
```

Output:

```
Value of num: 96
Value of maggy: 96
```

- In above program, three different classes i.e. White, Red and Magenta are created. 'Magenta' is inherited from 'Red' and 'Red' is inherited from 'White'. So 'Magenta' will contain all the members from both 'Red' and 'White' and including its own members. 'Red' will contain its own members and members from 'White'.
- When statement4 is executed, statement3, statement2 and statement1 are executed in sequence. Because, 'super ( )' has linked all of them. Remember, super ( ) can only call immediate super class' constructor. That is, 'super' from 'Magenta' can call only the constructor of 'Red' and 'super' from 'Red' can call only the constructor of 'White'. The object 'm' has access to both the variables used in the hierarchy i.e. 'num' and 'maggy'. It can call the 'print()' method.

3. Hierarchical Inheritance:

- When a class has more than one child classes (sub classes) have the same parent class then such kind of inheritance is known as Hierarchical Inheritance.
- In case of hierarchical inheritance we derive more than one subclass from a single super class.
- In below example class B, C and D inherits the same class A. A is parent class (or base class) of B, C and D.

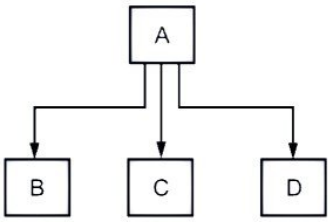


Fig. 2.6: Hierarchical Inheritance

- Fig. 2.7 represents hierarchical inheritance. Here, class Student is having children as 'Medical, Engineering, and Non-science at first level, where class Engineering itself is having some child at the second level. Hierarchical inheritance is basically the combination of more then one type of inheritance.

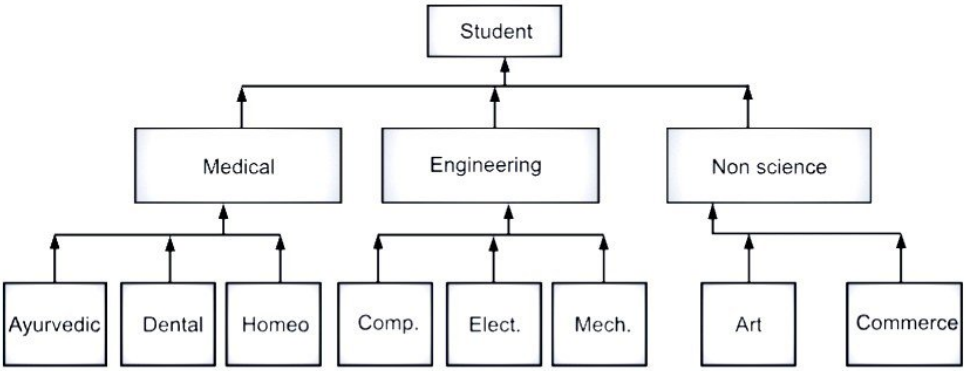


Fig. 2.7: Example of Hierarchical Inheritance

Program 2.4: Program for hierarchical inheritance.

```
class A
{
   public void methodA()
   {
      System.out.println("Method of Class A");
   }
}
class B extends A
{
   public void methodB()
   {
      System.out.println("Method of Class B");
   }
}
class C extends A
{
```

```
    public void methodC()
    {
        System.out.println("Method of Class C");
    }
}
class D extends A
{
    public void MethodD()
    {
        System.out.println("Method of Class D");
    }
}
class MyClass
{
    public void methodB()
    {
        System.out.println("Method of Class B");
    }
    public static void main(String args[])
    {
        B obj1 = new B();
        C obj2 = new C();
        D obj3 = new D();
        obj1.methodA();
        obj2.methodA();
        obj3.methodA();
    }
}
```

Output:

```
Method of Class A
Method of Class A
Method of Class A
```

2.2 SUPERCLASS AND SUBCLASS [W-18]

- Object Oriented Programming allows classes to inherit commonly used state and behaviour from other classes.
- The idea of inheritance is simple but powerful i.e., when you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class.

- A class that is derived from another class is called a **Subclass** (also a derived class, extended class, or child class). The class from which the subclass is derived is called a **Superclass** (also a base class or a parent class).
- Fig. 2.8 (a) shows, If class Y is a subclass of class X, we also say that class X is a superclass of class Y. (Sometimes the terms derived class and base class are used instead of subclass and superclass; this is the common terminology in Java).
- A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure).
- The relationship between subclass and superclass is shown in Fig. 2.8 (a).

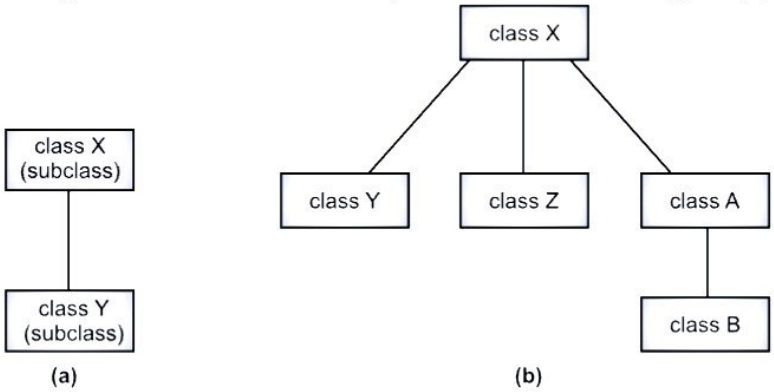


Fig. 2.8: The Relationship between Subclass and Superclass

- In Java, to create a class named "Y" as a subclass of a class named "X", you would write:

```
class Y extends X
{
    .  // additions to, and modifications of,
    .  // stuff inherited from class X
}
```

- Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the Fig. 2.8 (b) classes Y, Z and A are sibling classes.
- Inheritance can also extend over several "generations" of classes. This is shown in the Fig. 2.8 (b), where class B is a subclass of class A which is itself a subclass of class X. In this case, class B is considered to be a subclass of class X, even though it is not a direct subclass. This whole set of classes forms a small class hierarchy.

For example:

- Let's look at an example of Vehicles. Here Car, Truck, and Motorcycle are types of vehicles, they would be represented by subclasses of the Vehicle class, as shown in Fig. 2.9.

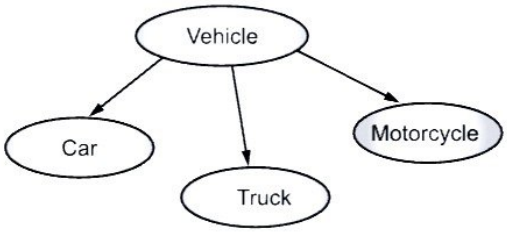


Fig. 2.9: Subclasses of the Vehicle Class

- The Vehicle class would include instance variables such as registrationNumber, owner and instance methods such as transferOwnership(). These are variables and methods common to all vehicles.
- The Car class might add an instance variable numberOfDoors, the Truck class might have numberOfAxles, and the Motorcycle class could have a boolean variable hasSidecar.
- The declarations of these classes in a Java program would look.

```
class Vehicle
{
   int registrationNumber;
   Person owner;  // (Assuming that a Person class has been defined!)
   void transferOwnership(Person newOwner)
   {
       ........
   }
   ........
}
class Car extends Vehicle
{
   int numberOfDoors;
   ........
}
class Truck extends Vehicle
{
   int numberOfAxles;
   ........
}
class Motorcycle extends Vehicle
{
   boolean hasSidecar;
   ........
}
```

2.2.1 Use of 'super' Keyword [W-18]

- A subclass inherits the accessible data fields and methods from its superclass, but the constructors of the superclass are not inherited in the subclass. They can only be invoked from constructors of the subclass using the keyword 'super'.
- When invoking a superclass version of an overridden method the super keyword is used.
- 'super' keyword is used by subclass to refer its immediate superclass.

Program 2.5: Program using the super keyword.

```
class Animal
{
   public void move()
   {
      System.out.println("Animals can move ...");
   }
}
class Cat extends Animal
{
   public void move()
   {
      super.move(); // invokes the super class method
      System.out.println("Cats can Walk and Run");
   }
}
public class TestCat
{
   public static void main(String args[])
   {
      Animal b = new Cat(); // Animal reference but Cat object
      b.move(); //Runs the method in Cat class
   }
}
```

Output:

```
Animals can move…
Cats can Walk and Run
```

- There are two different forms of using super.
  1. 'super' calls the superclass constructor.
  2. 'super' can access members of the superclass those have been hidden by members of a subclass.

1. 'super' calls Superclass Constructor:

- In the constructor of subclass, the keyword super can be used to invoke the constructor method of the superclass.
- A subclass can call the constructor defined by superclass by using the following form of the 'super' keyword.

```
class subclass extends superclass
{
   subclass(args)
   {
      super(arg_list);
      //Statements;
   }
   ................
}
```

- The use of super() in this form must follow the following conditions:
  (i) super() must always be the first statement in subclass constructor.
  (ii) The order and type of parameters specified in super() must matches with the parameters in constructor of superclass.

Program 2.6: Use of super keyword to access superclass constructor.

```
class Primary
{
   int cal;                  //declaration1
   Primary(int a)
   {
      cal = a;
   }
   void show()
   {
      System.out.println("Super class cal: "+cal);
   }
}
class Secondary extends Primary
{
   int cal;                  //declaration2
   Secondary(int x,int y)    //statement1
   {
      super(y);              //statement2
      cal = x;               //statement3
   }
```

```
    void display()
    {
        System.out.println("Sub class cal: "+cal);
    }
}
public class SuperUse2
{
    public static void main(String args[])
    {
        Secondary s = new Secondary(10,20);
        s.show();
        s.display();
    }
}
```

Output:

```
Super class cal: 22
Sub class cal: 15
```

- The super class 'Primary' has been added with a constructor of one parameter. This constructor has been called in 'sub class' constructor in statement2 by passing one parameter to super(). If the super class contains more number of constructors all that can be called using super keyword in 'sub class' constructor. Statement that contains 'super' must be a first statement in the 'sub class' constructor.

2. 'super' Access Member of Superclass:

- Any member of immediate superclass can be referred in subclass using super keyword.
- **form/syntax:**

```
super.member_name
```

- Use of super keyword is somewhat like this. But difference is that super always refers to the superclass of the class in which it is used, whereas this refers to the members of that class itself.
- This form of super keyword is most useful when member in subclass hides the member of superclass due to similar names.
- Consider the following program, the variable name n is used in superclass as well as in subclass. To access a variable n of superclass in subclass, we use the super keyword.

Program 2.7: Program for super refers member of superclass.

```
class SuperClass
{
    int n;
}
class Subclass extends SuperClass
{
    int n;        //This hides n of superclass
```

```
        SubClass (int x, int y)
        {
            if (x > y)
            {
                Super.n = x; n = y;
            }
            else
            {
                Super.n = y; n = x;
            }
        }
        void display()
        {
            System.out.println("n is SuperClass=" + super.n);
            System.out.println("n is SubClass=" + n);
        }
    }
    class TestSuper2
    {
        pubic static void main(String args[])
        {
            //create object of subclass
            SubClass myobj = new SubClass(10,4);
            //call display method
            myobj.display();
        }
    }
```

Output:

```
n in SuperClass = 10
n in SubClass = 4
```

- In similar way, super keyword can also be used to call methods of superclass that are hidden by subclass due to same names. This is called as Method Overriding.

2.3 METHOD OVERRIDING AND RUNTIME POLYMORPHISM

2.3.1 Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as Method Overriding.

- Method overriding is an example of runtime polymorphism. In method overriding, a method from subclass is overridden by a method with the same signature as that of in the superclass, the reference type is checked during compile time. During runtime, JVM checks the object type and would run the method that belongs to that particular object.

Usage of Method Overriding:

1. Method overriding is used to provide specific implementation of a method that is already provided by its super class.
2. Method overriding is used for run-time polymorphism.

- In short, redefining a super class method in a sub class is called Method Overriding.

Rules for Method Overriding:

1. The method signature i.e., method name, parameter list and return type have to match exactly.
2. The overridden method can widen the accessibility but not narrow it, i.e. if it is private in the base class, the child class can make it public but not vice versa.

Example:

- Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, HDFC, ICICI and SBI banks could provide 8%, 7% and 9% rate of interest respectively.

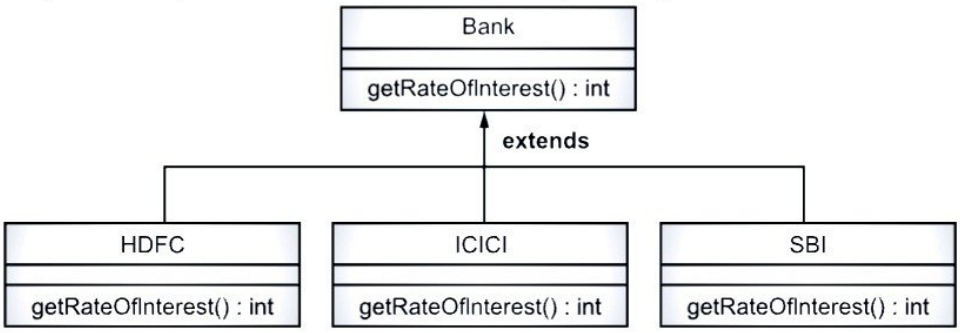


Fig. 2.10: Example of Method Overriding

Program 2.8: Program for method overriding.

```
class Bank
{
   int getRateOfInterest()
   {
      return 0;
   }
}
class HDFC extends Bank
{
   int getRateOfInterest(){return 8;}
}
```

```
class ICICI extends Bank
{
   int getRateOfInterest(){return 7;}
}
class SBI extends Bank
{
   int getRateOfInterest(){return 9;}
}
public class Test2
{
   public static void main(String args[])
   {
      HDFC s=new HDFC();
   ICICI i=new ICICI();
   SBI a=new SBI();
      System.out.println("HDFC Rate of Interest: "+s.getRateOfInterest());
      System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
      System.out.println("SBI Rate of Interest: "+a.getRateOfInterest());
   }
}
```

Output:

```
HDFC Rate of Interest: 8
ICICI Rate of Interest: 7
SBI Rate of Interest: 9
```

2.3.2 Runtime Polymorphism

- **Polymorphism in Java** is a concept by which different ways can we used to perform different action. Polymorphism implies many forms which are derived from two Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. Method overriding is an example of runtime polymorphism.
- Types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.
- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- We can perform polymorphism in Java by method overloading and method overriding.

- When reference variable of Parent class refers to the object of Child class, it is known as upcasting.
- In Program 3.19, we are creating two classes Bike and Platina. Platina class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at run-time.
- Since method invocation is determined by the JVM not compiler, it is known as Run-time Polymorphism.

Program 2.9: Program for run-time polymorphism.

```
class Bike
{
   void run()
   {
      System.out.println("running");
   }
}
class Platina extends Bike
{
   void run(){System.out.println("Running safely with 80km");}
   public static void main(String args[])
   {
       Bike b = new Platina();//upcasting
       b.run();
   }
}
```

Output:

```
Running safely with 80 km.
```

Program 2.10: Program for Runtime Polymorphism.

```
class Mexican_Dish
{
   void disp(){System.out.println("\n Inside Mexican Dish Class...");}
}
class Guacamole extends Mexican_Dish
{
   void disp()
   {
      System.out.println("\n WELCOME TO MEXICAN DISHES ...");
      System.out.println("\n Inside Guacamole Dish Class...");
   }
}
```

```
class Tacos extends Mexican_Dish
{
   void disp(){System.out.println("\n Inside Tacos Dish Class...");}
}
class Mexican_Pizza extends Mexican_Dish
{
   void disp(){System.out.println("\n Inside Mexican_Pizza Dish Class...");}
}
public class RunTime_Poly
{
   public static void main(String args[])
   {
      Mexican_Dish md;
      md=new Guacamole();
      md.disp();
      md=new Tacos();
      md.disp();
      md=new Mexican_Pizza();
      md.disp();
   }
}
```

Output:

```
WELCOME TO MEXICAN DISHES ...
Inside Guacamole Dish Class...
Inside Tacos Dish Class...
Inside Mexican_Pizza Dish Class...
```

2.4 USAGE OF 'final' KEYWORD [S-19]

- The word, 'final' is a keyword in Java Language.
- The 'final' keyword is used in several different contexts with the variable declaration, methods and classes as a modifier meaning that what it modifies cannot be changed in some sense.
- Any 'final' keyword when declared with variables, methods and classes specifically means:
  1. A final variable cannot be reassigned once initialized.
  2. A final method cannot be overridden.
  3. A final class cannot be extended.
- 'final' is a keyword which is used to prevent method overriding. This is possibly done by specifying 'final' as a modifier at the start of its declaration. The method whenever is declared as final, it cannot be overridden.

2.4.1 Use of Final Keyword Related to Variable

- If any variable in Java is declare using 'final' keyword then they are known as final variable.
- **Syntax:** `final datatype variableName;`
- When a variable is declared final, it is a constant which will not and cannot change.
- **Example:** `final PI=3.14;`
- The value of PI is declared as 'final' so, it cannot be changed in the course of the program.

Program 2.11: Program using 'final' variables.

```
public class FinalVar
{
    public static void main()
    {
        int x=10;
        final int y=20;
        System.out.println("x is:"+x);
        System.out.println("y is:"+y);
        x=30;
        y=40;
        System.out.println("x is:"+x);
        System.out.println("y is:"+y);
    }
}
```

Output:

```
error: cannot assign a value to final variable y
```

2.4.2 Use of Final Keyword Related to Methods

- A method that is declared final cannot be overridden in a subclass.
- If method of any class is defined using final keyword then they are known as final methods in Java because they cannot be redefined.
- The syntax is simple, just put the keyword 'final' after the access specifier and before the return type.
- **Syntax:**

```
access specifier final returntype methodname(arguments)
{
    //method body
}
```

- When you try to override method that has been declared final it will generate compile time error.

Program 2.12: Program for final with methods.

```
class FM
{
   final void disp()
   {
      System.out.println("\n Inside Temp Class ....");
   }
}
public class Temp1 extends FM
{
   void disp()
   {
      System.out.println("\n Inside Temp1 Class .... ");
   }
   public static void main(String args[])
   {
      Temp1 t= new Temp1();
      t.disp();
   }
}
```

Output:

```
error: disp() in Temp1 cannot override disp() in FM
void disp(){
     ^
overridden method is final
1 error
```

2.4.3 Use of Final keyword related to Classes

- If we want the class not be subclassed (or extended) by any other class, declare it final.
- In Java, if we define any class using 'final' keyword then this type of class is known as final class in Java.
- When classes declared as final, they cannot be extended. That is, any class can use the methods of a final class by creating an object of the final class and call the methods with the object(final class object).

  Syntax:

```
final class ClassName
{
   //Members
}
```

Example:

```
final class finalclass
{
    //Definitions of members ...
}
class errorclass extends finalclass
{
    //Definitions of members ...
}
```

- The above program segment contains error since we tried to extend a final class.

Program 2.13: Program using 'final' with classes.

```
import java.io.*;
final class Demo1
{
    public void display()
    {
        System.out.println("hi");
    }
}
public class Demo3 extends Demo1
{
    public static void main(String args[])
    {
        Demo1 d=new Demo1();
        d.display();
    }
}
```

Output:

```
error: cannot inherit from final Demo1
```

2.5 USE OF ABSTRACT CLASSES AND ABSTRACT METHODS

- An abstract class is a class in which one or more methods are declared, but not defined. That means the body of these methods are omitted. Such methods are called Abstract Methods.
- Abstract class cannot be instantiated. Class that is declared with abstract keyword, is known as Abstract Class in Java.

2.5.1 Abstract Classes

- Abstract class is a superclass that only defines a generalized form which will be shared by all of its subclasses.
- The purpose of abstract class is to function as base classes which can be extended by subclasses to create a full implementation.
- When a class contains one or more abstract methods, then it should be declared abstract. You must use abstract keyword to make a class abstract.

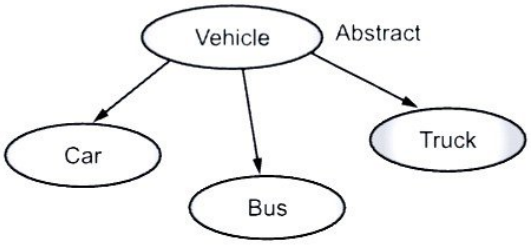


Fig. 2.11: Abstract Class

- An abstract class is used to provide abstraction.
- **Syntax:** `abstract class class_name{ }`
- The abstract methods of an abstract class must be defined in its subclass. We cannot declare abstract constructors or abstract static methods. Abstract class has both methods - abstract as well as non-abstract methods. Abstract class also has a member variables and constructors.

2.5.2 Abstract Methods

- An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.
- Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.
- It is the responsibility of the concrete subclass to implement all abstract method of superclass.

  In short, the methods which have only declaration and no method body in the class are called abstract methods.
- The abstract methods are declared using the keyword abstract. The abstract methods should be defined in the subclass. Abstract method does not have any body, it always ends with semicolon (;).

  Syntax: `abstract returntype method_name(arguments);`

  Example: `abstract void area();`

Program 2.14: Program using abstract class and methods. **[W-18, S-19]**

```
abstract class shape   //Abstract class
{
   double pi=3.14;
   abstract void area();            //Abstract method
   void display()
   {
      System.out.println("\n Non_abstract method of Class shape");
   }
}
class rectangle extends shape       //Sub class
{
   int l,b;
   rectangle(int x,int y)
   {
      l=x;
      b=y;
   }
   void area()  //Implementing abstract method
   {
      System.out.println("\n Area of Rectangle");
      System.out.println("Length="+ l);
      System.out.println("Breadth="+ b);
      System.out.println("Area  ="+ (l*b));
   }
}
class circle extends shape      //Sub class
{
   double r;
   circle(double x)
   {
      r=x;
   }
   void area()  //Implementing abstract method
   {
      System.out.println("\n Area of Circle");
      System.out.println("Radius="+ r);
      System.out.println("Area  ="+ (pi*r*r));
   }
}
```

```
/*  Main class */
public class abst
{
   public static void main(String args[])
   {
      shape s;
      rectangle r=new rectangle(10,5);
      s=r;   //Shape referenece rectangle
      s.area();
      circle c=new circle(2.5);
      c.area();
   }
}
```

Output:

```
Area of Rectangle
Length=10
Breadth=5
Area=50
Area of Circle
Radius=2.5
Area=19.625
```

- In above program, shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class, (i.e. hidden to the end user) and object of the implementation class is provided by the factory method.
- A factory method is the method that returns the instance of the class. We will learn about the factory method later.
- In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

Program 2.15: Program using abstract class and abstract methods.

```
abstract class Shape
{
   abstract void draw();
}
class Rectangle extends Shape
{
   void draw()
   {
      System.out.println("Drawing rectangle");
   }
}
```

```
class Circle1 extends Shape
{
    void draw()
    {
        System.out.println("Drawing circle");
    }
}
public class TestAbstraction1
{
    public static void main(String args[])
    {
        Shape s=new Circle1();
        s.draw();
    }
}
```

Output:

```
Drawing circle
```

2.6 INTERFACE

- An interface in Java is a blueprint of a class. It has static constants and abstract methods only.
- An interface is a way of describing what classes should do, without specifying how they should do it.
- There are mainly three reasons to use interface. They are given below:
  1. It is used to achieve fully abstraction.
  2. By interface, we can support the functionality of multiple inheritance.
  3. It can be used to achieve loose coupling.
- A class can implement more than one interface. In Java, an interface is not a class but a set of requirements for the class that we want to conform to the interface.
- All the methods of an interface are by default public. So, it is not required to use the keyword public when declaring a method in an interface.
- Interfaces can also have more than one method. Interfaces can also define constants but do not implement methods.

2.6.1 Defining Interface

- To support the concept of multiple inheritance, Java provides the alternate approach. This is known as Interface.
- Java Interface also represents IS-A relationship. Interface cannot be instantiated just like abstract class.

- Interface is similar to class which is collection of public static final variables (constants) and abstract methods.
- The interface is a mechanism to achieve fully abstraction in Java.
- An interface is defined like a class. Its general **form/syntax** is:

```
access_specifier interface InterfaceName
{
   Return-type Method-1(Parameters);
   Return-type Method-2(Parameters);
   Type variable-1=value;
   Type variable-2=value;
   .
   .
   Return-type Method-N(Parameters);
   Type variable-N=value;
}
```

- Following is the definition of an interface containing two constants and four abstract methods.

```
interface conversions
{
   double GM_TO_KG = 1000;
   double CM_TO_FT = 30;
   double gmtokg(double gm);
   double cmtoft(double cm);
   double kgtogm(double kg);
   double fttocm(double ft);
}
```

2.6.2 Implementing Interface

- The methods in an interface are always abstract, therefore it is necessary to create a class, which inherits the interface. One or more classes can implement the single interface.
- To make a class implement an interface, we must declare that the class intends to implement the given interface and we need to supply definitions for all methods using the implements keyword.
- When an interface has been defined, any of the classes can implement that interface.
- The general **form/syntax** of the class that includes an implements clause is:

```
class classname extends superclass implements interface
{
   //class body ...
}
```

Program 2.16: Program of interface implementation.

```
import java.io.*;
interface conversions
{
   double GM_TO_KG = 1000;
   double gmtokg(double gm);
   double kgtogm(double kg);
}
class convert implements conversions
{
   public double gmtokg(double gm)
   {
      return gm/GM_TO_KG;
   }
   public double kgtogm(double kg)
   {
      return kg*GM_TO_KG;
   }
}
public class ImplIface1
{
   public static void main(String args[])
   {
      convert ConvObj = new convert();
      conversions c;
      c = ConvObj;
      System.out.println("2000 gm =" + c.gmtokg(2000) + " kg ");
      System.out.println("50 kg = " + c.kgtogm(50) + " gm ");
   }
}
```

Output:

```
2000 gm = 2.0 kg
50 kg = 50000.0 gm
```

Partial Implementation of Interface:

- If a class implements an interface, but does not implement all the methods declared by that interface, then the class must be declared as abstract.

- For example, following class partially implementing the interface-conversions.

```
abstract class convert implements conversions
{
    public double gmtokg (double gm)
    {
        return gm/GM_TO_KG;
    }
}
```

- This class does not implement the method kgtogm(), so it is abstract.
- Now you cannot create objects of type convert class, since it is abstract, so you must define a subclass of convert class that implements the remaining methods in the interface.

Program 2.17: Program shows the partial implementation of an interface.

```
interface conversions
{
    double GM_TO_KG = 1000;
    double gmtokg (double gm);
    double kgtogm (double kg);
}
abstract class convert implements conversions
{
    public double gmtokg (double gm)
    {
        return gm / GM_TO_KG;
    }
}
class convert_sub extends convert implements conversions
{
    public double kgtogm (double kg)
    {
        return kg*GM_TO_KG;
    }
}
public class Implface2
{
    public static void main(String args[])
    {
```

```
        convert_sub ConvObj=new convert_sub();
        conversions c;
        c = ConvObj;
        System.out.println("2000 gm =" + c.gmtokg(2000) + " kg ");
        System.out.println("50 kg =" + c.kgtogm(50) + " gm ");
    }
}
```

Output :

```
2000 gm =2.0 kg
50 kg =50000.0 gm
```

2.7 RUNTIME POLYMORPHISM USING INTERFACE

- In this subclass overrides a method with the same signature as that of in its superclass and it implements the interface.

2.7.1 Extending Interfaces

- An interface can extend another interface, similarly to the way that a class can extend another class.
- The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
- When any class implements an extended interface, it must implement all the methods defined within the chain of inherited interfaces.
- Consider the following Program, an interface emp_show inherits all the members of an interface emp_info.
- The class emp is implementing an interface emp_show. It must define the code of all the methods declared in emp_info as well as emp_show.

Program 2.18: Program extending Interfaces.

```
interface emp_info
{
    int empid = 10;
    String empname = "Neetu";
    String empaddr = "Pune";
    void disp1();
}
interface emp_show extends emp_info
{
    void disp2();
}
```

```
class emp implements emp_show
{
   public void disp()
   {
      System.out.println("\n Inside Display Function .. ");
   }
   public void disp1()
   {
      disp();
      System.out.println("Empid="+empid);
      System.out.println("Name="+empname);
   }
   public void disp2()
   {
       disp();
      System.out.println("Name="+ empname);
      System.out.println("Address="+ empaddr);
   }
}
public class ImplIface3
{
   public static void main(String args[])
   {
      emp e = new emp();
      System.out.println("From Method disp1");
      e.disp1();
      System.out.println();
      System.out.println("From Method disp2");
      e.disp2();
   }
}
```

Output:

```
From Method disp1
Inside Display Function ..
Empid=10
Name=Neetu
From Method disp2
Inside Display Function ..
Name=Neetu
Address=Pune
```

Extending Multiple Interfaces:

- A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.
- The extends keyword is used once and the parent interfaces are declared in a comma-separated list.
- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

Fig. 2.12: Multiple Inheritance in Java

Program 2.19: Program to implementation of multiple inheritance.

```
class employee
{
   String ename;
   void get_ename(String name)
   {
      ename = name;
   }
   void print_ename()
   {
      System.out.println("Name = " + ename);
   }
}
class gross_sal extends employee
{
   float basic, da, hra, gsal;
   void calc_gross_sal(int b, int dapercent, int hrpercent)
   {
      basic = b;   //calculate DA & HRA
      da = basic* dapercent/100;
      hra = basic*hrpercent/100;
      gsal = basic + da + hra;
   }
```

```
    void print_gross_sal()
    {
        System.out.println("Gross sal = " + gsal);
    }
}
interface taxes
{
    float ptax = 200f ;
    float itaxpercent = 20f ;
    float calc_taxes();
}
class salary extends gross_sal implements taxes
{
    public float calc_taxes()
    {         //Method from interface
        return ptax + (gsal * itaxpercent/100);
    }
    void display()
    {      //Method not from interface
        float tax = calc_taxes();
        float nsal = gsal ;
        print_ename();
        print_gross_sal();
        System.out.println("taxes=" + tax);
        System.out.println("Net Sal = " + nsal);
    }
}
public class ImplIface6
{
    public static void main(String args[ ])
    {
        salary empsal = new salary();
        empsal.get_ename("Adhiraj");
        empsal.calc_gross_sal(9100,50,10);
        empsal.display();
    }
}
```

Output:

```
Name = Adhiraj
Gross sal = 14560.0
taxes=3112.0
Net Sal = 14560.0
```

2.7.2 Nested Interfaces

- An interface which is declared within another interface or class is known as Nested Interface.
- The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It cannot be accessed directly.

Points to Remember for Nested Interfaces:

- Nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.
- **Syntax** of nested interface which is declared within the interface:

```
interface interface_name
{
    ...
    interface nested_interface_name
    {
        ...
    }
}
```

- Syntax of nested interface which is declared within the class:

```
class class_name
{
    ...
    interface nested_interface_name
    {
        ...
    }
}
```

Program 2.20: Program declares the Nested Interface and how we can access it.

```
interface ShowableExe
{
    void show();
    interface Message
    {
        void msg();
    }
}
```

```
public class TestNestedInterface1 implements ShowableExe.Message
{
    public void msg()
    {
        System.out.println("Program for Nested Interface .....");
    }
    public static void main(String args[])
    {
        ShowableExe.Message message=new TestNestedInterface1();
        message.msg();
    }
}
```

Output:

```
Program for Nested Interface .....
```

2.8 PACKAGES

- Java is an Object Oriented programming language. Any Java programmer will quickly build large number of different classes for use in each application that they develop.
- Initially it may be tempting to store the classes in the same directory as the end application, but as the complexity of applications grows so will the number of classes. The answer is to organize the classes into packages.
- A Java package is a mechanism for organizing Java classes. Package is used in Java, in-order to avoid name conflicts and to control access of class, interface and enumeration etc.
- A package can be defined as, "a group of similar types of classes, interface, enumeration and sub-packages".

Defining a Package:

- A package is a collection of classes and interfaces.
- A package does not contain the source code of any type of Java file. Each Java source code file is a collection of one or more classes, or one or more interface with their members.
- After compilation, a separate class file is created for individual class or interface, either these are part of one Java file or stored in individual file.
- That class file (byte code) is placed in a particular related package using a package declaration.

2.8.1 Creating Packages [W-18, S-19]

- We can define package using 'package' keyword. There are two types of packages:
  1. **Built-in Package:** Existing built-in Java packages like java.lang, java.util, java.sql etc.
  2. **User-define Package:** Java package created by user to categorized classes, interface and enumeration.

- **Declaration Syntax:**

```
package <package name>;
```

- The package declaration statement must be the first statement of the source file. The package name is stored in the byte code (class file).
- The dot (.) notation is used to distinctively spot (identify) the package name in the package hierarchy. One naming convention for a package is that, package name is written in all lowercase. It will escape the confliction with classes and interfaces name.

Example:

```
package mypck;
```

- Here, mypck is the package name. Moreover a file belonging to same package can include the same package name.
- We can also create a hierarchy of packages, i.e. package within a package called subpackage for further classification and categorization of classes and interfaces.

```
package mypck.mypck2.mypck3;
```

- Here, mypck2 is a subpackage in package mypck, mypck3 is a subpackage in subpackage mypck2. If there is no package declaration in a source file, then the compiled file is stored in the default package (unnamed package), and that is the current working directory.

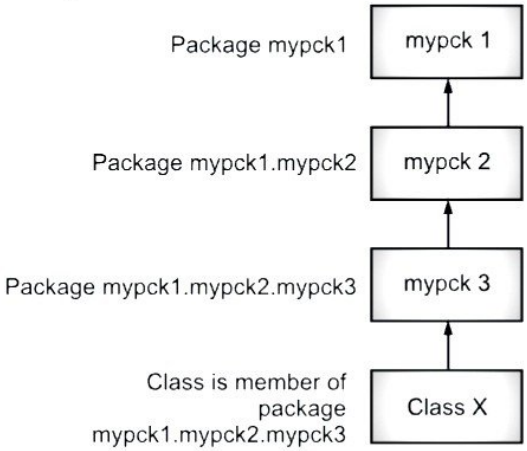


Fig. 2.13: Package Hierarchy

- In the above package hierarchy, class X is identified by the name mypckl.mypck2.mypck3.

2.8.2 Accessing and Using Packages [W-18]

- There are three ways to access the package from outside the package i.e., import package.*; import package.classname; fully qualified name.

1. Using packagename:

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Program 2.21: Program for package that import the packagename.

```
//Program save by X.java
package pkg;
public class X
{
   public void msg(){System.out.println("Hello, Take Care");}
}
//save by y.java
package mypkg;
import pkg.*;
class Y
{
   public static void main(String args[])
   {
      X obj = new X();
      obj.msg();
   }
}
```

Output:

```
Hello, Take Care
```

2. Using packagename.classname:

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

Program 2.22: Program Using packagename.classname.

```
//save by X.java
package pkg;
public class X
{
   public void msg(){System.out.println("Hello");}
}
//save by Y.java
package mypkg;
import pkg.X;
```

```
class Y
{
   public static void main(String args[])
   {
      X obj = new X();
      obj.msg();
   }
}
```

Output:

```
Hello
```

3. Using fully qualified name:

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Program 2.23: Program using fully qualified name.

```
//save by X.java
package pkg;
public class X
{
   public void msg() {System.out.println("Hello");}
}
//save by Y.java
package mypkg;
class Y
{
   public static void main(String args[])
   {
      pack.X obj = new pack.X();   //Using fully qualified name
      obj.msg();
   }
}
```

Output:

```
Hello
```

Program 2.24: Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. use the parent class constructor.

```
import java.util.Scanner;
class Person
{
   int pid;
   String pname;
   Person(int pid,String pname)
   {
      this.pid = pid;
      this.pname = pname;
   }
}
class Emp extends Person
{
   float esal;
   Emp(int id,String name,float esal)
   {
      super(id,name);       //parent constructor
      this.esal = esal;
   }
   void putdata()
   {
      System.out.println("Person Id = "+pid+"\nPerson Name
                                    =  "+pname+"\nPerson Salary "+esal);
   }
}
class Testconst
{
   public static void main(String[] args)
   {
      int p_id;
      float p_sal;
      String p_name;
      Scanner in = new Scanner(System.in);
      System.out.print("Enter the Person id : ");
```

```
        p_id = in.nextInt();
        System.out.print("Enter the Person Name :");
        p_name = in.next();
        System.out.print("Enter the Person Salary : ");
        p_sal = in.nextFloat();
        Emp e1 = new Emp(p_id,p_name,p_sal);
        e1.putdata();
    }
}
```

Output:

```
Enter the Person id : 10
Enter the Person Name :xxx
Enter the Person Salary : 9000
Person Id = 10
Person Name =  xxx
Person Salary 9000.0
```

Program 2.25: Example of Interface (using implements). Calculate Area of Circle and Square.

```
interface Area
{
    float pi = 3.14f;                //value declaration.
    float calArea(float x);
}
class Circle implements Area
{
    public float calArea(float x1)
    {
        float areacir = pi * x1 * x1;
        return areacir;
    }
}
class Square implements Area
{
    public float calArea(float x1)
    {
        float areasq = x1 * x1;
        return areasq;
    }
}
```

```
class ExInterface
{
   public static void main(String[] args)
   {
      Area a;                     // interface object.
      a = new Circle();           // object of circle.
      float cir = a.calArea(1.2f);
      System.out.println("Area of Circle : " +cir);
      a = new Square();           // object of square.
      float sq = a.calArea(2.5f);
      System.out.println("Area of square : " +sq);
   }
}
```

Output:

```
Area of Circle : 4.5216007
Area of square : 6.25
```

Program 2.26: Define an abstract class "Staff" with members name and address. Define two sub-classes of this class – "FullTimeStaff" (department, salary) and "PartTimeStaff" (number-of-hours, rate-perhour). Define appropriate constructors. Create n objects which could be of either FullTimeStaff or PartTimeStaff class by asking the user's choice. Display details of all "FullTimeStaff" objects and all "PartTimeStaff" objects.*/

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
abstract class Staff
{
   String name,address;
}
class FullTimeStaff extends Staff
{
   String department;
   double salary;
   public void accept() throws IOException
   {
      System.out.println("Enter the name, address, department and salary:");
      BufferedReader br=new BufferedReader
                                    (new InputStreamReader(System.in));
      name=br.readLine();
```

```
        address=br.readLine();
        department=br.readLine();
        salary=Double.parseDouble(br.readLine());
    }
    public void display()
    {
        System.out.println("Name: "+name);
        System.out.println("Address: "+address);
        System.out.println("Department: "+department);
        System.out.println("Salary: "+salary);
        System.out.println("----------------------");
    }
}
class PartTimeStaff extends Staff
{
    int hours, rate;
    public void accept() throws IOException
    {
        System.out.println("Enter the name, address, No of working hours and
        rate per hour: ");
        BufferedReader br=new BufferedReader
                                        (new InputStreamReader(System.in));
        name=br.readLine();
        address=br.readLine();
        hours=Integer.parseInt(br.readLine());
        rate=Integer.parseInt(br.readLine());
    }
    public void display()
    {
        System.out.println("Name: "+name);
        System.out.println("Address: "+address);
        System.out.println("No of Working Hours: "+hours);
        System.out.println("Rate per hour: "+rate);
        System.out.println("----------------------");
    }
}
```

```
public class sample
{
   public static void main(String [] args) throws IOException
   {
      int i;
      System.out.println("Select Any One: ");
      BufferedReader br=new BufferedReader
                                    (new InputStreamReader(System.in));
      System.out.println("1.Full Time Staff");
      System.out.println("2.Part Time Satff");
      int ch=Integer.parseInt(br.readLine());
      switch(ch)
      {
         case 1:
         System.out.println("Enter the number of Full Time Staff: ");
         int n=Integer.parseInt(br.readLine());
         FullTimeStaff [] l=new FullTimeStaff[n];
         for(i=0;i<n;i++)
         {
            l[i]=new FullTimeStaff();
            l[i].accept();
         }
         for(i=0;i<n;i++)
         {
            l[i].display();
         }
         break;
         case 2:
         System.out.println("Enter the number of Part Time Staff:");
         int m=Integer.parseInt(br.readLine());
         PartTimeStaff [] h=new PartTimeStaff[m];
         for(i=0;i<m;i++)
         {
            h[i]=new PartTimeStaff();
            h[i].accept();
         }
```

```
            for(i=0;i<m;i++)
            {
                h[i].display();
            }
            break;
        }
    }
}
```

Output:

```
Select Any One:
1.Full Time Staff
2.Part Time Satff
1
Enter the number of Full Time Staff:
2
Enter the name, address, department and salary:
pragati
pune
computer
400000
Enter the name, address, department and salary:
nirali
solapur
it
500000
Name: pragati
Address: pune
Department: computer
Salary: 400000.0
----------------------
Name: nirali
Address: solapur
Department: it
Salary: 500000.0
----------------------
```

Summary

- Object Oriented Programming (OOP) allows you to create new class based on a class that has been already defined using inheritance.
- Inheritance is one of the most dominant and vital feature of OOP because it supports the hierarchical classifications, reusability of class; and defined to specialization. Types: Single Inheritance, Multilevel Inheritance, Hierarchical Inheritance, Multiple Inheritance, Hybrid Inheritance.
- Object oriented programming allows classes to inherit commonly used state and behavior from other classes.
- A subclass inherits the accessible data fields and methods from its superclass, but the constructors of the superclass are not inherited in the subclass. They can only be invoked from constructors of the subclass using the keyword super.
- 'super' keyword is used by subclass to refer its immediate superclass. The word, 'final' is a keyword in Java Language.
- The 'final' keyword is used in several different contexts with the variable declaration, methods and classes as a modifier meaning that what it modifies cannot be changed in some sense.
- If we want the class not be sub-classed (or extended) by any other class, declare it final.
- In Java, if we define any class using final keyword then this type of class is known as final class in Java. Abstract class is a superclass that only defines a generalized form which will be shared by all of its subclasses.
- An interface in Java is a blueprint of a class. It has static constants and abstract methods only.
- An interface is a way of describing what classes should do, without specifying how they should do it. Interface is similar to class which is collection of public static final variables (constants) and abstract methods.
- The interface is a mechanism to achieve fully abstraction in Java. If subclass (child class) has the same method as declared in the parent class, it is known as method overriding.
- Polymorphism in Java is a concept by which we can perform a single action by different ways. Polymorphism is derived from two Greek words poly and morphs. The word "poly" means many and "morphs" means forms. Polymorphism means many forms.

Check Your Understanding

1. Which of these keywords must be used to inherit a class?
   (a) super (b) this
   (c) extent (d) extends
2. Which of these is correct way of inheriting class A by class B?
   (a) class B + class A {} (b) class B inherits class A {}
   (c) class B extends A {} (d) class B extends class A {}
3. Which of these keywords is used to refer to member of base class from a subclass?
   (a) upper (b) super
   (c) this (d) none of the mentioned
4. A class member declared protected becomes member of subclass of which type?
   (a) public member (b) private member
   (c) protected member (d) static member
5. Which of the following is not type of inheritance?
   (a) Single inheritance (b) Double inheritance
   (c) Hierarchical inheritance (d) Multiple inheritance
6. Which of the following is used for implementing inheritance through interface?
   (a) inherited (b) using
   (c) extends (d) implements
7. What would be the result if class extends two interfaces and both have method with same name and signature?
   (a) Runtime error
   (b) Compile time error
   (c) Code runs successfully
   (d) First called method is executed successfully
8. Which of the following is true?
   1. A class can extend more than one class.
   2. A class can extend only one class but many interfaces.
   3. An interface can extend many interfaces.
   4. An interface can implement many interfaces.
   5. A class can extend one class and implement many interfaces.

   (a) 1 and 2 (b) 2 and 4
   (c) 3 and 5 (d) 3 and 4

9. A _____ is a collection of classes and interfaces.
   (a) Object (b) Package
   (c) Inheritance (d) Method
10. Final keyword in Java is used with _____ .
   (a) class attributes (b) class
   (c) class functions (d) All of the above
11. try, catch, finally, throw, and throws keywords in java are related to _____.
   (a) function handling (b) string handling
   (c) exception handling (d) extension

Answers

1. (d)	2. (c)	3. (b)	4. (b)	5. (d)	6. (d)	7. (b)	8. (c)	9. (b)	10. (d)	11. (c)

Practice Questions

Q.I Answer the following questions in short:

1. What is inheritance?
2. What are the types of inheritance?
3. What do you mean by abstract classes and methods?
4. How to override methods in Java?
5. What is final class?
6. What is super class?
8. Give example of hierarchical inheritance.
9. Why abstract class is used?
10. Which keyword is used to define abstract class?
11. By default interface is public and abstract. State True or False.

Q.II Answer the following questions:

1. When method overriding occurs?
2. Explain final keyword giving suitable examples.
3. How to extending Interface? Explain with example.
4. How to create interface? Explain with example.
5. Explain nested interface.
6. Explain extends keyword with example.
7. Write short notes on: Runtime polymorphism using interface.

8. Explain method overriding and method overloading with suitable examples.
9. Explain usage of packages by giving suitable examples.
10. Explain use of super keyword with suitable examples.

Q.III Define the terms:

1. Inheritance
2. Interface
3. Packages
4. Superclass
5. Subclass
6. Abstract class
7. Runtime polymorphism
8. Method overriding

Previous Exam Questions

Winter 2018

1. What is a use of super keyword? **[1 M]**

Ans. Refer to Section 2.2.1.

2. Which class is super class of all classes in Java? **[1 M]**

Ans. Refer to Section 2.2.

3. What is inheritance? Explain any two types of inheritance with suitable example. **[5 M]**

Ans. Refer to Section 2.1.1, 2.1.3.

4. How to create and access a package in Java? **[3 M]**

Ans. Refer to Section 2.8.1, 2.8.2.

5. Write a java program to create abstract class shape with methods calculate-area and display-area. Derive two classes Circle (radius), Rectangle (Length, width) from it. Calculate area and display the area of all classes. **[4 M]**

Ans. Refer to Program 2.13.

Summer 2019

1. A package is a collection of _____. **[1 M]**

(i) Classes (ii) Interfaces
(iii) Editing tools (iv) Classes and interfaces

Ans. (iv)

2. What is use of final keyword? **[1 M]**

Ans. Refer to Section 2.4.

3. What is inheritance? Explain hierarchical and multiple inheritance with an example. **[5 M]**

Ans. Refer to Section 2.1.1, 2.1.3.

4. Write a short note on built-in package and user define package. **[3 M]**

Ans. Refer to Section 2.8.1.

5. Write a java program to create abstract class person derived two classes Employee and Worker from it. Use proper method to accept and display for the same. Employee(eno, ename, address), similar fields are worker. **[4 M]**

Ans. Refer to Program 2.13.

■■■

3... Collection, Exception Handling and I/O

Learning Objectives...

- To learn Collection Framework.
- To knowing various Collection Interfaces and Classes.
- To study Class Vector and Hashtable.
- To understand basic concepts of Exception Handling in Java.
- To learn I/O in Java.

3.1 COLLECTION [S-19]

- Java.util is an important package which contains a large collection of classes and interfaces which supports a broad range of functionality. It contains Java's most powerful subsystem collections framework.
- Collection Framework is a sophisticated hierarchy of classes and interfaces for managing groups of objects.

Table 3.1: List of classes from java.util package

AbstractCollection	EventObject	Random
AbstractList	FormattableFlags	ResourceBundle
AbstractMap	Formatter	Scanner
AbstractQueue	GregorianCalendar	ServiceLoader (Added by Java SE 6.)
AbstractSequentialList	HashMap	SimpleTimeZone
AbstractSet	HashSet	Stack
ArrayDeque	Hashtable	StringTokenizer
ArrayList	IdentityHashMap	Timer
Arrays	LinkedHashMap	TimerTask

contd. ...

BitSet	LinkedHashSet	TimeZone
Calendar	LinkedList	TreeMap
Collections	ListResourceBundle	TreeSet
Currency	Locale	UUID
Date	Observable	Vector
Dictionary	PriorityQueue	WeakHashMap
EnumMap	Properties	
EnumSet	PropertyPermission	
EventListenerProxy	PropertyResourceBundle	

Table 3.2: List of interfaces defined by java.util package

Collection	List	Queue
Comparator	ListIterator	RandomAccess
Deque	Map	Set
Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap	SortedSet
Formattable	NavigableSet	
Iterator	Observer	

- Java Collections is the standard way of groups of objects that are used by your program. Collections were not part of the initial Java release. Before Collections framework, Java had adhoc classes such as Dictionary, Vector, Stack and Properties for manipulating groups of objects.
- Collections altered the architecture of many utility classes. But it did not deprecate any of the previously used classes.

3.2 WRAPPER CLASSES

- Java's distinction between primitive types and reference types dilutes its object orientation.
- A"wrapper" class for each primitive type is provided in java.lang package. These wrapper classes serve two roles:
  1. They can be used to wrap a primitive type values within an object. This objectifies the primitive value and adds a number of useful methods that can act on that value. The state of the object is defined solely by the value of the primitive type that it wraps.
  2. They provide services or information about the primitive type they wrap. For example: The minimum and maximum values that the primitive type can represent are available.

- Conversions to and from the primitive type's human-readable string representations can be performed.
- Following table shows lists all of Java's wrapper classes. We will focus on the Integer class, but the other wrapper classes provide similar functionality for the types they wrap. Some useful Integer methods are shown in Table.

Table 3.3: List of Wrapper classes

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

- **Some methods of the integer class:**
  1. **`static int parseInt(String s)`:** A class method that converts a string representation of an integer to an int.

     For example, `int value = Integer.parseInt("256");`

     Note that, the following statement is illegal:

     ```
     int value = "256"; // Illegal!
     ```

     Since, a String object reference is not assignment compatible with an int.
  2. **`static int parseInt(String s, int b)`:** This overloaded method works with different number bases. For example,

     ```
     int value = Integer.parseInt("101", 2);
     ```

     Assigns the value five to value, since 1012 = 510.

     ```
     static Integer valueOf(String s)
     ```

     Like parseInt(), but returns a reference to a new Integer object, not a primitive int value.
  3. **`static Integer valueOf(String s, int b)`:** As with parseInt(), an overloaded version that deals with different number bases.
  4. **`int intValue()`:** Returns the primitive int value wrapped by this Integer object. Similar methods exist for the other primitive numeric types.
  5. **`String toString()`:** Returns the String representation the wrapped int value.
  6. **`static String toString(int i)`:** A class method that returns the String representation the value of i.

Program 3.1: Program using Wrapper Classes.

```
public class IntegerTest
{
    public static void main(String [] args )
    {
        Integer i1 = new Integer("500"); // Make it from a string
        Integer i2 = new Integer (200); // Make it from an int
        int i;
        i = i1.intValue ();
        System.out.println("i = " + i); // Prints 500
        i = i2.intValue ();
        System.out.println("i = " + i); // Prints 200
        i = Integer.parseInt("100");
        System.out.println("i = " + i); // Prints 100
        i1 = Integer.valueOf("600");
        i = i1.intValue ();
        System.out.println("i = " + i); // Prints 600
        i = Integer.valueOf("700"). intValue ();
        System.out.println("i = " + i); // Prints 700
    }
}
```

3.3 INTRODUCTION TO THE COLLECTION FRAMEWORK

- The Collections Framework defines several interfaces. Collections interfaces determine the fundamental nature of Collection classes.
- **Goals of Collection Framework:**
  - Framework had to be high performance. Implementations for fundamental collections are highly efficient.
  - Framework had to allow different types of collections to work in a similar manner.
  - Extending and / or adapting a collection had to be easy. Entire Collection Framework is built upon a set of standard interfaces. Java has provided several implementations of these interfaces like LinkedList, HashSet, TreeSet etc. But you can implement your own Collection as well.
  - Java provides mechanism for integrating standard arrays into the Collections Framework.
- Algorithm is another important part of Collection mechanism. They provide standard means of manipulating Collections.

- Iterator is also very closely associated with Collections Framework. Iterator interface offers a standard way of accessing elements within a collection.
- Framework also defines several map interfaces and classes. Map stores key and value pairs. Maps cannot be called as Collections though they come under Collections Framework. But you can obtain Collection view of Map.

3.4 COLLECTION CLASSES

- Collection class is a utility class used to work with Java Collections.
- Java Collection Interfaces are implemented by set of standard collection classes.
- Java.util.Collections class directly extend the object class
- Standard Collection classes are summarized below.

Table 3.4: Standard Collection classes

Class	Explanation
AbstractCollection	Implements Collection interface.
AbstractList	It extends AbstractCollection and implements List interface.
AbstractQueue	It extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	Extends AbstractList for use by a collection that uses sequential access of its elements.
LinkedList	Implements a linked list by extending AbstractSequentialList.
ArrayList	Implements a dynamic array by extending AbstractList.
ArrayDeque	Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface.
AbstractSet	Extends AbstractCollection and implements most of the Set interface.
EnumSet	Extends AbstractSet for use with enum elements.
HashSet	Extends AbstractSet for use with a hash table.
LinkedHashSet	Extends HashSet to allow insertion-order iterations.
PriorityQueue	Extends AbstractQueue to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends AbstractSet.

3.5 LIST- ARRAYLIST, LINKEDLIST AND VECTOR

- List interface extends Collection. This interface has a behavior that stores a sequence of elements. Elements can be inserted or accessed by their position in the list.
- A list may contain duplicate elements. In addition to the methods defined by Collection interface, List defines its own methods. It represents the ordered collection, (This order refers to the order in which items are added).

Table 3.5: Methods defined by List interface

Method	Explanation
`void add(int index, E obj)`	It adds `obj` into the invoking list at the given index passed in index parameter. Any previous elements at or after the point of insertion are shifted up. Thus, elements are not overwritten.
`E get(int index)`	Returns the object `E` stored at the specified index within the invoking collection.
`int indexOf(Object obj)`	Returns the index of the first instance of `obj` in the invoking list. If `obj` is not an element of the list, –1 is returned.
`ListIterator<E> listIterator( )`	Returns an iterator to the beginning of the invoking list.
`E remove(int index)`	Removes the element at the specified position `index` from the invoking list and returns the removed element. The resulting list is compressed. This means, the indexes of subsequent elements are decremented by one.
`E set(int index, E obj)`	Assigns `obj` to the position specified by index within the invoking list.

3.5.1 ArrayList

- ArrayList class extends AbstractList and implements the List interface. It supports dynamic arrays that can grow as and when needed.

Table 3.6: Difference between Array and ArrayList

Array	ArrayList
It is a dynamically created object.	It is a class of Java Collections framework. It has classes like Vector, HashTable, and HashMap.
Static in size.	Dynamic in size.
Fixed-length data structure.	Variable-length data structure.
It is mandatory to provide the size of an array while initializing it directly or indirectly.	ArrayList without specifying its size. Java creates ArrayList of default size.
It performs faster.	Slow down the performance.

contd. ...

It stores objects and primitives type.	It converts primitive type to object.
for loop or for each loop to iterate over an array.	Iterator to iterate over ArrayList.
We cannot use generics along with array because it is not a convertible type of array.	ArrayList allows us to store only generic/ type, that's why it is type-safe.
Array provides a length variable which denotes the length of an array.	ArrayList provides the size() method to determine the size of ArrayList.
We can add elements in an array by using the assignment operator.	Java provides the add() method to add elements in the ArrayList.
It is multi-dimensional.	It is single-dimensional.

ArrayList Constructors:

1. **`ArrayList()`:** An empty array list.
2. **`ArrayList(Collection c)`:** Array list with the elements of collection c.
3. **`ArrayList(int capacity)`:** Initial capacity of an array.

Program 3.2: The following program illustrates the use of ArrayList. We firstly create an object of ArrayList class and adds items to the array list using add() method. We can remove an item from the list by calling remove() method. We get the iterator with the help of iterator() method. Iterator iterates through the array list and display items from the list one by one using next() and hasNext() methods. **[S-19]**

```
File: ArraylistDemo.java
import java.util.*;
class ArraylistDemo
{
   public static void main(String args[])
   {
      ArrayList arraylist = new ArrayList();
      arraylist.add("Item 3");
      arraylist.add("Item 4");
      arraylist.add("Item 5");
      arraylist.add("Item 6");
      arraylist.add("Item 0");
      arraylist.add("Item 2");
      arraylist.add(1, "Item 1");
      System.out.println("\nUsing the add method\n");
      System.out.println(arraylist);
```

```
        arraylist.remove("Item 5");
        System.out.println(arraylist);
        System.out.println("\nUsing the Iterator interface");
        String s;
        Iterator e = arraylist.iterator();
        while (e.hasNext())
        {
            s = (String)e.next();
            System.out.println(s);
        }
    }
}
```

Output:

```
Using the add method
[Item 3, Item 1, Item 4, Item 5, Item 6, Item 0, Item 2]
[Item 3, Item 1, Item 4, Item 6, Item 0, Item 2]
Using the Iterator Interface
Item 3
Item 1
Item 4
Item 6
Item 0
Item 2
```

- If we want to insert the elements in arraylist of primitive data types, then wrapper class is used.

  For example, `arraylist.add(new Integer (10));` will insert integer 10 in arraylist.
- Increase the capacity of storing objects into ArrayList by calling `ensureCapacity(int capacity)` method.
- Call `trimToSize()`method which trims the capacity of this ArrayList instance to be the list's current size.
- To obtain an array from ArrayList, call `toArray()` method. Below is the format for calling `toArray()` method.

  ```
  T[] toArray(T array[]);
  //T is the type of array, the method returns.
  ```
- For example,

  ```
  Integer intAr[] = new Integer[al.size()];
  intAr = al.toArray(intAr);
  ```

3.5.2 LinkedList

- Linked List is Collection framework present in java.util.package. In this data structure every element is separate object with data and address. Each element i.e. nodes are linked using pointers and addresses.
- A linear collection supports element insertion and removal at both ends. The name deque is short implies "double ended queue". Deque implementations have no fixed limits on the number of elements they may contain.

```
public interface Deque<E>
extends Queue<E>
```

- LinkedList class extends AbstractSequentialList and implements List, Queue and Deque interfaces. It Provides a LinkedList data structure.

LinkedList Constructors:

1. **`LinkedList()`:** Constructs an empty list.
2. **`LinkedList(Collection<? extends E> c)`:** Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

- Because LinkedList implements Deque interface, you can access all the methods defined by Deque like `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, `removeLast()` etc.
- LinkedList also implement List interface, so it has access to all the methods defined by this List interface.

Program 3.3: The following program illustrates the use of class LinkedList. The following program stores the contents in the linked list data structure. Program adds items to the linkedlist. Items can be removed from the linked list by calling remove() method. It uses set() method to update the items from the given list.

```
File: LinkedlistDemo.java
import java.util.*;
public class LinkedlistDemo
{
    public static void main(String args[])
    {
        LinkedList linkedlist1 = new LinkedList();
        linkedlist1.add("Item 2");
        linkedlist1.add("Item 3");
        linkedlist1.add("Item 4");
        linkedlist1.add("Item 5");
        linkedlist1.addFirst("Item 0");
        linkedlist1.addLast("Item 6");
```

```
        linkedlist1.add(1, "Item 1");
        System.out.println(linkedlist1);
        linkedlist1.remove("Item 6");
        System.out.println(linkedlist1);
        linkedlist1.removeLast();
        System.out.println(linkedlist1);
        System.out.println("\nUpdating linked list items");
        linkedlist1.set(0, "Red");
        linkedlist1.set(1, "Blue");
        linkedlist1.set(2, "Green");
        linkedlist1.set(3, "Yellow");
        linkedlist1.set(4, "Purple");
        System.out.println(linkedlist1);
    }
}
```

Output:

```
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5, Item 6]
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5]
[Item 0, Item 1, Item 2, Item 3, Item 4]
Updating linked list Items
[Red, Blue, Green, Yellow, Purple]
```

3.5.3 Vector

- Vector implements dynamic array. It is similar to ArrayList with two major differences. Vector class is re-engineered to extend AbstractList and it implements List interface. It is also re-engineered to implement Iterable interface. Means, Vector can have its elements traversed through for loop. Vector is fully compatible with collections.

Table 3.7: Difference between ArrayList and Vector

ArrayList	Vector
1. It's faster since it's not synchronized.	It's slower since it's synchronized.
2. If the number of elements exceeds from its capacity 50% of current array size is increased.	If the total number of elements exceeds than its capacity it increases 100% of array size.
3. Not a legacy class.	Legacy class.
4. It uses the Iterator interface to traverse the elements.	It can use the Iterator interface or Enumeration interface to traverse the elements.

Vector Constructors:

1. `Vector()`: Default vector with initial size 10.
2. `Vector(int size)`: Vector whose initial capacity is specified by size.
3. `Vector(int size, int increment)`: Vector whose initial capacity is specified by size and increment is specified by increment. Increment is the number of elements allocated each time a vector is resized in upward direction. Constructors which do not accept increment, will resize vector with capacity 10 by default in upward direction.
4. `Vector(Collection<?extends E> c)`: Vector that contains the elements of collection c.

- Some of the important methods defined by Vector class are summarized in the following table.

Table 3.8: Legacy methods defined by Vector

Method	Explanation
`void addElement(E element)`	The object specified by element is added to the vector.
`int capacity()`	Returns the capacity of the vector.
`Object clone( )`	Returns a duplicate of the invoking vector.
`boolean contains(Object element)`	Returns true if elements is contained by the vector, and returns false if it is not.
`E elementAt(int index)`	Returns the element at the given location specified by index.
`Enumeration<E> elements( )`	Returns an enumeration of the elements in the vector.
`E firstElement( )`	Returns the first element in the vector.
`int indexOf(Object element)`	Returns the index of the first occurrence of element. If the object is not in the vector, –1 is returned.
`void insertElementAt(E element, int index)`	Adds element to the vector at the location specified by index.
`boolean isEmpty( )`	Returns true if the vector is empty, and returns false if it contains one or more elements.
`int lastindexOf(Object element)`	Returns the index of the last occurrence of element. If the object is not in the vector, –1 is returned.

contd. ...

`int lastindexOf(Object element, int start)`	Returns the index of the last occurrence of element before start. If the object is not in that portion of the vector, –1 is returned.
`void removeAllElements( )`	Empties the vector. After this method executes, the size of the vector is zero.
`void setElementAt(E element, int index)`	The location specified by index is assigned element.
`void setSize(int size)`	Sets the number of elements in the vector to size. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, null elements are added.
`int size( )`	Returns the number of elements currently in the vector.
`String to String( )`	Returns the string equivalent of the vector.
`void trimToSize( )`	Sets the vector's capacity equal to the number of elements that it currently holds.

Program 3.4: The following program demonstrates Vector class. A program uses a vector to store various types of numeric objects. It demonstrates several legacy methods defined by the Vector class. **[W-18, S-19]**

```
import java.util.*;
public class VectorDemo
{
    public static void main(String args[])
    {
        Vector vector = new Vector(5);
        System.out.println("Size: " + vector.size());
        System.out.println("Capacity: " + vector.capacity());
        vector.addElement(new Integer(0));
        vector.addElement(new Integer(1));
        vector.addElement(new Integer(2));
        vector.addElement(new Integer(3));
        vector.addElement(new Integer(4));
        vector.addElement(new Integer(5));
        vector.addElement(new Integer(6));
```

```
        vector.addElement(new Integer(5));
        vector.addElement(new Integer(6));
        vector.addElement(new Double(3.14159));
        vector.addElement(new Float(3.14159));
        System.out.println("Capacity: " + vector.capacity());
        System.out.println("Size: " + vector.size());
        System.out.println(vector);
        System.out.println("First item: " + (Integer)vector.firstElement());
        System.out.println("Last item: " + (Float) vector.lastElement());
        if(vector.contains(new Integer(3)))
        System.out.println("Found 3");
    }
}
```

Output:

```
Size: 0
Capacity: 5
Capacity: 20
Size: 11
[0, 1, 2, 3, 4, 5, 6, 5, 6, 3.14159, 3.14159]
First Item: 0
Last Item: 3.14159
Found 3
```

3.6 SET - HASHSET, TREESET, AND LINKEDHASHSET [W-18]

- The Set Interface defines a set. It extends Collection. Behavior of a collection is that it does not allow duplicate elements to store in the collection. It does not define any additional method of its own. It represents unordered collection.

3.6.1 HashSet

- HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a HashTable for storage. HashTable stores elements by using Hashing mechanism.
- **Hashing:** The information about the key value of each and every element is used to define a hash code. This hash code is used as an index where the data associated with the key value is stored. Conversion of key value into hashcode is performed automatically. You can not see the hash code by itself.
- Advantage of hashing is, the execution time of manipulation of elements remains constant even for large set of elements.

HashSet Constructors:

1. `HashSet()`: Constructs a default hash set.
2. `HashSet(Collection<? extends E> c)`: Hash set initialization using elements of c.
3. `HashSet(int capacity)`: Initializes the capacity of hash set (Default is 16).
4. `HashSet(int capacity, float fillRatio)`: Initializes capacity and fill ratio. The fill ratio must be between 0.0 and 1.0. It determines how full the hash set can be before it is resized upward. Constructors that do not accept fillRatio, default 0.75 fillRatio is used. Hash set does not provide extra methods other than defined by its super classes and interfaces. Hash set does not guarantee the order of elements.

Program 3.5: The following program demonstrates use of HashSet class.

```
File: HashsetDemo.java
import java.util.*;
public class HashsetDemo
{
   public static void main(String args[])
   {
      HashSet hashset1 = new HashSet();
      hashset1.add("Item 0");
      hashset1.add("Item 1");
      hashset1.add("Item 2");
      hashset1.add("Item 3");
      hashset1.add("Item 4");
      hashset1.add("Item 5");
      hashset1.add("Item 5");
      hashset1.add("Item 6");
      hashset1.add("Item 6");
      hashset1.add("Item 6");
      System.out.println(hashset1);
   }
}
```

Output:

```
[Item 0, Item 4, Item 3, Item 2, Item 1, Item 6, Item 5]
```

3.6.2 TreeSet

- TreeSet extends AbstractSet and implements NavigableSet interface. It creates a colletion that uses a tree to store the elements. All the objects are stored in sorted, ascending order. Access time and retrieval time is really fast.
- TreeSet is an excellent choice when we want to store information in a sorted manner and need to be found quickly.

TreeSet Constructors:

1. `TreeSet()`: Empty tree set.
2. `TreeSet(Collection<? extends E> c)`: Tree set which contains the elements of c.
3. `TreeSet(Comparator<? Super E> comp)`: Empty tree set which will be sorted according to the specified comparator comp.
4. `TreeSet(SortedSet<E> s)`: Tree set that contains the elements of s.

- TreeSet implements NavigableSet interface, so you can use all the methods defined by NavigableSet interface.

Program 3.6: Program that demonstrates TreeSet class.

```
File: TreesetDemo.java
import java.util.*;
public class TreeSetDemo
{
   public static void main(String args[])
   {
      TreeSet treeset1 = new TreeSet();
      treeset1.add("Item 3");
      treeset1.add("Item 0");
      treeset1.add("Item 1");
      treeset1.add("Item 2");
      treeset1.add("Item 4");
      treeset1.add("Item 6");
      treeset1.add("Item 5");
      System.out.println(treeset1);
   }
}
```

Output:

```
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5, Item 6]
```

3.6.3 LinkedHashSet

- It maintains a doubly-linked list of all elements. It is ordered version of HashSet. In HashSet while iterating the order is unpredictable but in LinkedHashSet it iterates through in which order they are inserted.

Constructors of LinkedHashSet:

1. `HashSet()`: It constructs a default HashSet.
2. `HashSet(Collection c)`: It initializes the hash set by using the elements of the collection c.

3. **`LinkedHashSet(int capacity)`:** It initializes the capacity of the linked hash set to the given integer value capacity.
4. **`LinkedHashSet(int capacity, float fillRatio)`:** It initializes both the capacity and the fillratio (also called load capacity) of the hash set from its argument.

Program 3.7: Program that demonstrates LinkedHashSet.

```
import java.util.*;
public class LinkHS
{
   public static void main(String args[])
   {
      LinkedHashSet<String> ls=new LinkedHashSet<String>();
      ls.add("Cafe Latte ..... ");
      ls.add("Expresso ..... ");
      ls.add("Cafe Borgia .... ");
      ls.add("Cappuccino ....");
      Iterator<String> i=ls.iterator();
      while(i.hasNext())
      {
         System.out.println(i.next());
      }
   }
}
```

Output:

```
Cafe Latte .....
Expresso .....
Cafe Borgia ....
Cappuccino ....
```

3.7 MAP – HASHMAP, LINKEDHASHMAP, HASHTABLE AND TREEMAP [S-19]

3.7.1 Map

- It has values based on key and value pair. It contains unique keys. If you want to update, search and delete elements you can use Map.
- Map uses interface for implementation in Java.
- The HashMap and LinkedHashMap classes implement the Map interface, whereas TreeMap implements the Map, NavigableMap, and SortedMap interface.
- In Map you have duplicate values but can't have duplicate keys.

- HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.
- A Map cannot be traversed, so you need to convert it into Set using `keySet()` or `entrySet()` method.

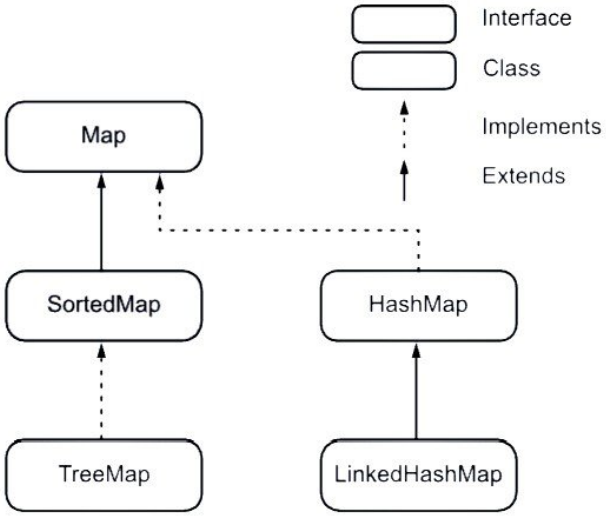


Fig. 3.1: Map Hierarchy

Table 3.9: Methods of Map interface

Method	Description
`put(Object key, Object value)`	It puts key and value in the map.
`remove(Object key)`	It removes an entry for the specified key.
`remove(Object key, Object value)`	It removes the specified values associated with specified keys from the map.
`keySet()`	It returns the Set view containing all the keys.
`entrySet()`	It returns the Set view containing all the keys and values.
`clear()`	It resets the map.
`containsValue(Object value)`	It returns true if some value equal to the value exists within the map, else return false.
`containsKey(Object key)`	It returns true if some key equal to the key exists within the map, else return false.
`equals(Object o)`	It compares the specified Object with the Map.
`get(Object key)`	This method returns the object that contains the value associated with the key.

contd. ...

hashCode()	It returns the hash code value for the Map
isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
replace(K key, V value)	Replace the value of the key K with the new specified value V.
Collection values()	It returns a collection view of the values contained in the map.
int size()	It returns the number of entries in the map.

Program 3.8: Program that demonstrates Map.

```
import java.util.*;
public class Mapeg
{
    public static void main(String args[])
    {
        Map<Integer,String> mp=new HashMap<Integer,String>();
        mp.put(11,"Mangoes");
        mp.put(13,"Apple");
        mp.put(12,"Pomegranate");
        //Elements can traverse in any order
        for(Map.Entry me:mp.entrySet())
        {
            System.out.println(me.getKey()+" "+me.getValue());
        }
    }
}
```

Output:

```
11 Mangoes
12 Pomegranate
13 Apple
```

3.7.2 LinkedHashMap

- Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface. It has values based on the key. It has unique elements where insertion order is maintained.
- LinkedHashMap class declaration.

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
```

Constructors of LinkedHashMap class:

1. `LinkedHashMap()`: It constructs a default LinkedHashMap.
2. `LinkedHashMap(int capacity)`: It initializes a LinkedHashMap with the given capacity.
3. `LinkedHashMap(int capacity, float loadFactor)`: It initializes both the capacity and the load factor.
4. `LinkedHashMap(int capacity, float loadFactor, boolean accessOrder)`: It initializes both the capacity and the load factor with specified ordering mode.
5. `LinkedHashMap(Map<? extends K,? extends V> m)`: It initializes the LinkedHashMap with the elements from the given Map class m.

Methods of LinkedHashMap class:

1. `get(Object key)`: It returns the value to which the specified key is mapped.
2. `clear()`: It is used removes all the key-value pairs from a map.
3. `entrySet()`: It returns a Set view of the mappings contained in the map.
4. `values()`: It returns a Collection view of the values contained in this map.
5. `keySet()`: It returns a Set view of the keys contained in the map.
6. `containsValue(Object value)`: It returns true if the map maps one or more keys to the specified value.
7. `getOrDefault(Object key, V defaultValue)`: It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key.
8. `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.
9. `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.
10. `removeEldestEntry(Map.Entry<K,V> eldest)`: It returns true on removing its eldest entry.

Program 3.9: Program that demonstrates LinkedHashMap.

```
import java.util.*;
public class LHMap1
{
    LHMap1()
    {
        System.out.println("\n Linked HashMap ");
    }
```

```
    public static void main(String args[])
    {
        LinkedHashMap<Integer,String>
        l=new LinkedHashMap<Integer,String>();
        l.put(1001,"Om");
        l.put(2001,"Alisha");
        l.put(3001,"Advik");
        for(Map.Entry k:l.entrySet())
        {
            System.out.println(k.getKey()+" "+k.getValue());
        }
    }
}
```

Output:

```
1001 Om
2001 Alisha
3001 Advik
```

3.7.3 TreeMap

- It is non-synchronized which provides an efficient means of storing key-value pairs in sorted ascending order. It contains only unique elements. It can't have a null key but can have multiple null values.

TreeMap class declaration:

```
public class TreeMap<K,V> extends AbstractMap<K,V> implements
NavigableMap<K,V>, Cloneable, Serializable
```

TreeMap class Parameters:

Parameters for java.util.TreeMap class.

- **K:** It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

Constructors of TreeMap class:

1. `TreeMap()`: It constructs an empty tree map that will be sorted using the natural order of its key.
2. `TreeMap(Comparator<? super K> comparator)`: It constructs an empty tree-based map that will be sorted using the comparator comp.
3. `TreeMap(Map<? extends K,? extends V> m)`: It initializes a treemap with the entries from m, which will be sorted using the natural order of the keys.
4. `TreeMap(SortedMap<K,? extends V> m)`: It initializes a treemap with the entries from the SortedMap sm, which will be sorted in the same order as sm.

Table 3.10: Methods of TreeMap class

Method	Description
`Map.Entry<K,V> ceilingEntry(K key)`	It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.
`void clear()`	It removes all the key-value pairs from a map.
`Object clone()`	It returns a shallow copy of TreeMap instance.
`Map.Entry firstEntry()`	It returns the key-value pair having the least key.
`SortedMap<K,V> headMap(K toKey)`	It returns the key-value pairs whose keys are strictly less than toKey.
`Set keySet()`	It returns the collection of keys exist in the map.
`Map.Entry<K,V> lastEntry()`	It returns the key-value pair having the greatest key, or null if there is no such key.
`put(K key, V value)`	It inserts the specified value with the specified key in the map.
`replace(K key, V value)`	It replaces the specified value for a specified key.
`boolean containsKey(Object key)`	It returns true if the map contains a mapping for the specified key.
`boolean containsValue(Object value)`	It returns true if the map maps one or more keys to the specified value.
`firstKey()`	It is used to return the first (lowest) key currently in this sorted map.
`get(Object key)`	It is used to return the value to which the map maps the specified key.
`lastKey()`	It is used to return the last (highest) key currently in the sorted map.
`remove(Object key)`	It removes the key-value pair of the specified key from the map.
`Set<Map.Entry<K,V>> entrySet()`	It returns a set view of the mappings contained in the map.
`int size()`	It returns the number of key-value pairs exists in the hashtable.
`Collection values()`	It returns a collection view of the values contained in the map.

Program 3.10: Program using TreeMap.

```
import java.util.*;
public class TreeMap1
{
   public static void main(String args[])
   {
      TreeMap<Integer,String> mp=new TreeMap<Integer,String>();
      mp.put(54001,"Raj");
      mp.put(53002,"Shamila");
      mp.put(54004,"Pari");
      mp.put(54003,"Aradhana");
      for(Map.Entry me:mp.entrySet())
      {
         System.out.println(me.getKey()+" "+me.getValue());
      }
   }
}
```

Output:

```
53002 Shamila
54001 Raj
54003 Aradhana
54004 Pari
```

3.7.4 HashTable

- HashTable is a concrete implementation of Dictionary class but it was reengineered to also implement Map interface. So it is now similar to HashMap but it is synchronized.
- HashTable stores key/value pairs in a hash table. However neither keys nor values can be null. The key is hashed and the resulting hash code is used as index where the value is stored within the table.

Constructors of Hashtable:

1. `HashTable()`: Default constructor.
2. `HashTable(int size)`: Hash table with initial size.
3. `HashTable(int size, float fillRatio)`: Hash table with initial size and fill ratio.
4. `HashTable(Map<?extends K, ?extends V> m)`: Hash table with the elements in m.

Methods of Hashtable:

1. `clear()`: It resets and empties the hash table.
2. `contains(Object value)`: It returns true if some value which is equal to value present in the hash table, false otherwise.
3. `containsKey(Object key)`: It returns true if some key which is equal to key present in the hash table, false otherwise.
4. `containsValue(Object value)`: It returns true if some value which is equal to value present in the hash table, false otherwise.
5. `elements()`:It returns an enumeration of the values contained in the hash table.
6. `get(Object key)`: It returns the object that contains the value associated with the key.
7. `isEmpty()`: It returns true if the hash table is empty, false otherwise.
8. `keys()`: It returns an enumeration of the keys present in the hash table.
9. `put(K key, V value)`: It inserts a key and a value in the hash table.
10. `int size()`: It returns the number of entries in the hash table.
11. `remove(Object key)`: It removes key and its value from the hash table. It returns the value associated with key.

Program 3.11: The following program illustrates Hashtable class. The program uses hash table to store the contents. Hash table does not directly supports iterators. Hence program uses enumeration to display the contents of the hash table. However, you can also obtain set-view of the hash table that permits the use of iterators. This can be done by calling entrySet() or keyset() method of Map interface like the one we have demonstrated in HashMap example. **[W-18]**

```
File: HashTableDemo.java
import java.util.*;
public class HashTableDemo
{
   public static void main(String args[])
   {
      Hashtable hashtable1 = new Hashtable();
      hashtable1.put("Item 0", "Value 0");
      hashtable1.put("Item 1", "Value 1");
      hashtable1.put("Item 2", "Value 2");
      hashtable1.put("Item 3", "Value 3");
      hashtable1.put("Item 4", "Value 4");
      hashtable1.put("Item 5", "Value 5");
      hashtable1.put("Item 6", "Value 6");
      Enumeration keys = hashtable1.keys();
```

```
        while(keys.hasMoreElements())
        {
            String key = (String) keys.nextElement();
            System.out.println(key + " / " + hashtable1.get(key));
        }
    }
}
```

Output:

```
Item 0 / Value 0
Item 6 / Value 6
Item 5 / Value 5
Item 4 / Value 4
Item 3 / Value 3
Item 2 / Value 2
Item 1 / Value 1
```

3.8 INTERFACES SUCH AS ITERATOR, LISTITERATOR, ENUMERATION

Iterator Interface:

- If you want to cycle through the elements of collection, then the collection must have an iterator.
- Iterator enables you to cycle through the collection. Following table shows the methods from Iterator interface.

Methods defined by Iterator:

1. **`boolean hasNext()`:** It returns true if there are more elements. Otherwise, returns false.
2. **`E next()`:** It returns the next element. Throws NoSuchElementException if there is not a next element.
3. **`void remove()`:** It removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

Program 3.12: Using Iterator Interface.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
public class JavaIteratorExample1
{
    public static void main(String[] args)
    {
        List<String> list = new LinkedList<>();
```

```
        list.add("Ohio");
        list.add("Florida");
        list.add("California");
        list.add("Alaska");
        System.out.println("The list is given as: "+list);
        Iterator<String> i = list.iterator();
        while(i.hasNext())
        {
           // next element returned
           System.out.println(i.next());
        }
        // last element removed
        i.remove();
        System.out.println("List after removing Last element: "+list);
     }
  }
```

Output:

```
The list is given as: [Ohio, Florida, California, Alaska]
Ohio
Florida
California
Alaska
List after removing Last element: [Ohio, Florida, California]
```

ListIterator Interface:

- ListIterator extends Iterator. With the help of ListIterator, we can traverse the elements of the list bi-directionally.
- Following table shows the methods from ListIterator interface.

Methods defined by ListIterator:

1. **`void add(E obj)`:** Adds obj into the list in front of the element that will be returned by the next call to next().
2. **`boolean hasNext()`:** Returns true if there is a next element. Otherwise, returns false.
3. **`boolean hasPrevious()`:** Returns true if there is a previous element. Otherwise, returns false.
4. **`E next()`:** Returns the next element. A NoSuchElementException is thrown if a next element is not present.
5. **`int nextIndex()`:** Returns the index of the next element. If there is no next element, it returns the size of the list.

6. `E previous()`: Returns the previous element. A NoSuchElementException is thrown if there is no such previous element.
7. `int previousIndex()`: Returns the index of the previous element. If there is no such previous element, returns –1.
8. `void remove()`: Removes the current element from the list. An IllegalStateException is thrown if remove( ) method is called before next( ) or previous( ) is invoked.
9. `void set(E obj)`: Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Steps to use an Iterator to cycle through the elements:

- Obtain an iterator by calling the collections iterator() method.
- Write a loop that will give a call to hasNext() method. This method returns true, as long as there are elements in the collection.
- Obtain each element of collection by calling next() method.
- For collections which implement List interface, you can also obtain an iterator by calling listIterator() method. With list iterator, you can access the elements of a collection in both directions forward as well as backward.

For-each alternative to Iterators:

- If you do not want to modify the contents of collection or you do not want to obtain the elements in reverse order, then For-Each loop is convenient to use to cycle through a collection than iterator. The collection that implements Iterable interface, can operate upon by 'for' loop.
- Use of 'for' loop is shorter and simpler than Iterator approach. But you cannot modify the contents and you can cycle through the collection only in forward direction while using for loop.

Program 3.13: Using List Iterator.

```
// Java program to demonstrate working of ListIterator
import java. util.* ;
public class Test
{
   public static void main(String args[])
   {
      Vector<Integer> k = new Vector<Integer>();
      k.add(111);
      k.add(222);
      k.add(333);
      ListIterator l1 = k.listIterator();
      System.out.println("Iterates in Forward direction …..");
```

```
        while (l1.hasNext())
            System.out.print(l1.next()+" ") ;
            System.out.print("\n\nIterates in backward direction ….") ;
        while (l1.hasPrevious())
            System.out.print(l1.previous()+" ");
    }
}
```

Output:

```
Iterates in Forward direction …..
111 222 333
Iterates in backward direction ….333 222 111
```

Enumeration Interface:

- Enumeration interface defines methods by which you can obtain elements one at a time from the collection of objects. Though this interface is not deprecated it has been superseded by Iterator. It is still used by several legacy classes such as Vector and Properties.
- Enumeration has two methods:
  1. **`boolean hasMoreElements()`:** Returns true if there are still more elements.
  2. **`E nextElement()`:** Returns next object in enumeration.

Program 3.14: Enumeration Interface.

```
public class Enum
{
    public static void main(String args[])
    {
        java.util.Vector v1;
        v1 = new java.util.Vector();
        System.out.println("\n New Elements In Vector ........ \n \n");
        v1.add(new Integer(1000));
        v1.add(new Integer(2000));
        v1.add(new Integer(3000));
        v1.addElement(new Boolean(true));
        java.util.Enumeration e1 = v1.elements();
        while(e1.hasMoreElements())
        {
            System.out.println(e1.nextElement());
        }
    }
}
```

- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Java exception handling is handled by five different keywords, these are try, catch, throw, throws and finally keywords.
  1. **try:** It is keyword which is utilized as a program statement. The statements which you want to monitor for exceptions are written inside try block. If any exception occurs it's thrown.
  2. **catch:** It is a keyword used to catch an exception object thrown by try block.
  3. **throw:** To throw exception manually, we use throw keyword.
  4. **throws:** In some cases, we need to use a method to throw an exception. When we want to throw an exception thrown out of a method is written here.
  5. **finally:** Whether an exception occurs or not, handled or not the finally block is executed. In other words, if an exception is not handled by any of the catch block, it will be handled by finally.
- Following is the general **form/syntax** of an exception handling block:

```
try
{
   // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
   // exception handler for ExceptionType1
}
```

```
catch (ExceptionType2 exOb)
{
   // exception handler for ExceptionType2
}
// ...
finally
{
   // block of code to be executed after try block ends
}
```

- Here, ExceptionType is the type of exception that has occurred.

3.9.1 Exception Class

- The base class for all Exception objects is java.lang.Throwable, together with its two subclasses java.lang.Exception and java.lang.Error.
- The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.
- The Exception class describes the error caused by program (e.g. FileNotFoundException, IOException). These errors could be caught and handled by program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).
- There are three types of Exceptions i.e., Checked Exceptions, Unchecked Exceptions and Error.

3.9.2 Checked Exception

- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions.
- These are the exceptions which occur during the compile time of the program. The compiler checks at the compile time that whether the program contains handlers for Checked Exceptions or not.
- These exceptions must be handled to avoid a compile-time error by the programmer. These exceptions extend the java.lang.Exception class.
- These exceptional conditions should be predicted and recovered by an application. Furthermore Checked exceptions are required to be caught.
- For example, if you call the readLine() method on a BufferedReader object then the IOException may occur. If you want to build a program that reads data using the method readLine() then the method should have code to handle the IOException.
- Here, is the list of Checked Exceptions:
  1. NoSuchFieldException
  2. InstantiationException

3. IllegalAccessException
4. ClassNotFoundException
5. NoSuchMethodException
6. CloneNotSupportedException
7. InterruptedException.

3.9.3 Unchecked Exception

- The classes that extend RuntimeException are known as Unchecked Exceptions. Unchecked exceptions are the exceptions which occur during the runtime of the program.
- Unchecked exceptions are internal to the application and extend the java.lang.RuntimeException that is inherited from java.lang.Exception class.
- These exceptions cannot be predicted and recovered like programming bugs, such as logic errors or improper use of an API.
- These types of exceptions are also called Runtime exceptions that are usually caused by data errors, like arithmetic overflow, divide by zero etc.
- Here, is the list of Unchecked Exceptions.
  1. IndexOutOfBoundsException
  2. ArrayIndexOutOfBoundsException
  3. ClassCastException
  4. ArithmeticException
  5. NullPointerException
  6. IllegalStateException
  7. SecurityException.

3.9.4 Catching Exception and Exception Handling – try, catch, finally, throwand throws, multiple catch block [S-19]

- Keywords used in Java for Exception handling are as follows: Try, Catch, Finally, Throw, Throws

1. Try: [W-18]

- This block of code might raise an exception in special block having try keyword. For example, we might suspect that there might be a "division by zero" operation in the code that will throw an exception.
- **Syntax** of the try block is as follows:

```
try
{
   //set of statements that can raise exception
}
```

2. Catch: **[W-18]**

- It is used to catch an exception when it is raised. It is always used with try and used when try block raises an exception.
- **Syntax** of the catch block is:

```
catch (Exception e)
{
   //code to handle exception e
}
```

- **Syntax** of the try-catch block:

```
try
{
   //code causing exception
}
catch (exception (exception_type) e (object))
{
   //exception handling code
}
```

- The try block can have multiple lines of code that can raise multiple exceptions. Each of these exceptions is handled by an independent catch block.
- **Java Try Catch Example:** In the try block, we define a division operation. The divisor is zero. The statement that divides the two numbers raises an Arithmetic exception. The catch block defines a handler for Arithmetic exception.

Program 3.15: Using Try-Catch.

```
public class Main
{
   public static void main(String args[])
   {
      int i, j;
      try
      {
         //Stmt to Raise Exceptions
         System.out.println("Inside Try Block … ");
         i = 0;
         j = 2078/ i;
         System.out.println("\n Value of j: "+j);
         System.out.println("Try Block Over …….");
      }
```

```
        catch (ArithmeticException e)
        {
            // ArithmeticException Handler
            System.out.println("Divide by Zero Arithmetic Expression …….");
        }
        System.out.println("Outside Try-Catch Block ……");
    }
}
```

Output:

```
Inside Try Block …
Divide by Zero Arithmetic Expression …….
Outside Try-Catch ……
```

3. Nested Try-Catch/Multiple Catch:

- A try block inside another try block is called a nested try block.
- **Syntax** of a nested try block is given below:

```
try
{
    //try_block 1;
    try
    {
        //try_block 2;
    }
    catch(Exception e)
    {
        //exception handler code
    }
}
catch(Exception e)
{
    //exception handler code
}
```

4. Finally: **[W-18]**

- It follows try block or try-catch block. Some code in our program that needs to be executed irrespective of whether or not the exception is thrown. The finally block cannot exist without a try block.

Program 3.16: Program using nested try-multiple catch block with finally.

```
public class Main
{
    public static void main(String args[])
    {
        //Main try block
        try
        {
            //try block1
            try
            {
                System.out.println("Inside Try Block1 ……..");
                int num =100/0;
                System.out.println(num);
            }
            catch(ArithmeticException e1)
            {
                System.out.println("Inside Catch Block1 Exception: e1");
            }
            //try block2
            try
            {
                System.out.println("Inside Try Block2 ……..");
                int num =200/0;
                System.out.println(num);
            }
            catch(ArrayIndexOutOfBoundsException e2)
            {
                System.out.println("Inside Block2 Exception: e2");
            }
            System.out.println("Block1 and Block2");
        }
        catch(ArithmeticException e3)
        {
            System.out.println("Arithmetic Exception …….");
        }
        catch(ArrayIndexOutOfBoundsException e4)
        {
            System.out.println("ArrayIndexOutOfBoundsException …….");
        }
```

```
        catch(Exception e5)
        {
           System.out.println("Main Block Exception");
        }
        finally
        {
           System.out.println ("Inside Finally Block ….");
           System.out.println ("Finally block executed … No Exception….");
        }
     }
  }
```

Output:

```
Inside Try Block1 ……..
Inside Catch Block1 Exception: e1
Inside Try Block2 ……..
Arithmetic Exception …….
Inside Finally Block ….
Finally block executed … No Exception….
```

5. Throw:

- Java provides a keyword "throw" using which we can explicitly throw the exceptions in the code or to throw custom exceptions. It can be used to throw the checked or unchecked exceptions.
- **Syntax** of the throw keyword is:

```
throw exception;
or
throw new exception_class("error message");
```

Program 3.17: Using throw keyword.

```
import java.lang.*;
public class Main
{
   static void check_sal(int sal)
   {
      //if specified Salary <200000, throw ArithmeticException
      if(sal < 200000)
      throw new ArithmeticException("\n Salary < 2000 not allowed ….");
      else //print the message
      System.out.println("\n Salary > 20000 .. Njoy ….");
   }
```

```
    public static void main(String args[])
    {
        //call Check_sal
        check_sal(500000);
        System.out.println("\n Be Happy ...");
    }
}
```

Output:

```
Salary > 20000 .. Njoy ….
Be Happy ...
```

6. Throws:

- It is used to declare exceptions but does not throw an exception. It is used to indicate that an exception might occur in the program or method.
- The declaration of exception using the "throws" keyword tells the programmer that there may be an exception specified after the "throws" keyword and the programmer should provide corresponding handler code for this exception to maintain the normal flow of the program.
- Declaring an exception with throws keyword in the method signature and then handling the method call using try-catch seems to be a viable solution.
- Another advantage of declaring exceptions using throws keyword is that we are forced to handle the exceptions. If we do not provide a handler for a declared exception then the program will raise an error.
- **Syntax** of the throws keyword:

```
return_type method_name() throws exception_class_name
{
    //method code
}
```

Program 3.18: Program using Throws.

```
import java.io.IOException;
public class Throws_Eg
{
    void music()throws IOException
    {
        throw new IOException("IO Error");
    }
    void play()throws IOException
    {
        music();
    }
```

```
    void enjoy()
    {
        try
        {
            play();
        }catch(Exception   e){System.out.println("Inside   Catch:   Exception
        Handeled ....");}
    }
    public static void main(String args[])
    {
        Throws_Eg t1=new Throws_Eg();
        t1.enjoy();
        System.out.println("Program End ...");
    }
}
```

Output:

```
Inside Catch: Exception Handeled ....
Program End ...
```

3.9.5 Creating User Defined Exception

- The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them create.
- Exception defines four constructors. Two were added by JDK 1.4 to support chained exceptions, described in the next section.
- The other two are shown here:
  1. Exception( )
  2. Exception(String msg)
- The first form creates an exception that has no description. The second form lets you specify a description of the exception.
- Although specifying a description when an exception is created is often useful, sometimes it is better to override toString( ). Here's why: The version of toString( ) defined by Throwable (and inherited by Exception) first displays the name of the exception followed by a colon, which is then followed by your description.
- By overriding toString( ), you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.
- The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing a carefully tailored description of the exception to be displayed.

Program 3.19: This program creates a custom exception type.

```
class MyException extends Exception
{
   private int detail;
   MyException(int a)
   {
      detail = a;
   }
   public String toString()
   {
      return "MyException[" + detail + "]";\
   }
}
public class ExceptionDemo
{
   static void compute(int a) throws MyException
   {
      System.out.println("Called compute(" + a + ")");
      if(a > 10)
      throw new MyException(a);
      System.out.println("Normal exit");
   }
   public static void main(String args[])
   {
      try
      {
         compute(1);
         compute(20);
      }
      catch (MyException e)
      {
         System.out.println("Caught " + e);
      }
   }
}
```

Output:

```
java ExceptionDemo
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

3.10 I/O

- Java I/O (Input and Output) is used to process the input and produce the output.
- The java.io package contains all the classes required for input and output operations.
- File handling in Java uses Java I/O API.
- **Stream:** It is a sequence of data. In Java, a stream is composed of bytes which are attached with console.
- Streams are:
  1. **System.out:** Standard output stream.
  2. **System.in:** Standard input stream.
  3. **System.err:** Standard error stream.
- **Example:** Print output and an error message to the console.
  1. System.out.println("Stay Safe .....");
  2. System.err.println("Error .....");

  Get input from console.
  1. int i=System.in.read(); //ASCII code of 1st character is returned
  2. System.out.println((char)i);

3.10.1 String Class (Basic Methods)

- The strings in Java are treated as objects of type 'String' class. This class is present in the package java.lang.
- This package contains two string classes:
  1. String class.
  2. StringBuffer class.
- The string class is used when we work with the string which cannot change whereas, StringBuffer class is used when we want to manipulate the contents of the string.
- When we create object of String Class they are designed to be immutable. As these objects are immutable, we can alias to a particular string for many time which we want. Even if we alias the string class objects, they won't affect each other because they are read only.
- We can use + operator to overload for string objects. Only two operators i.e. '+' and '+=" are overloaded for string classes. In Java, when we use + operator it allows us to concatenate two or more number of strings together.
- **Example:** `String str = "We" + "are" + "winners";`

  Output: `Wearewinners`
- In the above example, a method append() is called which creates the new String object having "We" and concatenate with the next contents. It generally calls toString() method which allows you to determine the String representation for objects and classes which are created.

Constructors of String:

- Constructors are used to initialize the object. The string class supports several constructors.
- The following table shows the constructors of the string class.

Table 3.11: String class Constructors

Constructor	Description
`String()`	This constructor creates an empty string.
`String(String value)`	This constructor creates a new string that is a copy of the given string.
`String(char[] value)`	This constructor constructs a new string based on the character array.
`String(char[] value, int begin, int count)`	This constructs a new string based on the character array starting from the position begin which is 'count' characters long.
`String(byte[] value)`	This constructor creates a new string by converting the given array of bytes.
`String(byte[] value, int offset, int length)`	This constructor creates a new string by converting the given sub of array of bytes.
`String(StringBuffer buffer)`	This constructor creates a new string based on a StringBuffer value.
`String(char[] value, int begin, int count, String enc) throws UnsupportedEncoding Exception`	This constructor creates a new string based on the given byte array and uses given character encoding that is denoted by enc.
`String(char[] value, String enc) throws UnsupportedEncoding Exception`	This constructor creates a new string based on the given byte array and uses given character encoding that is denoted by enc.

String class Methods: **[W-18]**

- The String class provides various methods that appear to modify strings.
- The java.lang.String class provides many useful methods to perform operations on sequence of char values.

Table 3.12: String class Methods

Methods	Description
`char charAt(int index)`	This method returns char at (value for) the particular index.
`int length()`	This method returns string length.
`static String format(String format, Object... args)`	This method returns formatted string.
`static String format(Locale l, String format, Object... args)`	This method returns formatted string with given locale.
`String substring(int beginIndex)`	This method returns substring for given begin index.
`String substring(int beginIndex, int endIndex)`	This method returns substring for given begin index and end index.
`boolean contains(CharSequence s)`	This method returns true or false after matching the sequence of char value.
`static String join(CharSequence delimiter, CharSequence... elements)`	This method returns a joined string with delimiter.
`static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)`	This method returns a joined string.
`boolean equals(Object another)`	This method checks the equality of string with object.
`boolean isEmpty()`	This method checks if string is empty.
`String concat(String str)`	This method concatenates specified string.
`String replace(char old, char new)`	This method replaces all occurrences of specified char value.
`String replace(CharSequence old, CharSequence new)`	This method replaces all occurrences of specified CharSequence.
`String trim()`	This method returns trimmed string omitted leading and trailing spaces.
`String split(String regex)`	This method returns splitted string matching regex.
`String split(String regex, int limit)`	This method returns splitted string matching regex and limit.

contd. ...

`String intern()`	This method returns interned string.
`int indexOf(int ch)`	This method returns specified char value index.
`int indexOf(int ch, int fromIndex)`	This method returns specified char value index starting with given index.
`int indexOf(String substring)`	This method returns specified substring index.
`int indexOf(String substring, int fromIndex)`	This method returns specified substring index starting with given index.
`String toLowerCase()`	This method returns string in lowercase.
`String toLowerCase(Locale l)`	This method returns string in lowercase using specified locale.
`String toUpperCase()`	This method returns string in uppercase.
`String toUpperCase(Locale l)`	This method returns string in uppercase using specified locale.

3.10.2 StringBuffer Class

- It is a peer class which provides the functionality of strings. The string generally represents fixed.length, immutable character sequence whereas StringBuffer represents growable and writeable character sequences.
- StringBuffer may have some characters and if needed substring can be inserted in the middle or appended at the end. StringBuffer automatically provides a room to grow such additions.
- Java generally manipulate the strings using + as overloaded operator. StringBuffer class in Java is used to created mutable (modifiable) string. The StringBuffer class in Java is same as String class except it is mutable i.e., it can be changed.
- StringBuffer class is a mutable class unlike the String class which is immutable. Both the capacity and character string of a StringBuffer Class. StringBuffer can be changed dynamically.

Advantages of StringBuffer Class:

1. Alternative to String class.
2. Can be used wherever a string is used.
3. More flexible than String.

StringBuffer Constructors:

1. **`StringBuffer()`:** It is the default constructor as it has no parameters
2. **`StringBuffer(int size)`:** It accepts an integer argument which explicitly sets the size of buffer.
3. **`StringBuffer(String str)`:** The string argument sets the initial contents of the StringBuffer object and reverse room for 16 more characters. These 16 characters room is allocated when no specific buffer length is requested. This is because reallocation is a costly process in terms of time.

- StringBuffer creates string objects that can be changed.
- String object is manufactured which contains the substring. In this the original string is unaltered and immutability remains intact.
- String provides a substring() method which returns a new string. It contains a specified portion of the invoking string. This method is substring().
- The **syntax** is given below:

  `String subString(int start_Index, int endIndex)`
- The startIndex specifies the beginning index and endIndex tells us about stopping point.

StringBuffer Class Methods:

1. `public StringBuffer append(String s)`: Updates the value of the object that invoked the method.
2. `public StringBuffer reverse()`: Reverses the value of the StringBuffer object that invoked the method.
3. `public delete(int start, int end)`: Deletes the string starting from start index until end index.
4. `public insert(int offset, int i)`: Inserts an string s at the position mentioned by offset.
5. `replace(int start, int end, String str)`: Replaces the characters in a substring of this StringBuffer with characters in the specified String.

- Here, is the list of other methods (Except set methods) which are very similar to String class:
  1. `int capacity()`: Returns the current capacity of the String buffer.
  2. `char charAt(int index)`: The specified character of the sequence currently represented by the string buffer, as indicated by the index argument, is returned.
  3. `void ensureCapacity(int minimumCapacity)`: Ensures that the capacity of the buffer is at least equal to the specified minimum.
  4. `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`: Characters are copied from this string buffer into the destination character array `dst`.
  5. `int indexOf(String str)`: Returns the index within this string of the first occurrence of the specified substring.
  6. `int indexOf(String str, int fromIndex)`: Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
  7. `int lastIndexOf(String str)`: Returns the index within this string of the rightmost occurrence of the specified substring.
  8. `int lastIndexOf(String str, int fromIndex)`: Returns the index within this string of the last occurrence of the specified substring.
  9. `int length()`: Returns the length (character count) of this string buffer.

10. `void setCharAt(int index, char ch)`: The character at the specified index of this string buffer is set to `ch`.
11. `void setLength(int newLength)`: Sets the length of this String buffer.
12. `CharSequence subSequence(int start, int end)`: Returns a new character sequence that is a subsequence of this sequence.
13. `String substring(int start)`: Returns a new String that contains a subsequence of characters currently contained in this StringBuffer.The substring begins at the specified index and extends to the end of the StringBuffer.
14. `String substring(int start, int end)`: Returns a new String that contains a subsequence of characters currently contained in this StringBuffer.
15. `String toString()`: Converts to a string representing the data in this string buffer.

Program 3.20: Program to show use of substring.

```
//use of substring
class strTry
{
   public static void main(String args[])
   {
      String st1 = "Java is platform independent";
      // to construct a substring we write
      String sub = st1.substring(9, 16);
      // lets print these values
      System.out.println("Original string is: + st1");
      System.out.println("Substring is: + sub");
   } //end main
} //end class
```

Output:

```
Original string is: Java is platform independent
Substring is: platform
```

Difference between String and StringBuffer:

- String objects are constants and immutable whereas StringBuffer objects are not.
- StringBuffer Class supports growable and modifiable string whereas String class supports constant strings.
- Strings once created we cannot modify them. Any such attempt will lead to the creation of new strings. Whereas StingBuffer objects after creation also can be able to delete or append any characteres to it.
- String values are resolved at run time whereas StringBuffer values are resolved at compile time.

3.10.3 File class

- Although most of the classes defined by java.io operate on streams, File class does not operate on streams. It directly deals with files and file system.
- File class does not specify how the information is retrieved from or stored in files. It also describes the properties of file. File object is used to manipulate such type of information associated with a disk file.
- For example, size, permission, date and time of creation and modification of file, directory path etc. Files are the main source for storing persistent information.
- You must have seen a list of files using a dir command. The list includes all the files along with their attributes in the specified directory.
- A directory in Java, simply is treated as a file with one additional property- a list of filenames in that directory that can be seen with the help of list() method.

Constructors of the File class:

1. `File(String directoryPath)`: directoryPath is the path name of the file.
2. `File(String directoryPath, String fileName)`: fileName is the name of the file or subdirectory.
3. `File(File dirObj, String fileName)`: dirObj is the name of the file object that specifies the directory.
4. `File(URI uriObj)`: uriObj is the name of the URI object that describes a file.

- All of these constructors throw NullPointerException if filename is null.

Examples:

```
File f1 = new File("/");                // A directory path as an argument.
File f2 = new File("/", "test.dat");    // A directory path and a filename.
File f3 = new File(f1, "test.dat");     // refers to the same file as f2.
```

- The methods of File class are given in Table 3.13.

Table 3.13: Methods of File Class

Method	Description
`String getName()`	Returns name of the file.
`String getParent()`	Returns name of parent directory.
`long length()`	Returns length of the file.
`boolean exists()`	Returns true if the files exists Otherwise returns false.
`boolean isFile()`	Returns true if the invoking object is a file. If the invoking object is a directory, device driver, special file or named pipe, the method returns false.

contd. ...

boolean isDirectory()	Returns true if the invoking object is a directory. Otherwise returns false.
boolean isAbsolute()	Returns true if the path is absolute. If the path is relative, returns false.
boolean canRead()	Returns true if file can be read.
boolean canWrite()	Returns true if file can be written.
String getPath()	Returns relative path of the file.
boolean rename(File newName)	Renames the file. Returns true if file is renamed successfully. If for some reasons like attempts to overwrite files, or move them to unauthorized locations, the method returns false.
boolean delete()	Returns true if the file is deleted successfully. Otherwise, returns false. This method can also be used for directories. Before making an attempt to delete a directory, you must ensure that it is empty.
boolean isHidden()	Returns true for hidden files and false for others.
boolean setLastModified(long msec)	Sets the time of last modification of the file. The argument specifies the number of milliseconds from Jan 1, 1970.
boolean setReadOnly()	Makes the file read only.
String[ ] list()	Returns the files and sub directories stored in the directory.
boolean createNewFile()	Creates a new file and returns true if the file is created successfully. Returns false, if for some reasons like file already exists, file is not created.
boolean mkdir()	Creates a directory and returns true if the directory is successfully created.

Program 3.21: A program to illustrate the use of methods of File class that displays information about the file.

```
import java.io.*;
public class FileDemo
{
   public static void main(String[] args)
```

```
    {
        try
        {
            File f = new File("DrawApplet.java");
            System.out.println("\nCan read? " + f.canRead());
            System.out.println("\nCan write? " + f.canWrite());
            System.out.println("\nAbsolute path " + f.getAbsolutePath());
            System.out.println("\nLength " + f.length());
            System.out.println("\nParent " + f.getParent());;
            System.out.println("\nIs file? " + f.isFile());
            System.out.println("\nIs directory? " + f.isDirectory());
            String []list = f.list();
            for(int i=0; i<list.length; i++)
            System.out.println(list[i]);
        }
        catch(NullPointerException e) { System.out.println("Error"); }
    }
}
```

Output:

```
Can read? true
Can write? true
Absolute path D:\Java Programs\DrawApplet.java
Length 888
Parent null
Is file? true
Is directory? false
```

Table 3.14: Common stream classes used for I/O operations

Source or Destination	Characters		Bytes	
	Read	Write	Read	Write
Memory	CharArrayReader	CharArrayWriter	ByteArrayInputStream	ByteArrayOutputStream
File	FileReader	FileWriter	FileInputStream	FileOutputStream
Pipe	PipedReader	PipedWriter	PipedInputStream	PipedOutputStream

FileInputStream:

- This class extends InputStream class to read binary data from a file.
- **Constructors of FileInputStream:**
  - FileInputStream(String path) throws FileNotFoundException.
  - FileInputStream(File fileObject) throws FileNotFoundException.

- Here,
  - path refers to the complete path of the file, and
  - fileObject refers to the object that describes the file.

Program 3.22: A program shows how FileInputStream class is used for reading bytes from file and FileOuputStream class is used to write bytes to a file.

File: FileRead.java

```
import java.io.*;
class FileRead
{
   public static void main(String[] args) throws IOException,
   FileNotFoundException
   {
      File f = new File("Ball.txt");
      FileInputStream fis = new FileInputStream(f);
      FileOutputStream fos = new FileOutputStream("NewBall.txt");
      int value;
      while((value=fis.read()) != -1)
      {
         System.out.print((char)value);
         fos.write(value);
      }
      fis.close();
      fos.close();
   }
}
```

Output:

```
(Contents of ball.txt - Hello !
We are learning Java.)
java FileRead
Hello !
We are learning Java.
```

- Program reads an existing file and displays its bytes on the screen. It also writes data to the new file 'NewBall.txt'. We must first create the file named 'Ball.txt' before we run this program. The new file 'NewBall.txt' is created if it does not already exist. If the file exists, the previous contents are lost and data is written from the beginning of the file. In case if the file to be read or new file to be written is not found for any reason, FileNotFoundException is thrown.

FileOutputStream:

- This stream writes bytes to a file. The constructors are as follows:
  - FileOutputStream (String path) throws FileNotFound Exception
  - FileOutputStream (String path, boolean apppend) throws FileNot Found Exception
  - FileOutputStream (File fileOb)
- All these constructors throw FindNotFoundException. Parameter path indicates complete path to the file and fileOb indicates file object. Parameter append is true if the new data should be appended to the existing data and false if the old data should be overwritten.

Program 3.23: Program using FileOutputStream.

```
import java.io.*;
class FileOutputStreamDemo
{
   public static void main(String args[]) throws Exception
   {
      FileOutputStream fileOut = new FileOutputStream("sample.dat");
      int i;
      for(i=0;i<10;i++)
      fileOut.write(i%5);
      fileOut.close();
      FileInputStream fin = new FileInputStream("sample.dat");
      while ( (i = fin.read()) != -1)
      System.out.println(i);
      fin.close();
   }
}
```

Output:

```
0 1 2 3 4 0 1 2 3 4
```

FileReader:

- This class creates a Reader which you can use to read the characters from the file. This class is extended from InputStreamReader class. It has two forms of constructors.
  - FileReader(String fPath);
  - FileReader(File fileObject);
- Both these forms of constructors throw FileNotFound Exception.

Program 3.24: Program using FileReader.

```
import java.io. *;
class FileReaderDemo
{
   public static void main(String[] arg) throws Exception
   {
      char inpChars[] = new char[ 1024] ;
      FileReader inpFile = new FileReader (arg[01]);
      int charRead = inpFile.read(inpChars);
      String s = new String(inpChars, 0, charRead);
      System.out.println(s.toUpperCase());
      inpFile.close();
   }
}
```

Output:

```
java FileReaderDemo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1 at
FileReaderDemo.main(fileread2.java:13)
```

FileWriter:

- This class creates a Writer which can be used to write the contents to the file.
- It has four forms of constructors.
  - FileWriter(String fPath);
  - FileWriter(String fPath, boolean append);
  - FileWriter(File fileObject);
  - FileWriter(File fileObject, boolean append);
- All these forms of constructors throw IOException. If append is true, then the output is appended to the end of the file.
- The following program uses two file stream classes to copy the contents of one file into the other file.

Program 3.25: Program using FileReader and FileWriter

```
import java.io.*;
class CopyCharacter
{
   public static void main (String args[])
   {
      File ifile = new File ("in.dat");
      File ofile = new File ("out.dat");
```

```
        try
        {
            FileReader in = new FileReader(ifile);
            FileWriter out = new FileWriter(ofile,true);
            int ch;
            while ((ch = in.read())!= -1)
            {
                out.write(ch);
            }
            in.close();
            out.close();
        }
        catch(IOException e)
        {
            System.out.println(e);
            System.exit(-1);
        }
        finally
        {
            System.out.println("\n Inside Finally .. ");
        }
    }
}
```

Output:

```
(Contents of in.dat:
We are learning File handling.)
out.dat ->
We are learning File handling.
Inside Finally
```

Program 3.26: To Create a File.

```
import java.io.File;
public class CreateFile
{
    public static void main(String[] args)
    {
        File f1 = new File("newfile.txt");
```

```
        try
        {
            // create a file
            boolean flag = f1.createNewFile();
            if (flag)
            {
                System.out.println("\n New File created .....");
            }
            else
            {
                System.out.println("\n File already exists .........");
            }
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Output:

```
New File created .....
```

3.10.4 DataInputStream and DataOutputStream class

- The DataInputStream class read primitive Java data types from an underlying input stream in a machine-independent way. While the DataOutputStream class write primitive Java data types to an output stream in a portable way.

DataInputStream:

- This is a highlevel stream that permits reading of primitive data types. When you work with byte streams, you have to convert them to characters or strings to understand their meaning. But this stream provides methods to directly work with meaningful types.
- DataInputStream implements DataInput interface that defines methods to read the sequence of bytes and convert them into values of primitive data types.
- The following Table 3.15 shows commonly used methods. All these methods throw IOException.

Table 3.15: Methods in DataInput Interface

Method	Description
`boolean readBoolean()`	Reads and returns a boolean value from the stream.
`boolean readByte()`	Reads and returns a byte value from the stream.
`boolean readChar()`	Reads and returns a char value from the stream.
`boolean readDouble()`	Reads and returns a double value from the stream.
`boolean readFloat()`	Reads and returns a float value from the stream.
`boolean readInt()`	Reads and returns an integer value from the stream.
`boolean readLong()`	Reads and returns a long value from the stream.
`boolean readshort()`	Reads and returns a short value from the stream.
`void readFully(byte buff[])`	Reads and fills the buffer buff with bytes.
`void readFully(byte buff[], int index, int number)`	Reads specified number of bytes and fills the buffer from buff[index].
`int skip Bytes(int n)`	Skips n bytes in the stream.
`void write (int j)`	Writes the byte in the lower 8 bits of j.
`void write (byte buff[ ])`	Writes contents of the buffer to the stream.
`void write (byte buff [ ], int index, int len)`	Writes a portion of the buffer to the stream (starting from index and of length len).
`void close ()`	Writes all data in the stream and closes the stream.
`void flush ()`	Flushes (writes) all data from the stream.

DataOutputStream:

- This stream also extends from FilterOutputStream. This writes primitive data to a byteoriented output stream. It supports all of its methods defined by its superclasses.
- It implements DataOutput interface which define methods that convert values from primitive data type into a byte sequence and then writes to the underlying stream. writeDouble(), writeBoolean(), writeInt() are some of the examples of such methods.

Table 3.16: Java DataOutputStream methods

Method	Description
`int size()`	It returns the number of bytes written to the data output stream.
`void write(int b)`	It writes the specified byte to the underlying output stream.
`void writeBoolean(boolean v)`	It writes Boolean to the output stream as a 1-byte value.

contd. ...

void writeChar(int v)	It writes char to the output stream as a 2-byte value.
void writeByte(int v)	It writes a byte to the output stream as a 1-byte value.
void writeInt(int v)	It writes an int to the output stream
void writeShort(int v)	It is writes a short to the output stream.
void writeLong(long v)	It writes a long to the output stream.
void writeUTF(String str)	It writes a string to the output stream using UTF-8 encoding in portable manner.
void flush()	It flushes the data output stream.

Program 3.27: Using DataOutputString.

```
package com.java1;
import java.io.*;
public class OutputExample
{
    public static void main(String[] args) throws IOException
    {
        FileOutputStream f = new FileOutputStream(C:\\temp.txt);
        DataOutputStream d= new DataOutputStream(f);
        d.writeInt(70);
        d.flush();
        d.close();
        System.out.println("File ...");
    }
}
```

Summary

- Java.util is an important package which contains a large collection of classes and interfaces which supports a broad range of functionality. It contains Java's most powerful subsystem collections framework. Collection Framework is a sophisticated hierarchy of classes and interfaces for managing groups of objects.
- Framework had to be high performance. Implementations for fundamental collections are highly efficient.

- The Collections Framework defines several interfaces. Collections interfaces determine the fundamental nature of Collection classes.
- List interface extends Collection. This interface has a behavior that stores a sequence of elements. Elements can be inserted or accessed by their position in the list.
- The Set Interface defines a set. It extends Collection. Behavior of a collection is that it does not allow duplicate elements to store in the collection. It does not define any additional method of its own. It represents unordered collection.
- HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a HashTable for storage. HashTable stores elements by using hashing mechanism.
- TreeSet extends AbstractSet and implements NavigableSet interface. It creates a collection that uses a tree to store the elements.
- Vector implements dynamic array. It is similar to ArrayList with two major differences. Vector class is re-engineered to extend AbstractList and it implements List interface. It is also re-engineered to implement Iterable interface. Means, Vector can have its elements traversed through the for loop. Vector is fully compatible with collections.
- HashTable is a concrete implementation of Dictionary but it was re-engineered to also implement Map interface. So it is now similar to HashMap but it is synchronized.
- HashTable stores key/value pairs in a hash table. However neither keys nor values can be null. The key is hashed and the resulting hash code is used as index where the value is stored within the table.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Check Your Understanding

1. Which of these interface declares core method that all collections will have?
   (a) Set (b) EventListner
   (c) Comparator (d) Collection
2. Which of these interface handle sequences?
   (a) Set (b) List
   (c) Comparator (d) Collection
3. Which of this interface must contain a unique element?
   (a) Set (b) List
   (c) Array (d) Collection

4. What is the output of this program?

```
import java.util.*;
class Maps
{
   public static void main(String args[])
   {
      TreeMap obj = new TreeMap();
      obj.put("A", new Integer(1));
      obj.put("B", new Integer(2));
      obj.put("C", new Integer(3));
      System.out.println(obj.entrySet());
   }
}
```

(a) [A, B, C] (b) [1, 2, 3]
(c) {A=1, B=2, C=3} (d) [A=1, B=2, C=3]

5. Which of this interface is not a part of Java's collection framework?
(a) List (b) Set
(c) SortedMap (d) SortedList

6. Which of these is Basic interface that all other interface inherits?
(a) Set (b) Array
(c) List (d) Collection

7. ArrayList is _____ in size.
(a) Static (b) Dynamic
(c) Robust (d) fixed

8. Linked List is Collection framework present in _____.
(a) java.util.package. (b) java.lang.*
(c) java.io.* (d) java.collection.*

9. The base class for all Exception objects is _____.
(a) java.io*; (b) java.*
(c) java.lang.Throwable (d) java.exception.*

10. _____ class write primitive Java data types to an output stream in a portable way.
(a) DataInputStream (b) FileInputsream
(c) Final (d) DataOutputStream

11. Which of these packages contain all the Java's built in exceptions?
(a) java.io (b) java.util
(c) java.lang (d) java.net

12. Which of these class is used to create user defined exception?
    (a) java.lang (b) Exception
    (c) RunTime (d) System
13. Which of these keywords must be used to monitor for exceptions?
    (a) try (b) finally
    (c) throw (d) catch

Answers

1. (d)	2. (b)	3. (a)	4. (d)	5. (d)	6. (d)	7. (b)	8. (a)	9. (c)	10. (d)
11. (c)	12. (b)	13. (a)							

Practice Questions

Q.I Answer the following questions in short:

1. What is meant by collection framework?
2. What are the interfaces used in collection framework?
3. Write short note on Collection interface,
4. Write short note on List interface.
5. Enlist classes to collection framework.
6. With the help of example explain iterator interfaces.
7. Explain ListIterator interface with example.
8. Write short note on: Enumeration interface.
9. What is list interface?
10. Describe vector with example.
11. Explain LinkedList with example.
12. What is an exception? In which package exceptions are defined?
13. Can user program throw own exceptions? How?
14. Explain the purpose of throws clause.
15. What is the purpose of finally block in the program?

Q.II Answer the following Questions:

1. Write a short note on wrapper class
2. Explain checked and unchecked exceptions
3. Write a short note on exception handling
4. Differntiate between array and arraylist
5. Explain String and StringBuffer
6. Explain different blocks try , catch , finally with suitable examples
7. How is file handling done in Java
8. How can you create user defined exception give suitable example
9. What are advantages of String Buffer

10. Explain HashMap with suitable example.
11. Write short note on List.
12. Write short note on TreeSet.

Q.III Define the terms:

1. Wrapper Class
2. Collection Interface
3. ListIterator
4. Arraylist
5. Vector
6. Exception
7. StringBuffer
8. Collection
9. TreeSet
10. HashMap

Previous Exam Questions

Winter 2018

1. Which of these classes is used to create user defined exception? **[1 M]**

(i) Object (ii) Exception
(iii) Java.util (iv) Runtime

Ans. (ii)

2. Which of these methods of class string is used to find ifirst occurrence of character or substring? **[1 M]**

(i) charAt() (ii) indexOf()
(iii) substring() (iv) split()

Ans. (ii)

3. Which is not a directive? **[1 M]**

(i) page (ii) include
(iii) expressions (iv) taglib

Ans. (iii)

4. Name the classes which implement the set interface. **[1 M]**

Ans. Refer to Section 3.6.

5. Write a java program to create Hashtable containing city names and pincode. Display the details of Hashtable using Enumeration. **[4 M]**

Ans. Refer to Program 3.11.

6. Explain the use of try, catch and finally block. **[3 M]**

Ans. Refer to Section 3.9.4.

7. Explain any four methods of string class with proper syntax. **[4 M]**

Ans. Refer to Section 3.10.1.

8. Write a java program to accept 'n' strings from user. Store it into vector. Display the first and last element of a vector. **[3 M]**

Ans. Refer to Program 3.4.

Summer 2019

1. Which method is used to remove the blank space from string? **[1 M]**

(i) to string( ) (ii) equal ignore case( )

(iii) trim( ) (iv) compare to( )

Ans. (iii)

2. "Vector is a growable or changeable size"—justify true or false. **[1 M]**

Ans. Refer to Section 3.4.

3. State any two differences between throw and throws. **[1 M]**

Ans. Refer to Section 3.9.4.

4. Name the package which import the tree set class. **[1 M]**

Ans. Refer to Section 3.1.

5. What is exception in Java? **[1 M]**

Ans. Refer to Section 3.9.

6. Write a java program to accept n numbers in vector and display all statements sum. **[4 M]**

Ans. Refer to Program 3.4.

7. Write a java program to accept n number of city. Insert into array list collection and display the content of the same array list, remove all this element use (clear( )) method. **[4 M]**

Ans. Refer to Program 3.2.

8. What is exception? Explain try, catch and finally block. **[3 M]**

Ans. Refer to Section 3.9.4.

9. Write a java program to accept n string from user and store it into vector. Display the first and last element of vector. **[3 M]**

Ans. Refer to Program 3.4.

10. What is interface? Explain the use map interface with suitable example. **[4 M]**

Ans. Refer to Section 3.7.

■■■

4... Swing

Learning Objectives...

- To learn Basic Concepts of Swing.
- To study the MVC architecture and Swing.
- To study Layout Manager and JComponent class.
- To study Event Handling in Java.
- To learn Anonymous inner class.

4.1 WHAT IS SWING?

- The swing components are defined in the package javax.swing. This package provides more powerful and flexible components.
- Swing is an extension to the AWT components which provides feature of pluggable "look and feel" for the components. It provides classes to design lightweight components.
- Swing is the next-generation GUI toolkit which Sun Microsystems is developing to enable enterprise development in Java.
- Swing is actually part of a large family of Java products which is known as Java Foundation Classes (JFC).
- Swing is used to create large-scale Java applications with a wide array of powerful components which can you easily extend or modify these components to control their appearance and behavior based on the current "look and feel" library that is being used.
- In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.
- Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, they are platform-independent. The term light weight is used to describe such elements.

4.1.1 Swing Features

- Various features of swing are listed below:
  1. **Light Weight**: Swing component are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
  2. **Rich Controls**: Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, color picker, table controls and so on.
  3. **Borders:** We can draw borders in many different styles around components using the `setborder()` method.
  4. **Easy Mouseless Operation:** It is easy to connect keystrokes to components.
  5. **Tooltips:** We can use the `setToolTipText` method of JComponent to give components a tooltip, one of those small windows that appear when the mouse hovers over a component and gives explanatory text.
  6. **Easy Scrolling:** We can connect scrolling to various components-something that was impossible in AWT.
  7. **Pluggable Look and Feel:** We can set the appearance of AWT and applications to one of three standard looks, i.e., Windows, Motif (Unix) or Metal (Standard swing look).
  8. **Highly Customizable**: Swing controls can be customized in very easy way as visual appearance is independent of internal representation.
  9. **New Layout Managers:** Swing introduces the BoxLayout and Overlay Layout managers.

4.1.2 Working with Swing

- While designing any GUI, we need to have a main window on which we can place or position the different visual components.
- Every top level container has one intermediate container called content pane. This content pane contains all the visible components in the GUI window.
- In swing, the main window also called a top level container is the root of a hierarchy which contains all the other swing components that appear inside the window. All swing applications have at least one top level container.
- The content pane is the base pane upon which all other components or container objects are placed. One exception to this rule is, if there is a menu bar in the top level container it will be placed outside the content pane.
- Fig. 4.1 illustrates the relationship between the components discussed above.

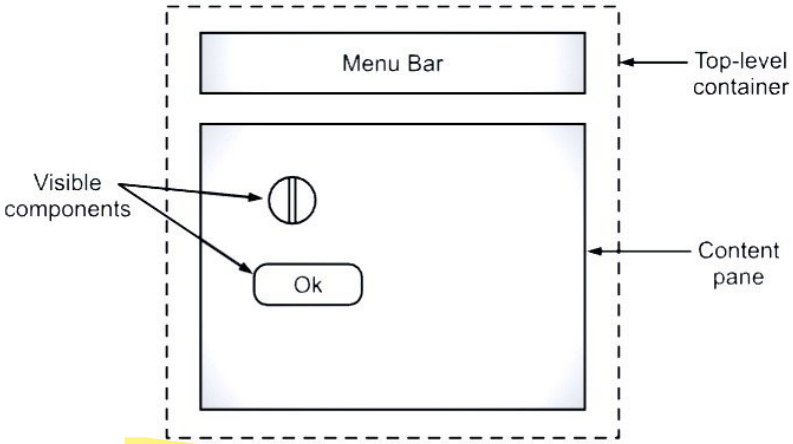


Fig. 4.1: Relationship between the Components

- The general procedure to create top-level container is:
  1. Create the container.
  2. Set its size (in the case of frames and panels).
  3. Set its visibility (in the case of frames and panels).
- All swing components names start with 'J'. For instance the swing button class is named as **JButton** commonly used top-level containers.
- A Swing application requires a top-level container. There are some top-level containers in Swing:
  1. **JFrame:** Used for the application's main window (with an icon, a title, minimize/maximize/close buttons, an optional menu-bar, and a content-pane), as illustrated.
  2. **JDialog:** Used for secondary pop-up window (with a title, a close button, and a content-pane).
- There are secondary containers (such as JPanel) which can be used to group and layout relevant components.
- Panels are an example of intermediate containers.

4.1.3 Advantages and Disadvantages of Swing

Advantages:

1. When you build your own component, swing provides paint debugging support for it.
2. Swing components are lightweight.
3. Swing components follow the Model-View-Controller (MVC) paradigm and thus can provide a much more flexible UI.
4. Swing provides both additional components like JTable, JTree etc. and added functionality to replacement of AWT components.
5. Swing provides built-in double buffering.

Disadvantages:

1. It can be slower than AWT (all components are drawn), when you are not careful about programming.
2. Swing components might not behave exactly like native components which look like native components.
3. It requires Java 2 or a separate JAR file.

4.2 THE MVC ARCHITECTURE WITH SWING

- The design of swing component classes is based on Model-View-Controller (MVC) architecture.
- MVC is a method of building reusable components that logically separate the structure, presentation and behaviour of a component into separate pieces.
- MVC has mainly three parts:
  1. The Model, which stores the data of component.
  2. The View, which creates visual representation of the component using the data in the model.
  3. The Controller, which deals with the user interaction and modifies the model or the view in response to user action.

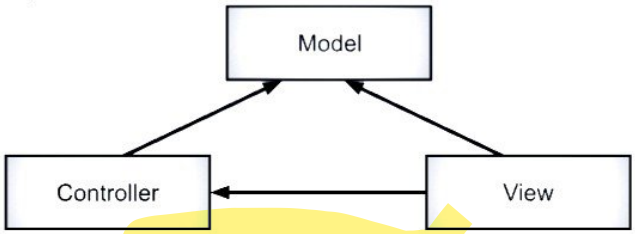


Fig. 4.2: MVC Architecture

- For example, a data in a spread sheet can be represented using different types of charts like bar chart, pie chart, etc. Here data in a spreadsheet is the model and various chart types are views.
- A single model can have many views. The changes to the model cause the changes in a view. The view and controller can be represented by a single composite object called **UI-delegate.**
- UI-delegate is responsible for the view and controller roles. It takes care of drawing the component and representing to user events.
- Fig. 4.3 shows separate Model Architecture in which the different UI-delegates can provide a component with a new look and feel. The way in which swing objects communicate, by passing events from sources to listeners is a part of MVC concept of separation.

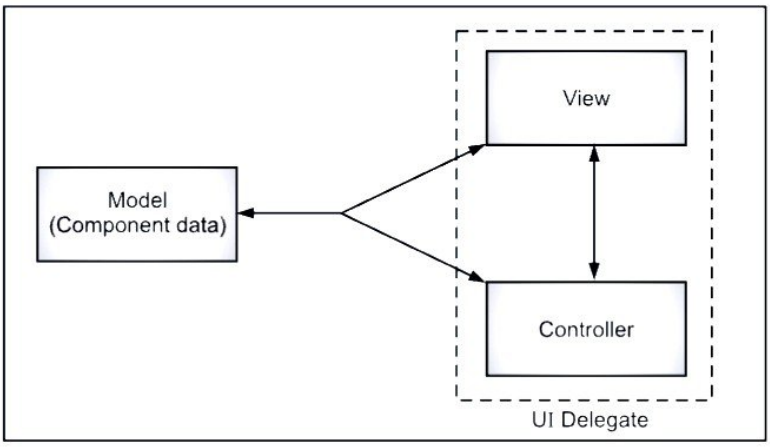


Fig. 4.3: Separable Model Architecture

- Event listeners are observers (controllers) and event sources are observerables (models). When an observable changes or performs a function, it notifies all of its observers of the activity.
- The `java.util.observer` interface and `java.util.observable` class are utilities which provides fundamental design pattern under the concept of separable model architecture.

4.3 LAYOUT MANAGERS AND LAYOUTS

- Layout means the arrangement of components within the container. In other way we can say that placing the components at a particular position within the container.
- Layout manager is an object which determines the way that components are arranged in a frame window.
- A layout manager automatically arranges our controls within a window by using some type of algorithms. A layout manager is an instance of any class that implements the `LayoutManager` interface.
- The layout manager is set by the `setLayout()` method. If no call to `setLayout()` is made, then the default layout manager is used.
- Whenever, a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.
- The setLayout( ) method has the following general Syntax:

  ```
  void setLayout(LayoutManager layoutObj)
  ```

  Here, layoutObj is a reference to the desired layout manager. If we wish to disable the layout manager and position components manually, pass null for layoutObj.
- If we do this, we will need to determine the shape and position of each component manually, using the `setBounds()` method defined by Component. Normally, we will want to use a layout manager.
- The layout manager automatically positions all the components within the container. Default values are provided otherwise.

- Java has several predefined Layout Manager Classes. Layout manager classes are founded in the java.awt package same as the classes used for displaying graphical components.
- The java.awt package provides five layout manager classes as given below:
  1. **FlowLayout:** Arranges components in variable-length rows.
  2. **BorderLayout:** Arranges components along the sides of the container and in the middle.
  3. **CardLayout:** Arranges components in "cards". Only one card is visible at a time.
  4. **GridBagLayout:** Aligns components horizontally and vertically; components can be of different sizes.
  5. **GridLayout:** Arranges components in fixed-length rows and columns.
- Let us see above flow layout managers in detail.

1. FlowLayout Manager:

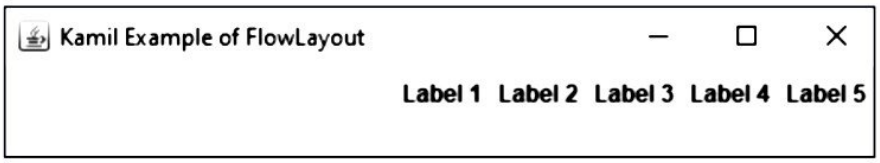
- The FlowLayout is one of the most popular and simplest layout managers. FlowLayout Manager places components in a container from left to right in the order in which they were added to the container.
- When one row is filled, layout advances to the next row. It is analogous to lines of text in a paragraph. Components managed by a FlowLayout are always set to their preferred size (both width and height) regardless of the size of the parent container.
- Whenever, the container is resized then the components in the container are also adjusted from the top-left corner. The FlowLayout is the default layout manager for a Panel and JPanel.
- Following are Constructors for FlowLayout:
  (i) `FlowLayout()`: This constructor centers all components and leaves five pixel spaces between each component.
  (ii) `FlowLayout(int align)`: This constructor allows to specify how each line of components is aligned: i.e., `Flowlayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`
  (iii) `FlowLayout(int align, int hspace, int vspace)`: This constructor allows to specify the alignment as well as the horizontal and vertical space between components.

Program 4.1: Program for FlowLayout Manager.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Example extends JFrame
{
   JLabel l1, l2, l3, l4, l5;
```

```
    public Example()
    {
        FlowLayout layout = new FlowLayout(FlowLayout.RIGHT);
        this.setLayout(layout);
        l1 = new JLabel("Label 1 ");
        l2 = new JLabel("Label 2 ");
        l3 = new JLabel("Label 3 ");
        l4 = new JLabel("Label 4 ");
        l5 = new JLabel("Label 5 ");
        this.add(l1);
        this.add(l2);
        this.add(l3);
        this.add(l4);
        this.add(l5);
    }
}
class Main
{
    public static void main(String[] args)
    {
        Example f = new Example();
        f.setTitle("Kamil Example of FlowLayout");
        f.setBounds(200, 100, 600, 400);
        f.setVisible(true);
    }
}
```

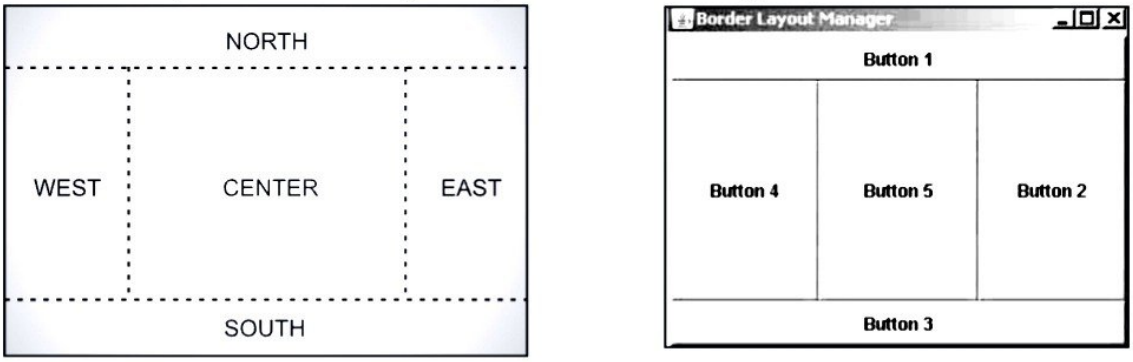
Output:



2. BorderLayout Manager:

- The BorderLayout class implements a common layout style for top-level windows.
- It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north (upper region), south (lower region), east (left region), and west (right region). The middle area is called the center (central region).

- Five zones, one for each component, as can be seen in Fig. 4.4.



(a) (b)

Fig. 4.4: Border Layout Manager

- The constructors defined by BorderLayout as follows:
  - **(i)** `BorderLayout()`: This constructor creates a default border layout.
  - **(ii)** `BorderLayout(int hspace, int vspace)`: This constructor allows us to specify the horizontal and vertical space left between components in horizontal and vertical, respectively.
- To add components in a frame using BorderLayout manager, following form of add method is used:

```
void add (Component_obj, Object region)
```

Where, region specifies any of the following five constants:

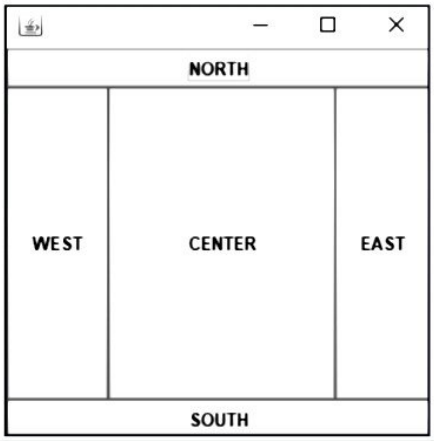
- **(i)** `BorderLayout.center`: Places component at center.
- **(ii)** `BorderLayout.EAST`: Places component at right border.
- **(iii)** `BorderLayout.WEST`: Places component at left border.
- **(iv)** `BorderLayout.SOUTH`: Places component at lower border.
- **(v)** `BorderLayout.NORTH`: Places component at upper border.

Program 4.2: Program for Border Layout.

```
import java.awt.*;
import javax.swing.*;
public class Main
{
    JFrame f;
    Main ()
    {
        f=new JFrame();

        JButton b1=new JButton("NORTH");;
        JButton b2=new JButton("SOUTH");;
```

```
        JButton b3=new JButton("EAST");;
        JButton b4=new JButton("WEST");;
        JButton b5=new JButton("CENTER");;

        f.add(b1,BorderLayout.NORTH);
        f.add(b2,BorderLayout.SOUTH);
        f.add(b3,BorderLayout.EAST);
        f.add(b4,BorderLayout.WEST);
        f.add(b5,BorderLayout.CENTER);

        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new Main ();
    }
}
```

Output:



3. GridLayout Manager:

- GridLayout lays out components in a two-dimensional grid. When we instantiate a GridLayout, we define the number of rows and columns.
- The GridLayout manager divides the container into a rectangular grid so that, component can be placed in rows and column. The intersection of each row and column is known as a cell.
- The components are laid out in cells and each cell has the same size, components are added to a GridLayout starting at the top left cell of the grid and continuing to the right until the row is full. The process continues left to right on the next row of the grid and so on.

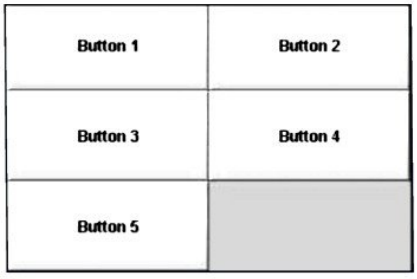


Fig. 4.5: Grid Layout Manager

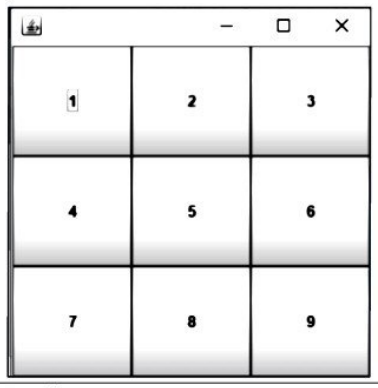
- The constructors supported by GridLayout are shown here:
  - **(i)** `GridLayout()`: Creates a single-column grid layout.
  - **(ii)** `GridLayout(int numRows, int numColumns)`: Creates a grid layout with the specified number of rows and columns.
  - **(iii)** `GridLayout(int numRows, int numColumns, int horz, int vert)`: Allows us to specify the horizontal and vertical space left between components in horz and vert, respectively. Either numRows or numColumns can be zero. Specifying numRows as zero allows for unlimited-length columns.

Program 4.3: Program for Grid layout.

```
import java.awt.*;
import javax.swing.*;
public class Main
{
   JFrame f;
   Main()
   {
      f=new JFrame();

      JButton b1=new JButton("1");
      JButton b2=new JButton("2");
      JButton b3=new JButton("3");
      JButton b4=new JButton("4");
      JButton b5=new JButton("5");
      JButton b6=new JButton("6");
      JButton b7=new JButton("7");
      JButton b8=new JButton("8");
      JButton b9=new JButton("9");
      f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
      f.add(b6);f.add(b7);f.add(b8);f.add(b9);
      f.setLayout(new GridLayout(3,3));
      f.setSize(300,300);
      f.setVisible(true);
   }
```

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

Output:



4.3.1 JComponent Class

- The class JComponent is the base class for all Swing components except top-level containers. To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level Swing container.
- Fig. 4.6 shows the class hierarchy of the swing GUI classes.
- Every swing controls inherits properties from following Component class hierarchy. A Container is the abstract base class for the non menu user-interface controls of swing.
- Component represents an object with graphical representation. A Container is a component that can contain other SWING components.

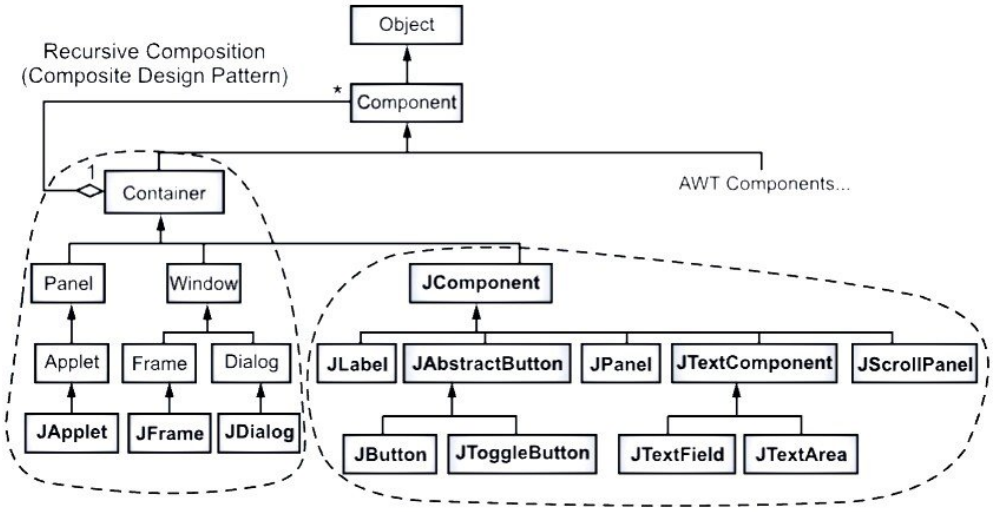


Fig. 4.6: Class hierarchy of the swing GUI classes

- Fig. 4.6 shows following classes of JComponent
  1. **AbstractButton:** Abstract super-class for Swing buttons.
  2. **ButtonGroup:** Encapsulates a mutually exclusive set of buttons.
  3. **ImageIcon:** Encapsulates an icon.
  4. **JButton:** The Swing push button class.
  5. **JCheckBox:** The Swing check box class.
  6. **JComboBox:** Encapsulates a combo box, (a combination of a dropdownlist and text field).
  7. **JLabel:** The Swing version of a label.
  8. **JRadioButton:** The Swing version of a radio button.
  9. **JScrollPane:** Encapsulates a scrollable window.
  10. **JTabbedPane:** Encapsulates a tabbed window.
  11. **JTable:** Encapsulates a table-based control.
  12. **JTextField:** The Swing version of a text field.
  13. **JTree:** Encapsulates a tree-based control.

4.4 COMPONENTS – JLABEL, JBUTTON, JTEXTFIELD, JTEXTAREA, JCHECKBOX, JRADIOBUTTON, JLIST, JCOMBOBOX, JMENU AND JPOPUPMENU, JMENUITEM CLASS

- Swing components are not implemented by platform specific code. Instead they are written entirely in Java and therefore, are platform independent.
- Swing component is called lightweight, as it does not depend on any non-java system classes.
- Swing components have pluggable look and feel so that with a single set of components you can achieve the same look and feel on any operating system platform.
- The swing related classes are contained in javax.swing and its subpackages, such as javax.swing.tree.
- To use a swing class, either use an import statement for a specific class to be imported from within a swing or import all classes in the swing package as:

```
javax.swing.*;
```

- Swing controls or components and their related methods are available in javax.swing and its subpackages.

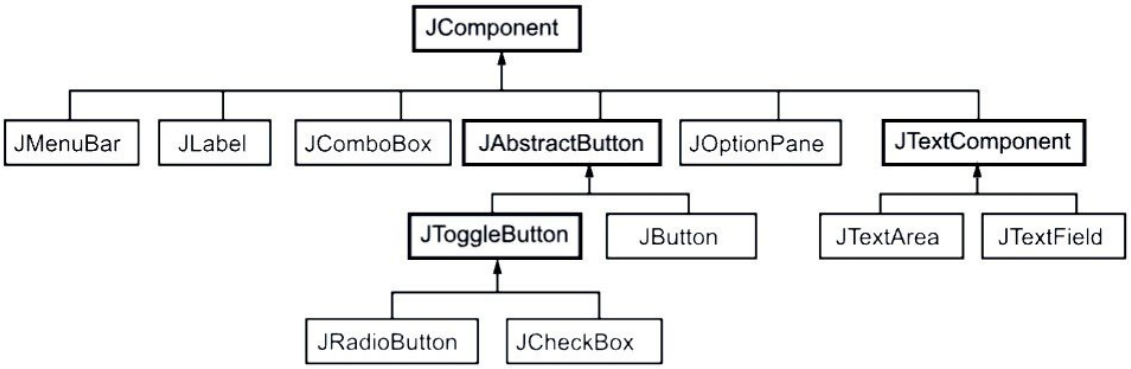


Fig. 4.7: Components of Swing

4.4.1 JLabel

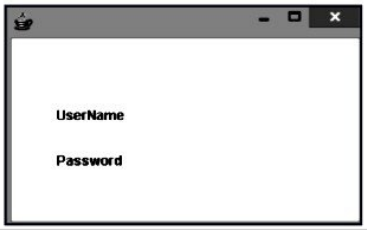
- One of the simplest Swing component is JLabel. A JLabel object is a component for placing text in a container. This class is used to create single line read only text which describes the other component.
- Labels can display text as well as Images.
- **Package:** `javax.swing`
- **Constructors**:
  1. `JLabel()`: Create a empty label having no text and icon.
  2. `JLabel(String str)`: Create a label having some text.
  3. `JLabel(ImageIcon i)`: Creates a label having icon image.
  4. `JLabel(String str,ImageIcon i)`:Creates a label having string text and icon image.
- **Methods:**
  1. `void setHorizontalAlignment(int a)`: This method sets the alignment of the text.
  2. `String getText()`: This method returns the text associated with the Label.
  3. `void setText( String s)`: This method is used to set the text to the Label.
  4. `void set icon(Icon i)`: This method sets the Icon to icon.

Program 4.4: Program to demonstrate JLabel.

```
import java.awt.Dimension;
import java.awt.Frame;
import java.awt.Rectangle;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextArea;
```

```
public class Frame1 extends JFrame
{
   private JLabel jLabel1 = new JLabel();
   private JLabel jLabel2 = new JLabel();
   public Frame1()
   {
      try
      {
         jbInit();
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
   private void jbInit() throws Exception
   {
      this.getContentPane().setLayout( null );
      this.setSize( new Dimension(400, 300) );
      jLabel1.setText("UserName");
      jLabel1.setBounds(new Rectangle(40, 55, 80, 25));
      jLabel2.setText("Password");
      jLabel2.setBounds(new Rectangle(40, 95, 60, 25));
      this.getContentPane().add(jLabel2, null);
      this.getContentPane().add(jLabel1, null);
   }
   public static void main(String args[])
   {
      Frame1 frame=new Frame1();
      frame.setVisible(true);
   }
}
```

Output:



JIcons:

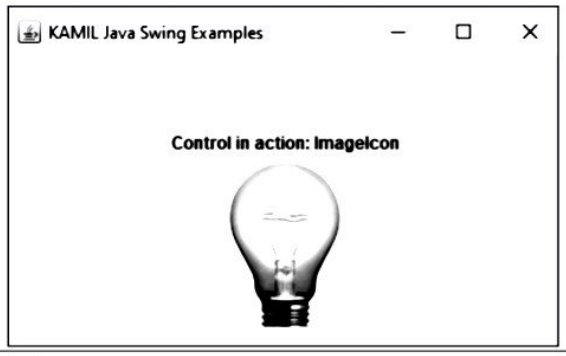
- Icons are encapsulated by the ImageIcon class, which prints an icon from an image.
- **Constructors:**
  1. **`ImageIcon(String filename)`:** This constructor creates object with image which is specified by filename.
  2. **`ImageIcon(URL url)`:** This constructor creates object with image in the resource identified by url.
- **Methods:**
  1. **`int getIconHeight()`:** Returns the height of the icon in pixels.
  2. **`int getIconWidth()`:** Returns the width of the icon in pixels.
  3. **`void paintIcon(Component comp, Graphics g, int x, int y)`:** Paints the icon at position x, y on the graphics context g. Additional information about the paint operation can be provided in component.

Program 4.5: Following program of demonstrate icons.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class SwingControlDemo
{
   private JFrame mainFrame;
   private JLabel headerLabel;
   private JLabel statusLabel;
   private JPanel controlPanel;
   public SwingControlDemo()
   {
      prepareGUI();
   }
   public static void main(String[] args)
   {
      SwingControlDemo  swingControlDemo = new SwingControlDemo();
      swingControlDemo.showImageIconDemo();
   }
   private void prepareGUI()
   {
      mainFrame = new JFrame("KAMIL Java Swing Examples");
      mainFrame.setSize(400,400);
      mainFrame.setLayout(new GridLayout(3, 1));
```

```
        mainFrame.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent windowEvent)
            {
                System.exit(0);
            }
        });
        headerLabel = new JLabel("", JLabel.CENTER);
        statusLabel = new JLabel("",JLabel.CENTER);
        statusLabel.setSize(350,100);
        controlPanel = new JPanel();
        controlPanel.setLayout(new FlowLayout());
        mainFrame.add(headerLabel);
        mainFrame.add(controlPanel);
        mainFrame.add(statusLabel);
        mainFrame.setVisible(true);
    }
    // Returns an ImageIcon, or null if the path was invalid.
    private static ImageIcon createImageIcon(String path,
    String description)
    {
        java.net.URL imgURL = SwingControlDemo.class.getResource(path);
        if (imgURL != null)
        {
            return new ImageIcon(imgURL, description);
        }
        else
        {
            System.err.println("Couldn't find file: " + path);
            return null;
        }
    }
    private void showImageIconDemo()
    {
        headerLabel.setText("Control in action: ImageIcon");
        ImageIcon icon = createImageIcon("on.png","Java");
        JLabel commentlabel = new JLabel("", icon,JLabel.CENTER);
        controlPanel.add(commentlabel);
        mainFrame.setVisible(true);
    }
```

Output:



4.4.2 JButton

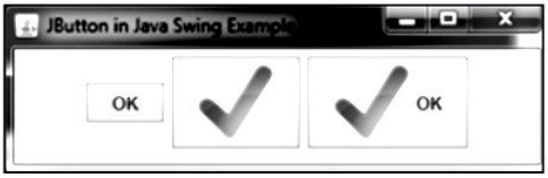
- This class is used to create a push buttons.
- **Package:** `javax.swing`
- **Constructors:**
  1. `JButton( ):` Create a empty button having no title and icon.
  2. `JButton(String str):` Create a button having label.
  3. `JButton(Icon i):` Creates a button having icon image.
  4. `JButton(String str, Icon i):` Creates a button having string label and icon image.
- This class is subclass of Abstract Button class. So it uses following important methods of abstract button class.
  1. `void addActionListenerActionListener obj):` This method is used to register the push button to throw an event.
  2. `String getText():` This method returns the text associated with the button.
  3. `void setText(String s):` This method is used to set new Label of the button.
  4. `void setHorizontalTextPosition(int pos):` Sets the horizontal text position relative to the graphics.
  5. `void setVerticalTextPosition(int pos):` Sets the vertical text position relative to the graphics.
  6. `void setIcon( Icon i):` This method sets the specified Icon to the button.
  7. `void setMnemonic(char c):` Sets the mnemonic character to the button.

Program 4.6: Program of demonstrate JButton.

```
import javax.swing.* ;
import java.awt.*;
class JButtonExample extends JFrame
{
    JButtonExample()
    {
        setLayout(new FlowLayout());
```

```
        JButton btnOk = new JButton("OK");
        ImageIcon icon = new ImageIcon("check.png");
        JButton btnIcon = new JButton(icon);
        JButton btnTxtIcon = new JButton("OK",icon);
        add(btnOk);
        add(btnIcon);
        add(btnTxtIcon);
    }
}
class JButtonJavaExample
{
    public static void main(String args[])
    {
        JButtonExample frame = new JButtonExample();
        frame.setTitle("JButton in Java Swing Example");
        frame.setBounds(200,250,250,100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Output:



4.4.3 JTextField

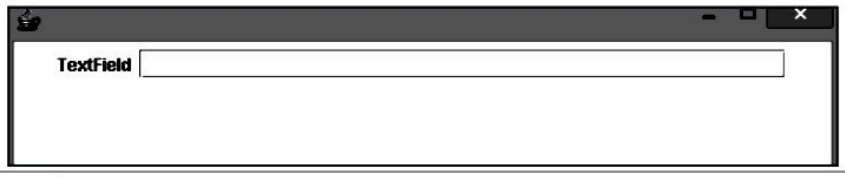
- The JTextField component allows us to enter/edit a single line of text.
- Following are the important constructors of the subclasses of the JTextField class.

1. **`JTextField()`:** Constructs a new TextField.
2. **`JTextField(Document doc, String text, int columns)`:** Constructs a new JTextField that uses the given text storage model and the given number of columns.
3. **`JTextField(int columns)`:** Constructs a new empty TextField with the specified number of columns.
4. **`JTextField(String text)`:** Constructs a new TextField initialized with the specified text.
5. **`JTextField(String text, int columns)`:** Constructs a new TextField initialized with the specified text and columns.

Program 4.7: Program to demonstrate JTextField.

```
import java.awt.*;
import javax.swing.*;
public class TextFieldDemo extends JFrame
{
    public TextFieldDemo()
    {
        Container con = getContentPane();
        con.setLayout(new FlowLayout());
        JLabel j1 = new JLabel ("TextField");
        con.add(j1);
        JTextField tf1 = new JTextField(40);
        con.add(tf1);
        setSize(600, 600);
        setVisible(true);
    }
    public static void main(String args[])
    {
        new TextFieldDemo();
    }
}
```

Output:



4.4.4 JTextArea

- The JTextArea is used to accept several lines of text from the user.
- JTextArea class itself does not handle scrolling but implements the Scrollable interface which allows it to be placed inside a JScrollPane and thus implement scrolling.
- Text wrapping is possible through the use of a bound property of JTextArea class.
- A JTextArea can be created using one of the following constructors:
  1. **`JTextArea()`:** Constructs a new TextArea.
  2. **`JTextArea(Document doc)`:** Constructs a new JTextArea with the given document model and defaults for all of the other arguments (null, 0, 0).
  3. **`JTextArea(Document doc, String text, int rows, int columns)`:** Constructs a new JTextArea with the specified number of rows and columns and the given model.

4. **`JTextArea(int rows, int columns)`:** Constructs a new empty TextArea with the specified number of rows and columns.
5. **`JTextArea(String text)`:** Constructs a new TextArea with the specified text displayed.
6. **`JTextArea(String text, int rows, int columns)`:** Constructs a new TextArea with the specified text and number of rows and columns.

Program 4.8: Program to demonstrate JTextArea.

```
import java.awt.*;
import javax.swing.*;
public class TextAreaapplextends JFrame
{
   public TextAreaappl()
   {
      Container con = getContentPane();
      con.setLayout(new FlowLayout());
      JLabel j11 = new JLabel ("TextArea");
      con.add(j11);
      JTextArea ta1 = new JTextArea(10, 20);
      con.add(ta1);
      setSize(600, 600);
      setVisible(true);
   }
   public static void main(String args[])
   {
      new TextAreaappl();
   }
}
```

Output:

- The above program creates a standalone Swing application having a frame window with a text area and a label. To create the text area that is ten rows and twenty columns wide we use the constructor:

```
JTextArea ta1= new JTextArea(10, 20);
```

4.4.5 JCheckBox [W-18]

- A checkbox is a control that may be turned ON or OFF by the user to indicate some option. The class `JCheckBox` is an implementation of a checkbox - an item that can be selected or deselected, and which displays its state to the user.
- The JCheckBox class is used to create CheckBox in Swing.
- **Constructors**:
  1. **`JCheckBox()`:** Creates an initially unselected checkbox button with no text, no icon.
  2. **`JCheckBox(Action a)`:** Creates a checkbox where properties are taken from the Action supplied.
  3. **`JCheckBox(Icon icon)`:** Creates an initially unselected checkbox with an icon.
  4. **`JCheckBox(Icon icon, boolean selected)`:** Creates a checkbox with an icon and specifies whether or not it is initially selected.
  5. **`JCheckBox(String text)`:** Creates an initially unselected checkbox with text.
  6. **`JCheckBox(String text, boolean selected)`:** Creates a checkbox with text and specifies whether or not it is initially selected.
  7. **`JCheckBox(String text, Icon icon)`:** Creates an initially unselected checkbox with the specified text and icon.
  8. **`JCheckBox(String text, Icon icon, boolean selected)`:** Creates a checkbox with text and icon, and specifies whether or not it is initially selected.

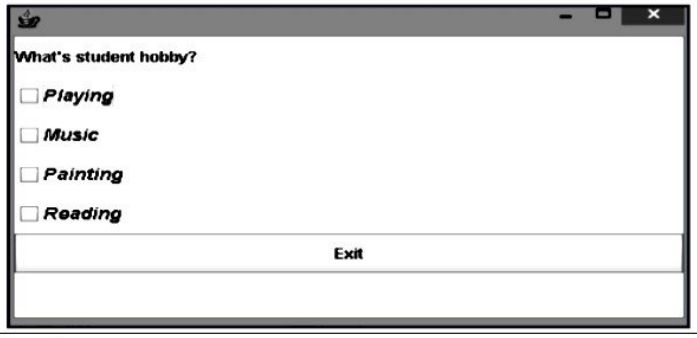
Program 4.9: Program to demonstrate JCheckBox.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class HobbyOfStudent extends JPanel implements ActionListener,ItemListener
{
    JCheckBox ch1 = new JCheckBox("Playing", false);
    JCheckBox ch2 = new JCheckBox("Music", false);
    JCheckBox ch3 = new JCheckBox("Painting", false);
    JCheckBox ch4 = new JCheckBox("Reading", false);
    JLabel j1 = new JLabel("What's student hobby?");
    JButton exitbtn = new JButton("Exit");
    public HobbyOfStudent()
    {
        setLayout(new GridLayout(7,1));
        ch1.setFont(new Font("Arial", Font.BOLD | Font.ITALIC, 14));
        ch2.setFont(new Font("Arial", Font.BOLD | Font.ITALIC, 14));
```

```
        ch3.setFont(new Font("Arial", Font.BOLD | Font.ITALIC, 14));
        ch4.setFont(new Font("Arial", Font.BOLD | Font.ITALIC, 14));
        ch1.addItemListener(this);
        ch2.addItemListener(this);    //this is event handling, so try
                                      first without these statements
        ch3.addItemListener(this);
        ch4.addItemListener(this);
        add(j1);
        add(ch1);
        add(ch2);
        add(ch3);
        add(ch4);
        add(exitbtn);
        exitbtn.addActionListener(this);        //event handling
    }
    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource().equals(exitbtn) )
        {
            System.exit(0);                     //Methods of event handling
        }
    }
    public void itemStateChanged(ItemEvent e)
    {
        String selected= ((JCheckBox)e.getSource()).getText();
        System.out.println(selected);
    }
}
class HobbyTest extends JFrame
{
    HobbyTest()
    {
        super();
        getContentPane().add(new HobbyOfStudent());
        setSize(600,600);
        setVisible(true);
    }
```

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

Output:



4.4.6 JRadioButton

- This class is used to create radio button with a text or icon. Radio buttons are supported by the JRadioButton class, which is a concrete implementation of AbstractButton.
- Once, a radio button is created they must be kept in one group, which is created by using the ButtonGroup class Radio buttons must be configured into a group. Only one of the button in that group can be selected at any time.
- **Package:** `javax.swing`
- **Constructors:**
  1. `JRadioButton()`: Create a empty radio button having no title and icon.
  2. `JRadioButton(String l)`: Create a radio button having label.
  3. `JRadioButton(Icon i)`: Creates a radio button having icon image.
  4. `JRadioButton(String l, Icon i)`: Creates a radio button having string label and icon image.
  5. `JRadioButton(String l, Icon i,boolean selected)`: Creates a radio button having string label and icon image and default selected if true is passed.
  6. `JRadioButton(String l, boolean selected)`: Creates a radio button having string label and option for default selected policy.

Button Group:

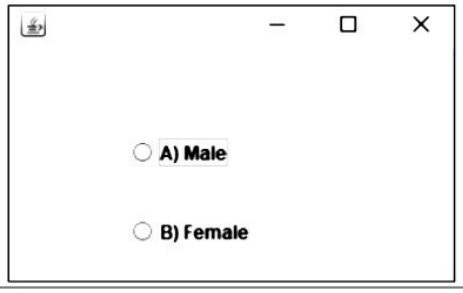
- This class is useful to group the component such as radio buttons.
- If radio button is not grouped, then they work as checkbox. And multiple selection is possible.
- To avoid that we group the Radio button object. This class provides the default constructor ButtonGroup by using this constructor object of button group class is created.

- Then different components are added to the button group object by using `add()` method.

Program 4.10: Program to demonstrate JRadioButton.

```
import javax.swing.*;
public class Main
{
    JFrame f;
    Main()
    {
        f=new JFrame();
        JRadioButton r1=new JRadioButton("A) Male");
        JRadioButton r2=new JRadioButton("B) Female");
        r1.setBounds(75,50,100,30);
        r2.setBounds(75,100,100,30);
        ButtonGroup bg=new ButtonGroup();
        bg.add(r1);bg.add(r2);
        f.add(r1);f.add(r2);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new Main();
    }
}
```

Output:



4.4.7 JList

- In Swing, the JList component is the implementation of the AWT List class. JList consists of a range of elements arranged one after another, which can be selected individually or in a group.
- JList class, unlike its AWT counterpart, is capable of displaying not just the strings, but also icons.
- Some of the constructors used for creating JList are explained below:
  1. **`public JList()`:** Constructs a JList with an empty model.
  2. **`public JList(ListModel dataModel)`:** Constructs a `JList()` that displays the elements in the specified, non-null list model.
  3. **`public JList(Object[] listData)`:** Constructs a `JList()` that displays the elements of the specified array "listData".
- `JList()` does not support scrolling. To create a scrolling list the `JList()` is implemented as the viewport view of a JScrollPane.

```
JScrollPane myScrollPane = new JScrollPane();
myScrollPane.getViewport().setView(dataList);
```

OR

```
JScrollPane myScrollPane = new JScrollPane(dataList);
```

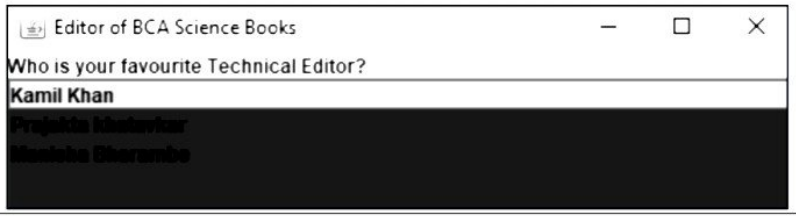
- JList does not provide any special support for handling double or triple mouse clicks. However, using the MouseListener makes it easy to handle these events. The `locationToIndex()` method is used to determine the cell that was checked.

Program 4.11: Program to demonstrate JList.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
class Employees extends JFrame implements ListSelectionListener
{
    String Employee[] = {"Kamil Khan", "Prajakta khatavkar",
    "Manisha Bharambe"};
    JList departments = new JList(Employee);
    JLabel lTE = new JLabel("Who is your favourite Technical Editor?");
    JTextField jt = new JTextField(40);
    public Employees(String s)
    {
        super(s);
        JPanel p = (JPanel)getContentPane();
```

```
        p.setLayout(new BorderLayout());
        p.add("North", lTE);
        departments.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        departments.setSelectedIndex(0);
        departments.addListSelectionListener(this);
        departments.setBackground(Color.red);
        departments.setForeground(Color.black);
        p.setBackground(Color.white);
        p.setForeground(Color.black);
        p.add("Center", new JScrollPane(departments));
        p.add("South", jt);
    }
    public static void main(String args[])
    {
        Employees e1 = new Employees ("Editor of Engineering Books");
        e1.setSize(500,500);
        e1.show();
    }
    public void valueChanged(ListSelectionEvent e)
    {
        if(e.getValueIsAdjusting() == false)
        {
            if(departments.getSelectedIndex()!=-1)
            {
                jt.setText((String)departments.getSelectedValue());
            }
        }
    }
}
```

Output:



- In JList, events are handled by implementing the ListSelectionListener interface.
  1. `public void addListSelectionListener(ListSelectionListener listener)`: This method adds a listener to the list.
  2. `public void valueChanged(ListSelectionEvent e)`: This method is called whenever the value of the selection changes.

4.4.8 JComboBox

- Swing provides a combobox, (a combination of a text field and a dropdown list) through the JComboBox class, which extends JComponent. A combobox normally displays one entry.
- However, it can also display a drop-down list that allows a user to select a different entry. We can also type our selection into the text field.
- Unlike list component, combo box allows user to select only one item at a time.
- **Constructors:**
  1. **`JComboBox()`:** This constructor creates an empty JComboBox instance.
  2. **`public JComboBox (ComboBoxModel asModel)`:** Creates a JComboBox that takes its items from an existing ComboBoxModel. asModel is the ComboBoxModel that provides the displayed list of items.
  3. **`public JComboBox (Object[] items)`:** Creates a JComboBox that contains the elements of the specified array.
- Items are added to the list of choices via the addItem( ) method, whose signature/syntax is shown here:

  ```
  void addItem(Object obj)
  ```

  Here, obj is the object to be added to the combo box.
- **Methods:**
  1. **`public void setEditable(boolean aFlag)`:** It determines whether the JComboBox field is editable or not?
  2. **`public boolean isEditable()`:** It returns true if the JComboBox is editable. By default, a combobox is not editable.
  3. **`public void setMaximumRowCount(int count)`:** It sets the maximum number of rows the JComboBox displays. If the number of objects in the model is greater than 'count', the combo box uses a scrollbar.
  4. **`public void setSelectedItem(Object anObject)`:** It sets the selected item in the combobox display area to the object in the argument. If an Object is in the list, the display area shows an Object selected.
  5. **`public void insertItemAt(Object anObject, int index)`:** It inserts an item 'anObject' into the item list at a given 'index'.
  6. **`public void removeItem(Object anObject)`:** It removes an item 'anObject' from the item list.
  7. **`public void removeItemAt(int anIndex)`:** It removes the item at 'anIndex'.

- The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for colors Green, Red, Yellow and Black. When a country is selected, the label is updated to display the color for that particular color. Color jpeg images are already stored in the current directory.

Program 4.12: Program to demonstrate JComboBox.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
class Employees extends JFrame implements ItemListener
{
   JComboBox CmbBx = new JComboBox();
   JLabel OutStrng = new  JLabel("");
   public Employees(String s)
   {
      super(s);
      Container cntnr = getContentPane();
      CmbBx.addItem("First Item");
      CmbBx.addItem("Second Item");
      CmbBx.addItem("Third Item");
      CmbBx.addItem("Fourth Item");
      CmbBx.addItem("Fifth Item");
      cntnr.setLayout(new FlowLayout());
      cntnr.add(CmbBx);
      CmbBx.addItemListener(this);
      cntnr.add(OutStrng);
   }
   public void itemStateChanged(ItemEvent e1)
   {
      if(e1.getStateChange() == ItemEvent.SELECTED)
      OutStrng.setText("Selected: " + (String)e1.getItem());
      else
      OutStrng.setText("DeSelected: " + (String)e1.getItem());
   }
   public static void main(String args[])
   {
      Employees e1 = new Employees ("Kamil Khan of BCA Science Books");
      e1.setSize(500,500);
      e1.show();
   }
}
```

Output:

4.4.9 JMenu

- A menu provides a space-saving way to let the user choose one of several options. The Menu class represents pull-down menu component which is deployed from a menu bar.
- In order to create a menu you need to use JMenu class. JMenu class represents the menu which can attach to a menu bar or another menu. Menu directly attached to a menu bar is known as top-level menu. If the menu is attached to a menu, it is called sub-menu.
- **Constructors**:
  1. `JMenu()`: This constructor creates an instance of JMenu without text.
  2. `JMenu(String s)`: This constructor creates an instance of JMenu a given text.
  3. `JMenu(String s, boolean tearOffMenu)`: This constructor creates an instance of JMenu a given text and specify the menu as a tear-off menu or not.
  4. `JMenu(Action a)`: This constructor creates an instance of JMenu whose properties are taken from the specified Action.

Program 4.13: Program for JMenu.

```
import java.awt.event.*;
import javax.swing.*;
public class JMenuExample extends JFrame implements ActionListener
{
    public static void main(String[] s)
    {
        new JMenuExample();
    }
    public JMenuExample()
    {
        super("JMenu Example");
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        };
```

```
        // Name the JMenu & Add Items
        JMenu menu = new JMenu("File");
        menu.add(makeMenuItem("Open"));
        menu.add(makeMenuItem("Save"));
        menu.add(makeMenuItem("Quit"));
        // Add JMenu bar
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(menu);
        setJMenuBar(menuBar);
        setSize(300, 300);
        setLocation(200, 200);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        // Menu item actions
        String command = e.getActionCommand();
        if (command.equals("Quit"))
        {
            System.exit(0);
        }
        else if (command.equals("Open"))
        {
            // Open menu item action
            System.out.println("Open menu item clicked");
        }
        else if (command.equals("Save"))
        {
            // Save menu item action
            System.out.println("Save menu item clicked");
        }
    }
    private JMenuItem makeMenuItem(String name)
    {
        JMenuItem m = new JMenuItem(name);
        m.addActionListener(this);
        return m;
    }
}
```

Output:

Menus and Submenus:

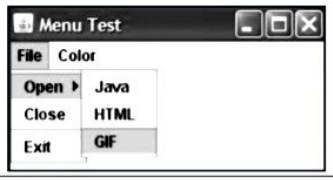
- Each menu can also have a submenu. This way we can put similar commands into groups.

Program 4.14: Program for menu and submenus.

```
import javax.swing.*;
public class JMenuTest
{
    JFrame myFrame = null;
    public static void main(String[] a) {(new JMenuTest()).test()}
    private void test()
    {
        myFrame = new JFrame("Menu Test");
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setBounds(50,50,250,150);
        myFrame.setContentPane(new JDesktopPane());
        JMenuBar myMenuBar = new JMenuBar();
        JMenu myMenu = getFileMenu();
        myMenuBar.add(myMenu);
        myMenu = getColorMenu();
        myMenuBar.add(myMenu);
        myFrame.setJMenuBar(myMenuBar);
        myFrame.setVisible(true);
    }
    private JMenu getFileMenu()
    {
        JMenu myMenu = new JMenu("File");
        JMenu mySubMenu = getOpenMenu();
        myMenu.add(mySubMenu);
        JMenuItem myItem = new JMenuItem("Close");
        myMenu.add(myItem);
        myMenu.addSeparator();
        myItem = new JMenuItem("Exit");
        myMenu.add(myItem);
        return myMenu;
    }
```

```
    private JMenu getColorMenu()
    {
        JMenu myMenu = new JMenu("Color");
        JMenuItem myItem = new JMenuItem("Red");
        myMenu.add(myItem);
        myItem = new JMenuItem("Green");
        myMenu.add(myItem);
        myItem = new JMenuItem("Blue");
        myMenu.add(myItem);
        return myMenu;
    }
    private JMenu getOpenMenu()
    {
        JMenu myMenu = new JMenu("Open");
        JMenuItem myItem = new JMenuItem("Java");
        myMenu.add(myItem);
        myItem = new JMenuItem("HTML");
        myMenu.add(myItem);
        myItem = new JMenuItem("GIF");
        myMenu.add(myItem);
        return myMenu;
    }
}
```

Output:



4.4.10 JPopupMenu

- Another type of a menu is a popup menu. Java Swing has a JPopupMenu class for this functionality.
- It is also called a context menu and usually shown when we right click on a component. The idea is to provide only the commands that are relevant in the current context. Say we have an image. By right clicking on the image, we get a popup window with commands to save, scale, or move the image.
- Popup menu represents a menu which can be dynamically popped up at a specified position within a component.

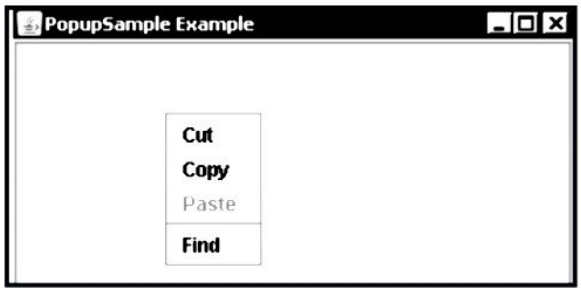
- Popup menu is a free-floating menu which associates with an underlying component. This component is called the invoker. Most of the time, popup menu is linked to a specific component to display context-sensitive choices.
- In order to create a popup menu, you use the class JPopupMenu. Then you can add menu itemsJMenuItem to popup menu like normal menu. To display the popup menu, you call method show(). Normally popup menu is called in response to a mouse event.
- **Constructors:**
  1. **`JPopupMenu()`:** This constructor constructs a JPopupMenu without an "invoker".
  2. **`JPopupMenu(String label)`:** This constructor constructs a JPopupMenu with the specified title.

Program 4.15: Program for JPopMenu.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JPopupMenu;
public class PopupSample
{
    public static void main(final String args[])
    {
        JFrame frame = new JFrame("PopupSample Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Create popup menu, attach popup menu listener
        JPopupMenu popupMenu = new JPopupMenu("Title");
        // Cut
        JMenuItem cutMenuItem = new JMenuItem("Cut");
        popupMenu.add(cutMenuItem);
        // Copy
        JMenuItem copyMenuItem = new JMenuItem("Copy");
        popupMenu.add(copyMenuItem);
        // Paste
        JMenuItem pasteMenuItem = new JMenuItem("Paste");
        pasteMenuItem.setEnabled(false);
        popupMenu.add(pasteMenuItem);
        // Separator
        popupMenu.addSeparator();
        // Find
        JMenuItem findMenuItem = new JMenuItem("Find");
```

```
        popupMenu.add(findMenuItem);
        JButton label = new JButton();
        frame.add(label);
        label.setComponentPopupMenu(popupMenu);
        frame.setSize(350, 250);
        frame.setVisible(true);
    }
}
```

Output:



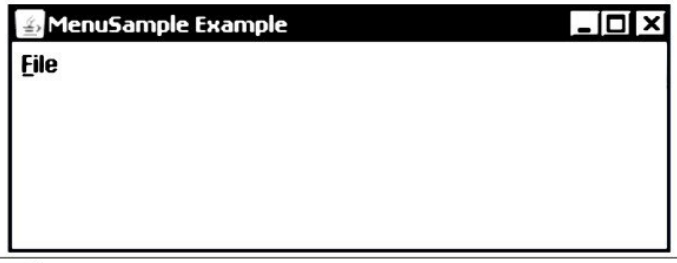
4.4.11 JMenuItem

- The JMenuItem class represents the actual item in a menu. All items in a menu should derive from class JMenuItem, or one of its subclasses. By default, it embodies a simple labeled menu item.
- In order to create menu items in Swing, you need to create new instances of JMenuItem and set different properties for them. You can create menu item with both text and icon.
- **Constructors:**
  1. `JMenuItem()`: This constructor creates a JMenuItem instance without icon or text.
  2. `JMenuItem(Icon icon)`: This constructor creates a JMenuItem instance with a given icon.
  3. `JMenuItem(String text)`: This constructor creates a JMenuItem instance with a given text.
  4. `JMenuItem(String text, Icon icon)`: This constructor creates a JMenuItem instance with a given text and icon.
  5. `JMenuItem(String text, int mnemonic)`: This constructor creates a JMenuItem instance with the given text and keyboard mnemonic.
  6. `JMenuItem(Action a)`: This constructor creates a JMenuItem instance whose properties are taken from the a given Action.

Program 4.16: Program for JMenuItem.

```
import java.awt.event.KeyEvent;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
public class MenuCreation
{
    public static void main(final String args[])
    {
        JFrame frame = new JFrame("MenuSample Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JMenuBar menuBar = new JMenuBar();
        // File Menu, F - Mnemonic
        JMenu fileMenu = new JMenu("File");
        fileMenu.setMnemonic(KeyEvent.VK_F);
        menuBar.add(fileMenu);
        // File->New, N - Mnemonic
        JMenuItem newMenuItem = new JMenuItem("New", KeyEvent.VK_N);
        fileMenu.add(newMenuItem);
        frame.setJMenuBar(menuBar);
        frame.setSize(350, 250);
        frame.setVisible(true);
    }
}
```

Output:



4.5 DIALOGS (MESSAGE, CONFIRMATION, INPUT)

- A dialog can be either modal or modeless. Modal dialogs block input to other top-level windows. Modeless dialogs allow input to other windows.
- This class extends java.awt.Dialog class. Dialogs are used to accept some inputs from User.
- Default layout manager for JDialog class is BorderLayout.
- **Package:** `javax.swing`

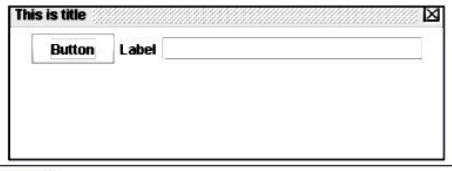
- **Constructors:**
  1. **`JDialog(Frame parent):`** This constructor creates a new JDialog object which appears on specified parent Frame.
  2. **`JDialog(Frame parent, Boolean modal):`** This constructor creates a new JDialog object which appears on the specified parent frame. If we pass second parameter as 'true', then we can not work on the parent window when dialog is visible, such a Dialog boxes are called as Modal dialog boxes. If we pass second parameter as 'false', then we can work on the parent window when dialog is visible such a dialog box is called as Non-modal dialog box.
  3. **`JDialog (Frame parent. String title):`** This constructor creates a dialog box which have some title.
  4. **`JDialog (Frame parent. Siring title. Boolean modal):`** This constructor creates a Dialogbox which appears on specified parent frame, with title specified by the user and modal or non-modal nature is specified.
- **Methods:**
  1. **`void hide()`:** This method is used to hide the Dialog box.
  2. **`void show()`:** This method is used to show the Dialog box.
  3. **`Container getContentPanel()`:** This method returns, a Content Pane for the JDialog.
  4. **`void setLayout(LayoutManager)`:** This method sets the LayoutManager for the Dialog.
  5. **`void setJMenuBar(JMenuBar)`:** This method sets the JMenu Bar to the JDialog box.
  6. **`boolean isModal()`:** This method checks whether, the dialog is modal or non-modal.

Program 4.17: Program to demonstrate JDialog.

```
import javax.swing.*;
import java.awt.*;
class JDialogExample extends JFrame
{
   JDialog d1;
   public JDialogExample()
   {
      createAndShowGUI();
   }
   private void createAndShowGUI()
   {
      setTitle("JDialog Example");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
        setLayout(new FlowLayout());
        JDialog.setDefaultLookAndFeelDecorated(true);
        d1=new JDialog(this,"This is title",true);
        // Set size
        d1.setSize(400,400);
        // Set some layout
        d1.setLayout(new FlowLayout());
        d1.add(new JButton("Button"));
        d1.add(new JLabel("Label"));
        d1.add(new JTextField(20));
        setSize(400,400);
        setVisible(true);
        /* Like JFrame, JDialog isn't visible, you'll have to make it
        visible. Remember to show JDialog after its parent is shown so that
        its parent is visible*/
        d1.setVisible(true);
    }
    public static void main(String args[])
    {
        new JDialogExample();
    }
}
```

Output:



4.5.1 Message Dialogs

- Message dialogs are simple dialogs that provide information to the user. The message is usually a string constant and represents the text contained in the body.
- If message is an array, it will be interpreted as a series of messages (the interpretation is recursive - each object will be interpreted according to its type).
- The class `JOptionPane` is a component which provides standard methods, to pop up a standard dialog box for a value or informs user of something.
- Following is the declaration for javax.swing.JOptionPane class:

```
public class JOptionPane
extends JComponent
implements Accessible
```

- Message dialogs are created with the `JOptionPane.showMessageDialog()` method. The messageType defines the style of message. The available options are:

Sr. No.	Symbol	Description
1.		ERROR_MESSAGE for displaying an error message.
2.		INFORMATION_MESSAGE for displaying an informational message.
3.		QUESTION_MESSAGE for displaying a query message.
4.		WARNING_MESSAGE for displaying a warning message.
5.		PLAIN_MESSAGE for displaying any other type of message.

- JOptionPane provides you with two useful static methods such as `showMessageDialog()` and showOptionDialog(). The `showMessageDialog()` method shows a very simple dialog with one button while showOptionDialog() method displays a highly customized dialogs with different buttons texts.
- Besides these methods, JOptionPane provides `showConfirmDialog()` method to ask users to confirm an action and showInputDialog() to get simple input from the users.

Program 4.18: In this program, we use showMessageDialog() method of JOptionPane to show various simple dialog with one button and different icons which represent different message types such as information, warning, error and question message.

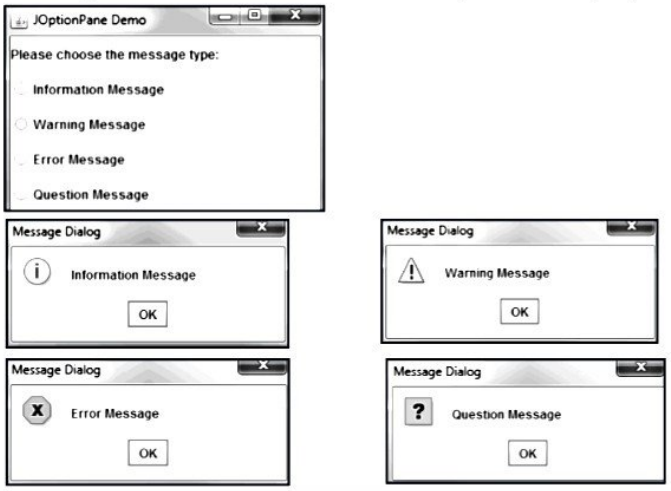
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Main
{
    public static void main(String[] args)
    {
        final JFrame frame = new JFrame("JOptionPane Demo");
        // implement ItemListener interface
        class MyItemListener implements ItemListener
        {
            public void itemStateChanged(ItemEvent ev)
            {
                boolean selected = (ev.getStateChange() == ItemEvent.SELECTED);
                AbstractButton button = (AbstractButton)ev.getItemSelectable();
                String command = button.getActionCommand();
```

```
            if (selected)
            {
               int messageType = -1;
               String message = "";
               if (command.equals("INFORMATION"))
               {
                  messageType = JOptionPane.INFORMATION_MESSAGE;
                  message = "Information Message";
               }
               else if (command.equals("WARNING"))
               {
                  messageType = JOptionPane.WARNING_MESSAGE;
                  message = "Warning Message";
               }
               else if (command.equals("ERROR"))
               {
                  messageType = JOptionPane.ERROR_MESSAGE;
                  message = "Error Message";
               }
               else if (command.equals("QUESTION"))
               {
                  messageType = JOptionPane.QUESTION_MESSAGE;
                  message = "Question Message";
               }
               // show message
               JOptionPane.showMessageDialog
                           (frame, message, "Message Dialog",messageType);
            }
         }
      }
      JRadioButton r1 = new JRadioButton("Information Message");
      r1.setActionCommand("INFORMATION");
      JRadioButton r2 = new JRadioButton("Warning Message");
      r2.setActionCommand("WARNING");
      JRadioButton r3 = new JRadioButton("Error Message");
      r3.setActionCommand("ERROR");
      JRadioButton r4 = new JRadioButton("Question Message");
      r4.setActionCommand("QUESTION");
```

```
        // add event handler
        MyItemListener myItemListener = new MyItemListener();
        r1.addItemListener(myItemListener);
        r2.addItemListener(myItemListener);
        r3.addItemListener(myItemListener);
        r4.addItemListener(myItemListener);
        // add radio buttons to a ButtonGroup
        final ButtonGroup group = new ButtonGroup();
        group.add(r1);
        group.add(r2);
        group.add(r3);
        group.add(r4);
        // Frame setting
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        Container cont = frame.getContentPane();
        cont.setLayout(new GridLayout(0, 1));
        cont.add(new JLabel("Please choose the message type:"));
        cont.add(r1);
        cont.add(r2);
        cont.add(r3);
        cont.add(r4);
        frame.setVisible(true);
    }
}
```

Output:

When you click on each radio button, it will display corresponding message type.



4.5.2 Confirmation Dialog

- A confirmation dialog is a simple dialog that asks the user to confirm an action. They are commonly used in situations where triggering an action prematurely would lead to diminished user experience or execute an irreversible application functionality.
- Typical examples are actions that result in additional work for the end user when triggered by accident like logging out of the application or storing wrong/incomplete user data. To avoid annoying the user, confirmation dialogs should not be overused.
- Actions that are easily reversible should therefore not require confirmation dialogs. An excellent example of this would be actions that trigger navigation events.
- We are using the following APIs.
  1. **`JOptionPane`:** To create a standard dialog box.
  2. **`JOptionPane.showConfirmDialog()`:** To show the confirm message box.
  3. **`JOptionPane.YES_NO_OPTION`:** To get Yes and No buttons.

Program 4.19: Program for confirmation dialog.

```
import javax.swing.JOptionPane;
public class Main
{
   public static void main(String[] argv) throws Exception
   {
      int response = JOptionPane.showConfirmDialog(null, "Kamil, Should i
      delete all files?");
  }
}
```

Output:

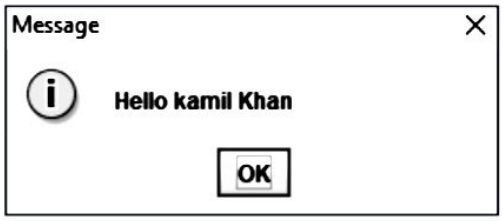
4.5.3 Input Dialog

- A dialog box that prompts the user for input.
- Useful class for accepting user input, and displaying results, is the JOptionPane class. This is located in the javax.swing library.
- Entering Text in a Dialog uses another predefined JOptionPane dialog called an input dialog that allows the user to enter data into a program. The program asks for the user's name and responds with a message dialog containing a greeting and the name that the user entered.

Program 4.20: Program for input dialog.

```
import javax.swing.JOptionPane;
public class Main
{
    public static void main(String[] args)
    {
        String name;
        name = JOptionPane.showInputDialog("What is your name? ");
        JOptionPane.showMessageDialog(null, "Hello " + name);
    }
}
```

Output:



4.5.4 JFileChooser

- File chooser dialog box is used for navigating the file system. With file chooser you can choose either file or directory from file or directory list. In order to display file chooser you use JFileChooser API.
- **Constructors:**
  1. **`JFileChooser()`:** This constructor constructs a JFileChooser pointing to the user's default directory.
  2. **`JFileChooser(File currentDirectory)`:** This constructor constructs a JFileChooser using the given File as the path.
  3. **`JFileChooser(File currentDirectory, FileSystemView fsv)`:** This constructor constructs a JFileChooser using the given current directory and FileSystemView.
  4. **`JFileChooser(FileSystemView fsv)`:** This constructor constructs a JFileChooser using the given FileSystemView.
  5. **`JFileChooser(String currentDirectoryPath)`:** This constructor constructs a JFileChooser using the given path.
  6. **`JFileChooser(String currentDirectoryPath, FileSystemView fsv)`:** This constructor constructs a JFileChooser using the given current directory path and FileSystemView.
- The return value of the three methods is one of the following:
  1. JFileChooser.CANCEL_OPTION, if the user click Cancel.
  2. JFileChooser.APPROVE_OPTION, if the user click an OK/Open/Save button.
  3. JFileChooser.ERROR_OPTION, if the user closes the dialog.

Program 4.21: Program for JFileChooser.

```
package jfilechooserdemo;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
public class Main
{
   public static void main(String[] args)
   {
      final JFrame frame = new JFrame("JFileChooser Demo");
      final JFileChooser fc = new JFileChooser();
      fc.setMultiSelectionEnabled(true);
      fc.setCurrentDirectory(new File("C:\\tmp"));
      JButton btn1 = new JButton("Show Dialog");
      btn1.addActionListener(new ActionListener())
      {
         public void actionPerformed(ActionEvent e)
         {
            fc.showDialog(frame, "Choose");
         }
      });
      JButton btn2 = new JButton("Show Open Dialog");
      btn2.addActionListener(new ActionListener())
      {
         public void actionPerformed(ActionEvent e)
         {
            int retVal = fc.showOpenDialog(frame);
            if (retVal == JFileChooser.APPROVE_OPTION)
            {
               File[] selectedfiles = fc.getSelectedFiles();
               StringBuilder sb = new StringBuilder();
               for (int i = 0; i < selectedfiles.length; i++)
               {
                   sb.append(selectedfiles[i].getName() + "\n");
               }
               JOptionPane.showMessageDialog(frame, sb.toString());
            }
         }
      });
      JButton btn3 = new JButton("Show Save Dialog");
```

```
        btn3.addActionListener(new ActionListener())
        {
            public void actionPerformed(ActionEvent e)
            {
                fc.showSaveDialog(frame);
            }
        });
        Container pane = frame.getContentPane();
        pane.setLayout(new GridLayout(3, 1, 10, 10));
        pane.add(btn1);
        pane.add(btn2);
        pane.add(btn3);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Output:

4.6 EVENT HANDLING

What is an Event?

- Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components.
- Each event must return a Boolean value (true or false), indicating whether the event should be made available to other event handlers.
- The events can be broadly classified into two categories:
  1. **Foreground Events:** Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

2. **Background Events:** Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

- Event handling is a process of responding to events that can occur at any time during execution of a program.
- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs.
- Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.
- The Delegation Event Model has the following key participants namely:
  1. An event is a change of state of an object.
  2. The source is an object on which event occurs.
  3. Listener is also known as event handler. Listener is responsible for generating response to an event.

Steps involved in Event Handling:

Step 1 : The User clicks the button and the event is generated.

Step 2 : Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.

Step 3 : Event object is forwarded to the method of registered listener class the method is now get executed and returns.

4.6.1 Event Sources

- An Event source is the object that generates an event. This occurs when the internal state of that object changes in some way.
- Sources may generate more than one type of event. A source may 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. Example: if you click a button, an ActionEvent Object is generated. The object of the ActionEvent class contains information about the event.
- The job of the event source is to accept registrations, get events from the user and call the listener's event handling method.

- Following is the table to describe some of the Event Sources.

Table 4.1: Event Sources

Event Sources	Description
1. Button	Generates action events when the button is pressed.
2. Checkbox	Generates item events when the check box is selected or deselected.
3. List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
4. Choice	Generates items events when the choice is changed.
5. MenuItem	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
6. Scrollbar	Generates adjustment events when the scrollbar is manipulated.
7. Text components	Generates text events when the user enters a character.
8. Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened or quit.

4.6.2 Listeners

- It is an object that watch for (i.e. listen for) events and handles them when they occur.
- It is basically a consumer that receives events from the source. To sum up, the job of an event listener is to implement the interface, register with the source and provide the event handling.
- The Event listener represents the interfaces responsible to handle events. Java provides us various Event listener classes but we will discuss those which are more frequently used.
- Every method of an event listener method 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.
- EventListner interface is a marker interface which every listener interface has to extend. This class is defined in java.util package.
- Following is the declaration for java.util.EventListener interface:

```
public interface EventListener
```

- Following is the list of commonly used event listeners:
  1. **`ActionListener`:** This interface is used for receiving the action events.
  2. **`ComponentListener`:** This interface is used for receiving the component events.
  3. **`ItemListener`:** This interface is used for receiving the item events.

4. `KeyListener:` This interface is used for receiving the key events.
5. `MouseListener:` This interface is used for receiving the mouse events.
6. `WindowListener:` This interface is used for receiving the window events.
7. `AdjustmentListener:` This interface is used for receiving the adjusmtent events.
8. `ContainerListener:` This interface is used for receiving the container events.
9. `MouseMotionListener:` This interface is used for receiving the mouse motion events.
10. `FocusListener:` This interface is used for receiving the focus events.

- Some of commonly used listener interfaces are as below:

1. **ActionListener:**

- The class which processes the ActionEvent should implement this interface.
- The object of that class must be registered with a component. The object can be registered using the addActionListener() method.
- **Method:**

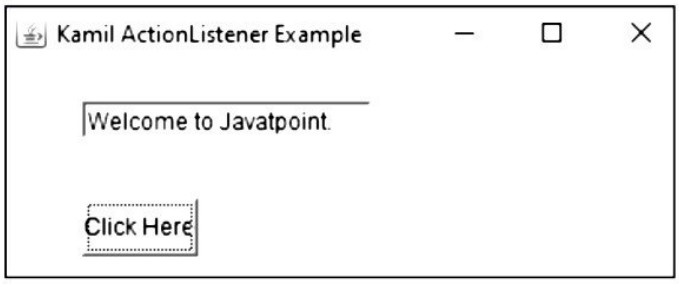
`void actionPerformed(ActionEvent e)`: This method invoked when an action occurs.

Program 4.22: Java program for ActionListener.

```
import java.awt.*;
import java.awt.event.*;
public class ActionListenerExample
{
   public static void main(String[] args)
   {
      Frame f=new Frame("Kamil ActionListener Example");
      final TextField tf=new TextField();
      tf.setBounds(50,50, 150,20);
      Button b=new Button("Click Here");
      b.setBounds(50,100,60,30);

      b.addActionListener(new ActionListener()
      {
         public void actionPerformed(ActionEvent e)
         {
            tf.setText("Welcome to Javatpoint.");
         }
      });
      f.add(b);f.add(tf);
      f.setSize(200,200);
```

```
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Output:



2. ItemListener:

- The class which processes the ItemEvent should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the `addItemListener()` method.
- **Method:**

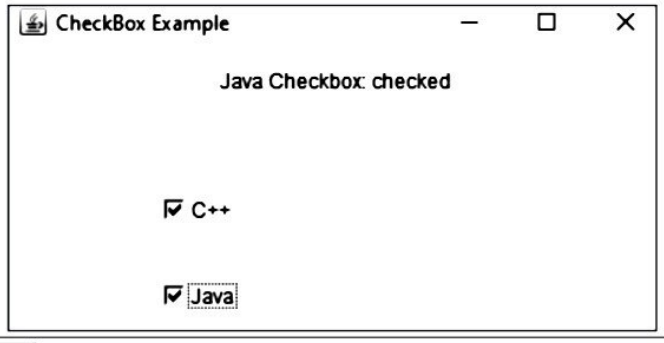
  `void itemStateChanged(ItemEvent e)`: This method invoked when an item has been selected or deselected by the user.

Program 4.23: Java program for ItemListener.

```
import java.awt.*;
import java.awt.event.*;
public class ItemListenerExample implements ItemListener
{
    Checkbox checkBox1,checkBox2;
    Label label;
    ItemListenerExample()
    {
        Frame f= new Frame("Kamil CheckBox Example");
        label = new Label();
        label.setAlignment(Label.CENTER);
        label.setSize(400,100);
        checkBox1 = new Checkbox("C++");
        checkBox1.setBounds(100,100, 50,50);
        checkBox2 = new Checkbox("Java");
        checkBox2.setBounds(100,150, 50,50);
        f.add(checkBox1); f.add(checkBox2); f.add(label);
```

```
        checkBox1.addItemListener(this);
        checkBox2.addItemListener(this);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void itemStateChanged(ItemEvent e)
    {
        if(e.getSource()==checkBox1)
            label.setText("C++ Checkbox: "
            + (e.getStateChange()==1?"checked":"unchecked"));
        if(e.getSource()==checkBox2)
        label.setText("Java Checkbox: "
        + (e.getStateChange()==1?"checked":"unchecked"));
    }
    public static void main(String args[])
    {
        new ItemListenerExample();
    }
}
```

Output:



4.7 MOUSE AND KEYBOARD EVENT HANDLING

Mouse Events:

- A mouse generates six types of event messages that you can capture in your program. These events are listed below, along with their descriptions and the method that handles them:
  1. **MOUSE_DOWN:** This event, which is handled by the `mouseDown()` method, is caused when the user presses the mouse button.

2. **MOUSE_UP:** This event is handled by the `mouseUp()` method, is caused when the user releases the left mouse button.
3. **MOUSE_MOVE:** This event is handled by the `mouseMove()` method, occurs when the user moves the mouse pointer on the screen.
4. **MOUSE_DRAG:** This event is handled by the `mouseDrag()` method, is generated when the user moves the mouse pointer while holding down the left mouse button.
5. **MOUSE_ENTER:** This event is handled by the `mouseEnter()` method, is sent when the mouse pointer enters the area owned by a component.
6. **MOUSE_EXIT:** This event is handled by the `mouseExit()` method, occurs when the mouse pointer leaves the area owned by a component.

Keyboard Events:

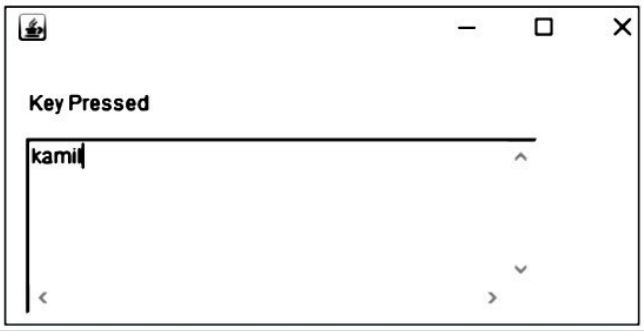
- The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent.
- The KeyListener interface is found in java.awt.event package.
- The signature of 3 methods found in KeyListener interface are given below:
  1. **KEY_PRESSED:** This event, which is handled by the keyPressed(KeyEvent e) method, is caused when the user pressed key.
  2. **KEY_RELEASED:** This event, which is handled by the keyReleased(KeyEvent e) method, is caused when the user released key.
  3. **KEY_TYPED:** This event, which is handled by the keyTyped(KeyEvent e) method, is caused when a character is typed on the keyboard, that is a key is pressed and then released.

Program 4.24: Java program for Keyboard Events.

```
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener
{
    Label l;
    TextArea area;
    KeyListenerExample()
    {
        l=new Label();
        l.setBounds(20,50,100,20);
        area=new TextArea();
        area.setBounds(20,80,300, 300);
        area.addKeyListener(this);
        add(l);add(area);
```

```
        setSize(400,400);
        setLayout(null);
        setVisible(true);
    }
    public void keyPressed(KeyEvent e)
    {
        l.setText("Key Pressed");
    }
    public void keyReleased(KeyEvent e)
    {
        l.setText("Key Released");
    }
    public void keyTyped(KeyEvent e)
    {
        l.setText("Key Typed");
    }
    public static void main(String[] args)
    {
        new KeyListenerExample();
    }
}
```

Output:



4.8 ADAPTERS – MOUSEADAPTER, KEYADAPTER

Mouse Adapter:

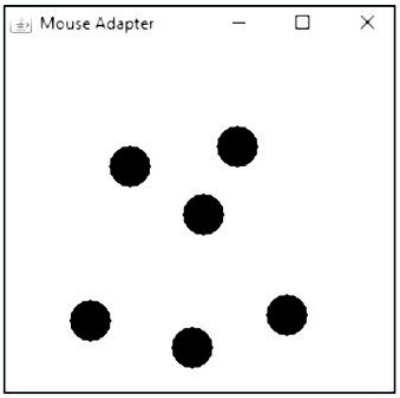
- Java adapter classes provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces.
- An abstract adapter class for receiving mouse events. The methods in this class are empty. This class exists as convenience for creating listener objects.

- Mouse events let you track when a mouse is pressed, released, clicked, moved, dragged, when it enters a component, when it exits and when a mouse wheel is moved.
- Extend this class to create a MouseEvent (including drag and motion events) or/and MouseWheelEvent listener and override the methods for the events of interest. (If you implement the MouseListener, MouseMotionListener interface, you have to define all of the methods in it. This abstract class defines null methods for them all, so you can only have to define methods for events you care about.)
- Create a listener object using the extended class and then register it with a component using the component's addMouseListener addMouseMotionListener, addMouseWheelListener methods. The relevant method in the listener object is invoked and the MouseEvent or MouseWheelEvent is passed to it in following cases:
  1. When a mouse button is pressed, released, or clicked (pressed and released).
  2. When the mouse cursor enters or exits the component.
  3. When the mouse wheel rotated, or mouse moved or dragged.

Program 4.25: Java Program for Mouse Adapter Class.

```
import java.awt.*;
import java.awt.event.*;
public class MouseAdapterExample extends MouseAdapter
{
    Frame f;
    MouseAdapterExample()
    {
        f=new Frame("Mouse Adapter");
        f.addMouseListener(this);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void mouseClicked(MouseEvent e)
    {
        Graphics g=f.getGraphics();
        g.setColor(Color.BLUE);
        g.fillOval(e.getX(),e.getY(),30,30);
    }
    public static void main(String[] args)
    {
        new MouseAdapterExample();
    }
}
```

Output:



Key Adapter

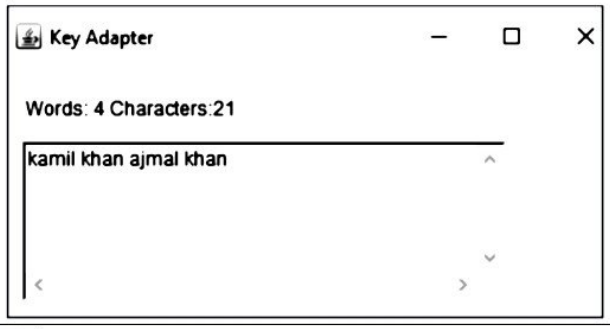
- Extend this class to create a KeyEvent listener and override the methods for the events of interest.
- If you implement the KeyListener interface, you have to define all of the methods in it. This abstract class defines null methods for them all, so you can only have to define methods for events you care about.
- Create a listener object using the extended class and then register it with a component using the component's addKeyListener method. When a key is pressed, released, or typed, the relevant method in the listener object is invoked, and the KeyEvent is passed to it.

Program 4.26: Example of Key Adapter Class.

```
import java.awt.*;
import java.awt.event.*;
public class KeyAdapterExample extends KeyAdapter
{
   Label l;
   TextArea area;
   Frame f;
   KeyAdapterExample()
   {
      f=new Frame("Key Adapter");
      l=new Label();
      l.setBounds(20,50,200,20);
      area=new TextArea();
      area.setBounds(20,80,300, 300);
      area.addKeyListener(this);
      f.add(l);f.add(area);
```

```
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void keyReleased(KeyEvent e)
    {
        String text=area.getText();
        String words[]=text.split("\s");
        l.setText("Words: "+words.length+" Characters:"+text.length());
    }
    public static void main(String[] args)
    {
        new KeyAdapterExample();
    }
}
```

Output:



4.9 ANONYMOUS INNER CLASS

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

Program 4.27: Java anonymous inner class example using class.

```
abstract class Demo
{
    abstract void walk();
}
class TestAnonymousInner
{
```

```
    public static void main(String args[])
    {
        Demo p=new Demo()
        {
            void walk(){System.out.println("nice Kamil ");
        }
    };
    p.walk();
}
```

Output:

```
C:\Windows\system32\cmd.exe

C:\Users\kamilkhan\Desktop>javac TestAnonymousInner.java

C:\Users\kamilkhan\Desktop>java TestAnonymousInner
nice Kamil

C:\Users\kamilkhan\Desktop>
```

Summary

- The swing components are defined in the package javax.swing. This package provides more powerful and flexible components.
- Swing is a Java foundation class's library and it is an extension to do Abstract Windowing Toolkit (AWT).
- Swing is an extension to the AWT components which provides feature of pluggable look and feel for the components. It provides classes to design lightweight components.
- MVC stands for Model View and Controller. MVC is a design pattern that separates the business logic, presentation logic and data.
- In MVC:
  - Controller acts as an interface between View and Model. Controller intercepts all the incoming requests.
  - Model represents the state of the application i.e. data. It can also have business logic.
  - View represents the presentaion i.e., UI (User Interface).
- The layout managers are used to arrange components in a particular manner. In layout manager is an interface that is implemented by all the classes of layout managers. Different classes that represent the layout managers are BorderLayout, FlowLayout, GridLayout and so on.
- The class JComponent is the base class for all Swing components except top-level containers.

- To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level Swing container.
- The JButton class is used to create a button that have platform-independent implementation.
- The class JLabel can display either text, an image, or both.
- The class JTextField is a component which allows the editing of a single line of text.
- The JMenuItem class represents the actual item in a menu. All items in a menu should derive from class JMenuItem, or one of its subclasses.
- The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options.
- The JTextArea class is used to create a text area. It is a multiline area that displays the plain text only.
- The JComboBox class is used to create the combobox (drop-down list). At a time only one item can be selected from the item list.
- In a computer application a dialog is a window which is used to "talk" to the application.
- A dialog is used to input data, modify data, change the application settings etc. Dialogs are important means of communication between a user and a computer program.
- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- An event source is a graphical component that is capable of firing off an event. Event sources can be any graphical component (e.g., JButton, JList, JComoboBox,JCheckBox, etc.) the user can interact with.
- Event listeners can then be used to handle the events. The Event listener represent the interfaces responsible to handle events.
- Mouse event indicates a mouse action occurred in a component. This low-level event is generated by a component object for Mouse Events and Mouse motion events,a mouse button is pressed,a mouse button is released, a mouse button is clicked (pressed and released), a mouse is moved, the mouse is dragged.

Check Your Understanding

1. _____ is a GUI widget toolkit for Java which is part of Oracle's Java Foundation Classes (JFC).
   (a) Swing (b) AWT
   (c) Servlet (d) All of the Mentioned
2. MVC stands for _____ architecture.
   (a) Model (b) View
   (c) Controller (d) All of the Mentioned

3. Change in the state of an object is known as _____ and are generated as result of user interaction with the graphical user interface components.
   (a) error (b) event
   (c) exception (d) All of the Mentioned
4. A _____ is a base class for all swing UI components.
   (a) JMenu (b) JComponent
   (c) JPanel (d) All of the Mentioned
5. The code that performs a task in response to an event is called _____ handler and a process of responding to events that can occur at any time during execution of a program is known as event handling.
   (a) event (b) error
   (c) exception (d) All of the Mentioned
6. The class _____ is a component which provides a simple mechanism for the user to choose a file.
   (a) JFrameChooser (b) JFileChooser
   (c) JFolderChooser (d) None of the Above
7. To create a menu user need to use _____ class.
   (a) JRadioButton (b) JFrame
   (c) JMenu (d) None of the Mentioned
8. 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

Answers

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

Practice Questions

Q.I Answer the following questions in short:

1. What is swing?
2. What is the purpose of layout manager?
3. List types of layout managers.
4. What repaint method does?
5. Which swing classes are used to create menu?
6. List out layout manager.
7. What is a listener?

Q.II Answer the following questions:

1. What do you mean by event listener?
2. Explain in brief Delegation Event model for handling events.
3. Enlist the features of swing?
4. Explain in brief the fundamental idea behind MVC architecture?
5. How events are generated?
6. What is event? How to handle events in AWT? Explain with example.
7. What is Keyboard Event Handling?
8. What is Mouse Event Handling?
9. Write short note on anonymous inner class.
10. Write short note on JComponent class.

Q.III Define the terms:

1. Adaptes
2. Anonymous inner class
3. Event Handling
4. Components
5. Key Adapter
6. Mouse Adapter

Previous Exam Questions

Winter 2018

1. Write any two constructors of JCheckBox class. **[1 M]**

Ans. Refer to Section 4.4.5.

■■■

5... Database Programming

Learning Objectives...

- To understand Basic Concepts of Database Programming.
- To understand Role of JDBC, JDBC Configuration.
- To study different Type of Drivers.
- To learn Connectivity with Database.
- To study different JDBC Statements.
- To learn JDBC ResultSet, MetaData etc.

5.1 INTRODUCTION TO JDBC [S-19]

- A database is an organized collection of data from which users can efficiently retrieve the desired information.
- Java Database Connectivity (JDBC) is Java application programming interface that allows the Java programmers to access DBMS through Java code. It is a java API that provides methods for querying and updating data in a database.
- JDBC classes are contained in java.sql and javax.sql packages.
- Java Database Connectivity is similar to Open Database Connectivity (ODBC) which is used for accessing and managing database, but the difference is that JDBC is designed specifically for Java programs, ODBC is not depended upon any language.
- Using java API, it is possible to access various RDBMS like Sybase, Oracle, Informix, Ingres, PostgreSQL and MySQL.
- JDBC API enables the java application to interact with different database.
- Advantages of using Java with JDBC are as follows:
  1. Easy and economical.
  2. Development time of application is short.
  3. Installation is simple.

5.1.1 The Role of JDBC

- Basically, JDBC makes it possible to do establish a connection with a data source, send queries and update statements, and process the results.

- Simply, JDBC makes it possible to do the following things within a Java application:
  1. Establish a connection with a data source.
  2. Send queries and update statements to the data source.
  3. Process the results.
- The JDBC API is implemented through the JDBC Driver. The JDBC Driver is a set of classes that implement the JDBC interfaces to process JDBC calls and return result sets to a Java Application. The database (or data store) stores the data retrieved by the application using the JDBC Driver.

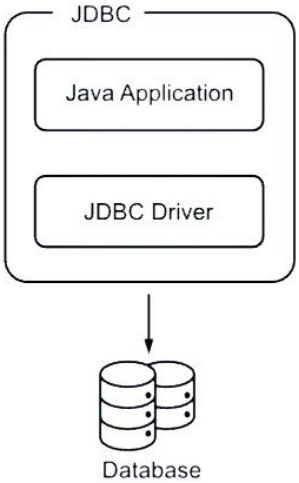


Fig. 5.1: JDBC API

- The main objects of the JDBC API include:
  - A **DataSource** object is used to establish connections. Although the Driver Manager can also be used to establish a connection, connecting through a DataSource object is the preferred method.
  - A **Connection** object controls the connection to the database. An application can alter the behavior of a connection by invoking the methods associated with this object. An application uses the connection object to create statements.
  - Statement, PreparedStatement, and CallableStatement objects are used for executing SQL statements. A **PreparedStatement** object is used when an application plans to reuse a statement multiple times. The application prepares the SQL it plans to use. Once prepared, the application can specify values for parameters in the prepared SQL statement. The statement can be executed multiple times with different parameter values specified for each execution. A **CallableStatement** is used to call stored procedures that return values. The CallableStatement has methods for retrieving the return values of the stored procedure.
  - A **ResultSet** object contains the results of a query. A ResultSet is returned to an application when a SQL query is executed by a statement object. The ResultSet object provides methods for iterating through the results of the query.

5.1.2 JDBC Configuration

- In latest version of java, we no need configure JDBC but if we use any IDE it requires to configure JDBC.
- If you define a JDBC data source in your agent, the following required, common configuration fields are added to the agent automatically:
  - **JDBC database type:** Type of database to which you are connecting, IBM® DB2®, Microsoft SQL Server, or Oracle Database Server.
  - **JDBC user name:** User name that is used to authenticate with the database server.
  - **JDBC password:** Password that is used to authenticate with the database server.
  - **Base paths:** List of directories that are searched for JAR files that are named in the Class Path field, or directories that are named in the JAR directories field that are not fully qualified. Directory names are separated by a semi-colon (;) on Windows, and by a semi-colon (;) or colon (:) on UNIX systems.
  - **Class path:** Explicitly named JAR files to be searched by the agent. Any files that are not fully qualified are appended to each of the Base Paths until the JAR file is found.
  - **JAR directories:** List of directories that are searched for JAR files. Directory names are separated by a semi-colon (;) on Windows, and by a semi-colon (;) or colon (:) on UNIX systems. The JAR files in these directories do not have to be explicitly identified; they are found because they are in one of these directories. Subdirectories of these directories are not searched. Any directories that are not fully qualified are appended to each of the Base Paths until the directory is found.
  - **JDBC connection URL:** Vendor-specific connection URL that provides details on which host the database is located and the port number to which to connect. The URL format typically looks as follows:

```
jdbc:identifier://server:port/database
```

5.2 JDBC DRIVERS

- JDBC driver is a software component enabling a Java application to interact with a database.
- To connect to each database, JDBC requires driver for each database.
- JDBC driver gives out connection to the database and implements the protocol for transferring the query and results between client and database.
- Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates.
- Sun Microsystem has defined four categories of JDBC drivers. The categories describe the differences in architecture for the drivers. One difference between architectures lies in whether a given driver is implemented in native code or in Java code.
- By native code, it means whatever machine code is supported by a particular hardware configuration. Another difference lies in how the driver makes the actual connection to the database.

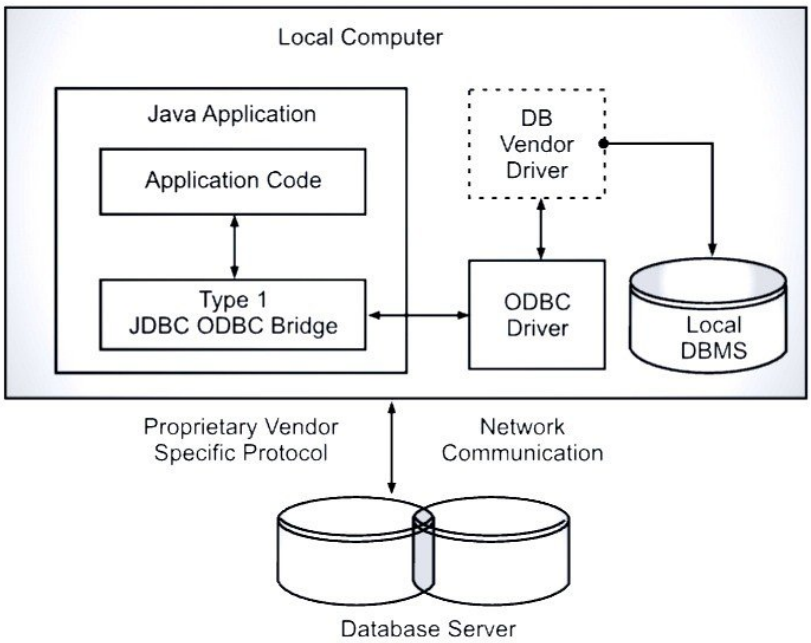


Fig. 5.2: Types of JDBC drivers

- This is a class at the start of JDBC driver. From this place a connection to the database is made. This enables a `DriverManager` class.

5.2.1 Types of Drivers [S-19]

- There are four types of JDBC drivers known as:
  1. JDBC-ODBC Bridge plus ODBC driver, also called Type 1.
  2. Native-API, partly Java driver, also called Type 2.
  3. JDBC-Net, pure Java driver, also called Type 3.
  4. Native-protocol, pure Java driver, also called Type 4.
- JDBC Driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates.

Type 1: JDBC-ODBC Bridge Driver:

- In a Type 1 driver, a JDBC Bridge is used to access ODBC drivers installed on each client machine. While using ODBC, Data Source Name (DSN) that represents target database, should be configured on your system.
- When Java first arrived, this was a useful driver because most databases only supported ODBC access. Now this type of driver is recommended only for experimental use or when no other alternative is available.
- The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API Partly Java Driver:

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers are typically provided by the database vendors and are used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.
- If we change the Database, we have to change the native API as it is specific to a database. They are out of date now but it will definitely increase the speed, because it eliminates ODBC's overhead.

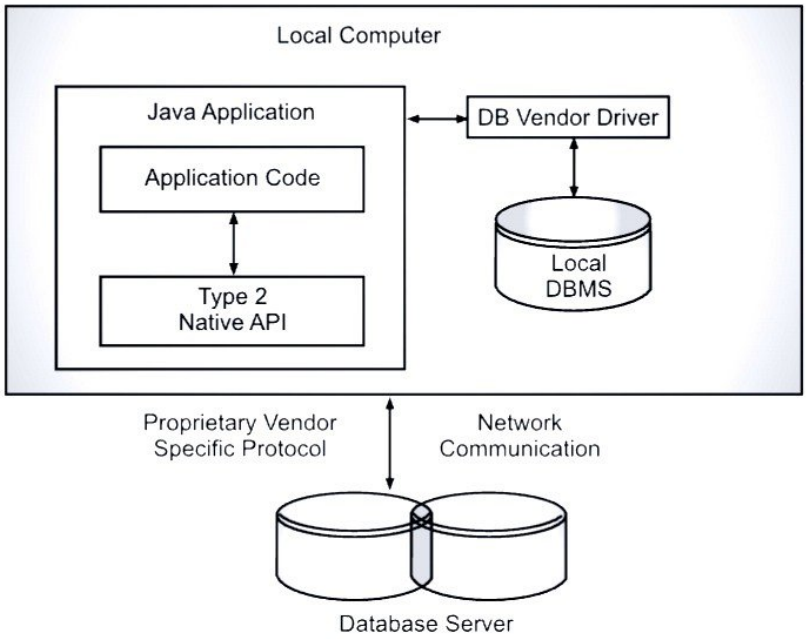


Fig. 5.3: Architecture of Type 2 Driver

- The Oracle Call Interface (OCI) driver is an example of a Type 2 Driver.

Type 3: JDBC-Net Pure Java Driver:

- In a Type 3 driver, a three-tier approach is used to access the databases. The JDBC clients use standard network sockets to communicate with middleware application server.
- The socket information is then translated by the middleware application server into the call format required by the DBMS and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installation on the client and a single driver can actually provide access to multiple databases.
- An application server can be thought of as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver's type.
- For example, your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the fine distinction will prove helpful.

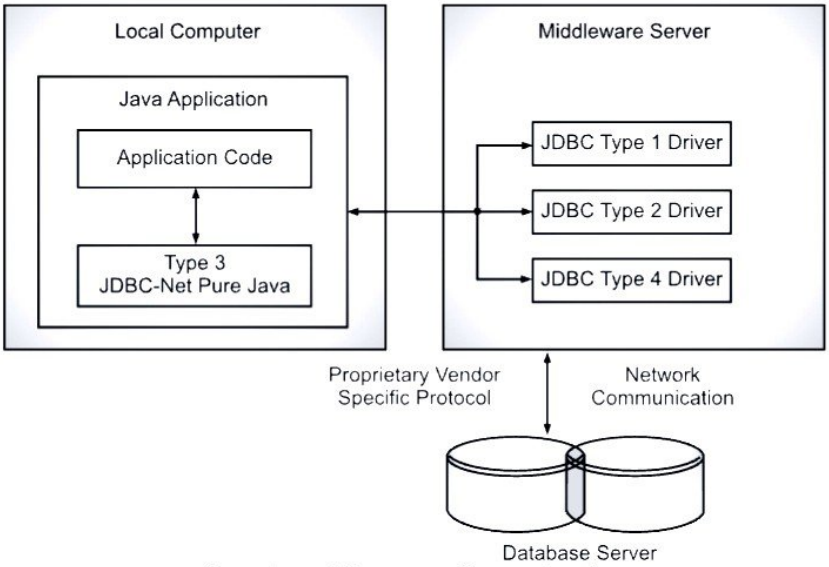


Fig. 5.4: Architecture of Type 3 Driver

- Type 3 driver is advantageous as it does not need any client installation and it does not need any native binary code on the client machine.
- The main drawback of type 3 driver is it can be difficult to set up as the architecture is complicated by the network interface.

Type 4: Native-protocol, 100% Pure Java Driver:

- In a Type 4 driver, a pure Java-based driver communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

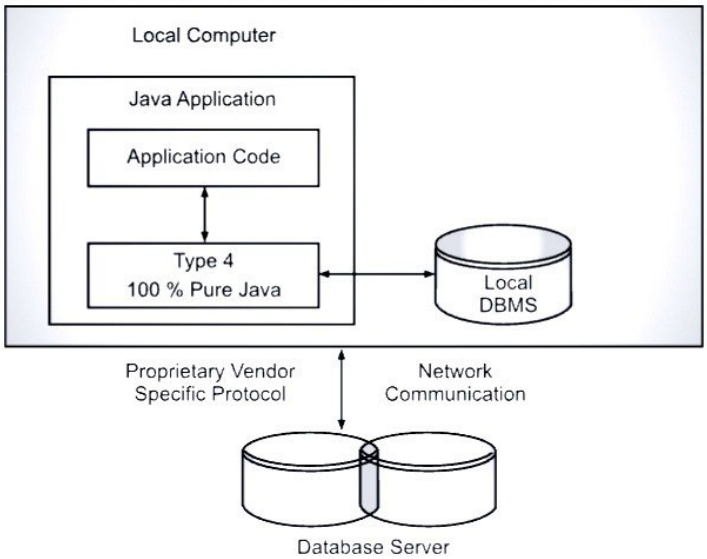


Fig. 5.5: Architecture of Type 4 Driver

- MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 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.

5.3 CONNECTIVITY WITH DATABASE (PostgreSQL)

- Steps to create connectivity to PostgreSQL:

  Step 1 : Establish a database connection to the PostgreSQL server.

  Step 2 : Create an instance of the Statement object.

  Step 3 : Execute a statement to get a ResultSet object.

  Step 4 : Process the ResultSet object.

  Step 5 : Close the database connection.

Establishing a Database Connection:

- To connect to the PostgreSQL database, you need to provide account information such as username, password, and the connection string.
- For example, method connects to a PostgreSQL database and returns a Connection object:

```
public Connection connect() throws SQLException
{
        return DriverManager.getConnection(url, user, password);
}
```

- The URL, user and password are as follows:

```
private final String url = "jdbc:postgresql://localhost/Temp";
private final String user = "postgres";
private final String password = "postgres";
```

- JDBC provides you with three kinds of Statement objects:
  1. **Statement:** You use the Statement to implement a simple SQL statement that has no parameters.
  2. **PreparedStatement:** It is the subclass of the Statement class. It gives you the ability to add the parameters to the SQL statements.
  3. **CallableStatement:** It extends the PreparedStatement class used to execute a stored procedure that may have parameters.

Executing the Query:

- To execute a query, you use one of the following methods of the Statement object:
  1. **execute:** Returns true if the first object of the query is a ResultSet object. You can get the ResultSet by calling the method getResultSet.
  2. **executeQuery:** Returns only one ResultSet object.
  3. **executeUpdate:** Returns the number of rows affected by the statement. You use this method for the INSERT, DELETE or UPDATE statement.

Processing the ResultSet Object:

- After having a ResultSet object, you use a cursor to loop through the result set by calling the methods of the ResultSet object.

5.4 JDBC STATEMENTS

5.4.1 Statement

- A Statement object represents an SQL statement. First, you create a Statement object from the Connection object. Then, you execute the Statement object to get a ResultSet object that represents a database result set.
- The Statement object cannot be directly created. The createStatement() method of Connection object returns a Statement object as shown below:

```
Statement stmt = null;
try
{
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e)
{
   . . .
}
finally
{
   . . .
}
```

- After creating a Statement object, we can then use it to execute an SQL statement with one of its three execute methods.
  - **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

- **int executeUpdate (String SQL):** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL):** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Program 5.1: Program for statement class.

```
import java.sql.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class PostgreSQLJDBC
{
   public static void main( String args[] )
   {
      Connection c = null;
      Statement stmt = null;
      try
      {
         Class.forName("org.postgresql.Driver");
         c = DriverManager
         .getConnection("jdbc:postgresql://localhost:5432/testdb",
         "kamil", "kamil");//user name password as per your system set
         System.out.println("Opened database successfully");
         stmt = c.createStatement();
         String sql = "CREATE TABLE COMPANY " +
           "(ID INT PRIMARY KEY     NOT NULL," +
           " NAME           TEXT    NOT NULL, " +
           " AGE            INT     NOT NULL, " +
           " ADDRESS        CHAR(50), " +
           " SALARY         REAL)";
         stmt.executeUpdate(sql);
         stmt.close();
         c.close();
      } catch ( Exception e )
```

```
        {
            System.err.println(e.getClass().getName()+": "+ e.getMessage());
            System.exit(0);
        }
        System.out.println("Table created successfully");
    }
}
```

Output:

```
Opened database successfully
Table created successfully
```

5.4.2 PreparedStatement

- Sometimes, it is more convenient to use a PreparedStatement object for sending SQL statements to the database. This special type of statement is derived from the Statement.
- Use of PreparedStatement object reduces execution time when the same SQL statement needs to be executed many times. When the PreparedStatement object is created, SQL statement is given to it. This SQL statement is sent to DBMS immediately where it is compiled.
- As a result, the PreparedStatement object has a precompiled SQL statement. This means that when the PreparedStatement is executed, the DBMS can just run the SQL statement without having to compile it first.
- Although PreparedStatement objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it.
- The PreparedStatement interface provides you with a means to execute dynamic SQL statements. To create a PreparedStatement object, you can call the prepareStatement method of the connection object.
- You use the following syntax for this:

```
String sql = "select * from Student where college =?";
PreparedStatement prestmt = conn.prepareStatement(sql);
```

- Notice that, in place of name of the college, we have specified '?'. You must supply values in place of the question mark placeholders (if there are any) before you execute a PreparedStatement object. Do this by calling one of the setter methods defined in the PreparedStatement class.
- PreparedStatement provides various methods for executing different types of SQL statements as well as methods for handling the insertion and removal of parameters to the SQL statements.

Table 5.1: Methods of PreparedStatement

Method	Description
clearParameters	Clears all of the parameters currently being used by the SQL statement contained in the interface.
Execute	Executes the current SQL statement or stored procedure for the nterface.
executeQuery	Executes a SQL statement or procedure that returns a result set.
executeUpdate	Executes a SQL statement that updates the database.
setAsciiStream	Sets the specified parameter to the passed ASCII stream.
setBinaryStream	Sets the specified parameter to the passed Binary stream.
setBoolean	Sets the specified parameter to the passed Boolean value.
setByte	Sets the specified parameter to the passed Byte variable.
setBytes	Sets the specified parameter to the passed Byte array.
setDate	Sets the specified parameter to the passed Date object.
setDouble	Sets the specified parameter to the passed Double value.
setFloat	Sets the specified parameter to the passed Float value.
setInt	Sets the specified parameter to the passed Integer value.
setLong	Sets the specified parameter to the passed Long value.
setNull	Sets the specified parameter to a NULL value.
setNumeric	Sets the specified parameter to a Numeric value.
setObject	Sets the specified parameter to the passed object.
setShort	Sets the specified parameter to the passed Short value.
setString	Sets the specified parameter to the passed String value.
setTime	Sets the specified parameter to the passed Time object.
setTimestamp	Sets the specified parameter to the passed Timestamp value.
setUnicodeStream	Sets the specified parameter to the passed Unicode stream.
clearParameters()	Clears all of the parameters currently being used by the dynamic SQL statement or stored procedure for the PreparedStatement object.

Program 5.2: program for Prepared statement.

```
import java.sql.*;
public class jdbcConn
{
   public static void main(String[] args) throws Exception
   {
      Class.forName("org.postgresql.Driver");
      con = DriverManager.getConnection
             ("jdbc:postgresql://localhost:5432/testdb","kamil", "kamil");
      System.out.println("Opened database successfully");
      PreparedStatement updateemp = con.prepareStatement
                                        ("insert into emp values(?,?,?)");
      updateemp.setInt(1,23);
      updateemp.setString(2,"Kamil");
      updateemp.setString(3, "CEO");
      updateemp.executeUpdate();
      Statement stmt = con.createStatement();
      String query = "select * from emp";
      ResultSet rs =  stmt.executeQuery(query);
      System.out.println("Id Name    Job");
      while (rs.next())
      {
         int id = rs.getInt("id");
         String name = rs.getString("name");
         String job = rs.getString("job");
         System.out.println(id + "  " + name+"   "+job);
      }
   }
}
```

Output:

```
Opened database successfully
Id   Name    Job
23   Kamil   CEO
```

5.4.3 CallableStatement

- Just as a Connection object creates the Statement and PreparedStatement objects. It also creates the CallableStatement object which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object:

- Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all three parameters.
- Here, are the definitions of each parameter:

Table 5.2: Parameter Definition

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement when it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

- The following code snippet shows how to employ the Connection.prepareCall() method to instantiate a CallableStatement object based on the preceding stored procedure:

```
CallableStatement cstmt = null;
try
{
   String SQL = "{call getEmpName (?,?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e)
{
   . . .
}
finally
{
   . . .
}
```

- The String variable SQL represents the stored procedure, with parameter placeholders. Using CallableStatement objects is much like using PreparedStatement objects. You must bind values to all parameters before executing the statement, or you will receive a SQLException.

Closing CallableStatement Object:

- Just as you close other Statement object, for the same reason you should also close the CallableStatement object.
- A simple call to the close() method will do the job. If you close the Connection object first it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try
{
   String SQL = "{call getEmpName (?,?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e)
{
   . . .
}
finally
{
   cstmt.close();
}
```

Program 5.3: Program for Callable Statement.

```
import java.sql.*;
public class FuncSum
{
   public static void main(String[] args) throws Exception
   {
      Class.forName("org.postgresql.Driver");
      con = DriverManager.getConnection
                              ("jdbc:postgresql://localhost:5432/testdb",
              "kamil", "kamil");//user name password as per your system set
      CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");
      stmt.setInt(2,10);
```

```
        stmt.setInt(3,43);
        stmt.registerOutParameter(1,Types.INTEGER);
        stmt.execute();
        System.out.println(stmt.getInt(1));
    }
}
```

Output:

```
53
```

Closing a Database Connection:

- In JDBC you use a try-with-resources statement to close ResultSet, Statement, and Connection objects automatically.

Querying Data Examples:

- Method returns the number of actresss in the 'actress' table using the COUNT function.

```
public int getActressCount()
{
    String SQL = "SELECT count(*) FROM Actress";
    int cnt = 0;
    try (Connection conn = connect();
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(SQL))
    {
        rs.next();
        cnt = rs.getInt(1);
    } catch (SQLException ex)
    {
        System.out.println(ex.getMessage());
    }
     return cnt;
}
```

Program 5.4: Program to display actress information.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
public class Main
{
    private final String url = "jdbc:postgresql://localhost/Temp";
    private final String user = "postgres";
    private final String password = "postgres";
    public Connection connect() throws SQLException
    {
        return DriverManager.getConnection(url, user, password);
    }
    public void getActresss()
    {
        String SQL = "SELECT Actress_id, fname, lname FROM Actress";
        try (Connection conn = connect();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(SQL))
        {
            // display Actress information
            displayActress(rs);
        } catch (SQLException ex)
        {
            System.out.println(ex.getMessage());
        }
    }
    public int getActressCount()
    {
        String SQL = "SELECT count(*) FROM Actress";
        int cnt = 0;
        try (Connection conn = connect();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(SQL))
        {
            rs.next();
            cnt = rs.getInt(1);
        }
        catch (SQLException ex)
        {
            System.out.println(ex.getMessage());
        }
        return cnt;
    }
```

```
    private void displayActress(ResultSet rs) throws SQLException
    {
        while (rs.next())
        {
            System.out.println("Actress :"+rs.getString("Actress_id") + "\t"
            + rs.getString("fname") + "\t" + rs.getString("last_name"));
        }
    }
    public void findActressByID(int ActressID)
    {
        String SQL = "SELECT Actress_id,fname,lname" + "FROM Actress" +
        "WHERE Actress_id =?";
        try (Connection conn = connect();
        PreparedStatement pstmt=conn.prepareStatement(SQL))
        {
            pstmt.setInt(1, ActressID);
            ResultSet rs = pstmt.executeQuery();
            displayActress(rs);
        } catch (SQLException ex)
        {
            System.out.println(ex.getMessage());
        }
    }
    public static void main(String[] args)
    {
        Main main = new Main();
        main.findActressByID(200);
    }
}
```

Output:

```
Actress : 200 Katrina Kaif
```

5.5 SCROLLABLE AND UPDATABLE RESULT SETS

- The ResultSet interface provides methods for retrieving and manipulating the results of executed queries. These objects can have different functionality and characteristics.
- These characteristics are type, concurrency, and cursor holdability. The sensitivity of a ResultSet object is determined by one of three different ResultSet types.

5.5.1 ResultSet [W-18]

- JDBC provides the following methods from Connection object to create statements with certain types of ResultSet.
  - `createStatement(int resultSetType, int resultSetConcurrency);`
  - `prepareStatement(String SQL, int resultSetType, int resultSetConcurrency);`
  - `prepareCall(String sql, int resultSetType, int resultSetConcurrency);`
- The following code creates a Statement object to create a FORWARD_ONLY, READ_ONLY ResultSet object:

```
Statement stmt = conn.createStatement
{
   ResultSet.TYPE_FORWARD_ONLY,
   ResultSet.CONCUR_READ_ONLY;
}
ResultSet Navigation
```

- We can use the following methods from ResultSet interface to move the cursor.

Table 5.3: Methods from ResultSet interface to move the cursor

Methods	Description
`beforeFirst()`	Moves the cursor to before the first row.
`afterLast()`	Moves the cursor to after the last row.
`first()`	Moves the cursor to the first row.
`last()`	Moves the cursor to the last row.
`absolute(int row)`	Moves the cursor to the specified row.
`relative(int row)`	Moves the cursor number of rows forward or backwards relative to where it is.
`previous()`	Moves the cursor to the previous row.
`next()`	Moves the cursor to the next row.
`int getRow()`	Returns the row number that the cursor is pointing to.
`moveToInsertRow()`	Moves the cursor to where we can insert a new row into the database. The current row number is not changed.
`moveToCurrentRow()`	Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing.

ResultSet Column Data:

- We have two ways to get data in the ResultSet.
  1. By Column Index.
  2. By Column Name.

- Methods get int value from a column:
  `public int getInt(String columnName)`
  `public int getInt(int columnIndex)`
  The column index starts at 1.

Update ResultSet:

- We can update current row in ResultSet object.
- We need to indicate the column name or index during the update.
- To update a String column of the current row following methods are used.
  `public void updateString(int columnIndex, String s) throws SQLException`
  `public void updateString(String columnName, String s) throws SQLException`
- To push the update changes to the database, invoke one of the following methods:

Table 5.4: Methods of Push and Update changes of Database

Methods	Description
`updateRow()`	Updates the corresponding row in the database.
`deleteRow()`	Deletes the current row from the database.
`refreshRow()`	Refreshes the result set to reflect any changes in the database.
`cancelRowUpdates()`	Cancels any updates made on the current row.
`insertRow()`	Inserts a row into the database when the cursor is pointing to the insert row.

Program 5.5: Program Scrollable and updatable result sets.

```
import java.sql.*;
class FetchRecord
{
   public static void main(String args[])throws Exception
   {
      Class.forName("org.postgresql.Driver");
      con = DriverManager.getConnection
         ("jdbc:postgresql://localhost:5432/testdb","kamil", "kamil");
      Statement stmt=con.createStatement
          (ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
      ResultSet rs=stmt.executeQuery("select * from emp765");
      //getting the record of 3rd row
      rs.absolute(3);
      System.out.println(rs.getString(1)+"
                                 "+rs.getString(2)+" "+rs.getString(3));
      con.close();
   }
}
```

Output:

```
Kamil   Khan  2000
```

5.5.2 Types of ResultSets [W-18]

- Various types of ResultSet is as follows:
  1. **TYPE_FORWARD_ONLY:** The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved. The default ResultSet type is TYPE_FORWARD_ONLY.
  2. **TYPE_SCROLL_INSENSITIVE:** The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
  3. **TYPE_SCROLL_SENSITIVE:** The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

ResultSet Concurrency:

- The concurrency of a ResultSet object determines what level of update functionality is supported.
- There are two concurrency levels:
  1. **CONCUR_READ_ONLY:** The ResultSet object cannot be updated using the ResultSet interface. The default ResultSet concurrency is CONCUR_READ_ONLY.
  2. **CONCUR_UPDATABLE:** The ResultSet object can be updated using the ResultSet interface.

5.6 METADATA - DATABASEMETADATA, RESULTSETMETADATA

- Metadata means data about data. We can get further information from the data. If you have to get metadata of a table like total number of columns, column name, column type etc.

Java DatabaseMetaData interface:

- DatabaseMetaData interface provides methods to get metadata 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.
- **Commonly used methods of DatabaseMetaData interface:**
  1. **public String getDriverName()throws SQLException:** It returns the name of the JDBC driver.
  2. **public String getDriverVersion()throws SQLException:** It returns the version number of the JDBC driver.
  3. **public String getUserName()throws SQLException:** It returns the username of the database.

4. **public String getDatabaseProductName()throws SQLException:** It returns the product name of the database.
5. **public String getDatabaseProductVersion()throws SQLException:** It returns the product version of the database.
6. **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.

Program 5.6: Program for DatabaseMetaData.

```
import java.sql.*;
class Dbmd
{
   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");
         DatabaseMetaData dbmd=con.getMetaData();
         System.out.println("Driver Name: "+dbmd.getDriverName());
         System.out.println("Driver Version: "+dbmd.getDriverVersion());
         System.out.println("UserName: "+dbmd.getUserName());
         System.out.println("Database Product Name:
                                        "+dbmd.getDatabaseProductName());
         System.out.println("Database Product Version:
                                      "+dbmd.getDatabaseProductVersion());
         con.close();
      }catch(Exception e){ System.out.println(e);}
   }
}
```

Output:

```
Driver Name: Oracle JDBC Driver
Driver Version: 10.2.0.1.0XE
Database Product Name: Oracle
Database Product Version: Oracle Database 10g Express Edition
                          Release 10.2.0.1.0 -Production
```

Java 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.

Table 5.5: Commonly used methods of ResultSetMetaData interface

Method	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.

- If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.
- That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions:
  1. To increase performance.
  2. To maintain the integrity of business processes.
  3. To use distributed transactions.
- Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.
- To enable manual- transaction support instead of the auto-commit mode that the JDBC driver uses by default, use the Connection object's setAutoCommit() method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.
- For example, if you have a Connection object named conn, code the following to turn off auto-commit:

```
conn.setAutoCommit(false);
```

Commit and Rollback:

- Once, you are done with your changes and want to commit the changes then call commit() method on connection object as follows:

```
conn.commit( );
```

- Otherwise, to roll back updates to the database made using the Connection named conn, use the following code:

```
conn.rollback( );
```

- The following example illustrates the use of a Commit and Rollback object:

```
try
{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
   String SQL = "INSERT INTO Employees" + "VALUES (106, 20, 'Siya',
  'Patil')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERT IN Employees" + "VALUES (107, 22, 'Shrimit',
   'Ingale')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se)
{
   // If there is any error.
   conn.rollback();
}
```

- In this case, none of the above INSERT statement would success and everything would be rolled back.
- This interface is useful because it provides methods to get metadata from the ResultSet object. The methods are as follows:

```
1. public int getColumnCount() throws SQLException
2. public String getColumnName(int index) throws SQLException
3. public String getColumnTypeName(int index) throws SQLException
4. public String getTableName(int index) throws SQLException
5. How to get object of ResultSetMetaData
6. getMetaData() method of ResultSet interface returns the object of
   ResultSetMetaData
7. public ResultSetMetaData getMetaData() throws SQLException
```

Program 5.7: Program for ResultsetMetaData.

```
import java.sql.*;
class Rsmd
{
   public static void main(String args[])
   {
      try
      {
         Class.forName("org.postgresql.Driver");
         con = DriverManager.getConnection
                              ("jdbc:postgresql://localhost:5432/testdb",
              "kamil", "kamil");//user name password as per your system set
         PreparedStatement ps=con.prepareStatement("select * from emp");
         ResultSet rs=ps.executeQuery();
         ResultSetMetaData rsmd=rs.getMetaData();
         System.out.println("Total columns: "+rsmd.getColumnCount());
         System.out.println("Column Name of 1st column:
                                                 "+rsmd.getColumnName(1));
         System.out.println("Column Type Name of 1st column:
                                              "+rsmd.getColumnTypeName(1));
         con.close();
         }catch(Exception e)
         {
            System.out.println(e);
         }
      }
   }
```

Output:

```
Total columns: 2
Column Name of 1st column: ID
Column Type Name of 1st column: NUMBER
```

Program 5.8: JDBC program connects to the database, creates a Student table into the database and closes the connection.

```
import java.sql.*;
public class sample1
{
   public static void main(String[] args)
   {
      Connection conn = null;
      Statement stmt = null;
```

```
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            conn= DriverManager.getConnection("jdbc:odbc:testdsn");
            System.out.println("Connected to database");
            stmt = conn.createStatement();
            String sql = "CREATE TABLE student " +
                      "(id INTEGER not NULL, " +
                      " first VARCHAR(255), " +
                      " last VARCHAR(255), " +
                      " age INTEGER, " +
                      " PRIMARY KEY ( id ))";
            stmt.executeUpdate(sql);
            System.out.println("table created");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            try
            {
                if(stmt!=null)
                conn.close();
            }
            catch(SQLException se)
            {
                se.printStackTrace();
            }
            try
            {
                if(conn!=null)
                conn.close();
            }
            catch(SQLException se)
            {
                se.printStackTrace();
            }
        }
    }
}
```

Output:

```
A table student will be created in database on console:
Connected to database table created
```

Program 5.9: Program connects to the database and updates student record from student table.

```
import java.sql.*;
public class sample3
{
    public static void main(String[] args)
    {
        Connection conn = null;
        Statement stmt = null;
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            conn = DriverManager.getConnection("jdbc:odbc:testdsn");
            System.out.println("Connected to database");
            stmt = conn.createStatement();
            String sql = "update student set first=? where id=? ";
            PreparedStatement pr=conn.prepareStatement(sql);
            pr.setString(1,"sakshi SANTOSH");
            pr.setInt(2,1);
            pr.executeUpdate();
            pr.setString(1,"sakshi pendase");
            pr.setInt(2,2);
            pr.executeUpdate();
            System.out.println("table updated");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            try
            {
                if(stmt!=null)
                conn.close();
            }
```

```
            catch(SQLException se)
            {
               se.printStackTrace();
            }
            try
            {
               if(conn!=null)
               conn.close();
            }
            catch(SQLException se)
            {
               se.printStackTrace();
            }
         }
      }
   }
```

JDBC sample code to connect a PostgreSQL Database:

```
Class.forName("org.postgresql.Driver");
Connection connection = null;
connection = DriverManager.getConnection("jdbc:postgresql://
                              hostname:port/dbname","username", "password");
connection.close();
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.SQLException;
public class JDBCExample
{
   public static void main(String[] argv)
   {
      System.out.println("-------- PostgreSQL" + "JDBC Connection Testing
      ------------");
      try
      {
         Class.forName("org.postgresql.Driver");
      }
```

```
        catch (ClassNotFoundException e)
        {
            System.out.println("Where is your PostgreSQL JDBC Driver? "
                                        + "Include in your library path!");
            e.printStackTrace();
            return;
        }
        System.out.println("PostgreSQL JDBC Driver Registered!");
        Connection connection = null;
        try
        {
            connection = DriverManager.getConnection("jdbc:postgresql:
                                //127.0.0.1:5432/TempDb", "mkyong", "123456");
        }
        catch (SQLException e)
        {
            System.out.println("Connection Failed! Check output console");
            e.printStackTrace();
            return;
        }
        if (connection != null)
        {
            System.out.println("Connect your database");
        }
        else
        {
            System.out.println("Failed to make connection!");
        }
    }
}
```

Connecting to Database:

- The following Java code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```
import java.sql.Connection;
import java.sql.DriverManager;
public class PostgreSQLJDBC {
   public static void main(String args[]) {
      Connection c = null;
```

```
        try {
            Class.forName("org.postgresql.Driver");
            c = DriverManager
            tConnection("jdbc:postgresql://localhost:5432/TempDb",
                                                        "postgres", "123");
        } catch (Exception e) {
           e.printStackTrace();
           System.err.println(e.getClass().getName()+": "+e.getMessage());
           System.exit(0);
        }
        System.out.println("Opened database successfully");
    }
}
```

Create a Table:

- Java program used to create a table in previously opened database.

```
import java.sql.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class PostgreSQLJDBC
{
   public static void main( String args[] )
   {
      Connection c = null;
      Statement stmt = null;
      try
      {
         Class.forName("org.postgresql.Driver");
         c =DriverManager.getConnection("jdbc:postgresql:
                //localhost:5432/TempDb","Demo", "123");
         System.out.println("Opened database successfully");
         stmt = c.createStatement();
         String sql = "CREATE TABLE EMP " +
            "(ID INT PRIMARY KEY     NOT NULL," +
            " NAME           TEXT    NOT NULL, " +
            " AGE            INT     NOT NULL, " +
            " ADDRESS        CHAR(50), " +
            " SALARY         REAL)";
```

```
            stmt.executeUpdate(sql);
            stmt.close();
            c.close();
         } catch ( Exception e ) {
            System.err.println( e.getClass().getName()+": "+ e.getMessage() );
            System.exit(0);
         }
         System.out.println("Successfully created table …");
      }
   }
```

- Java code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned.

```
import java.sql.Connection;
import java.sql.DriverManager;
public class PostgreSQLJDBC {
   public static void main(String args[]) {
      Connection c = null;
      try {
         Class.forName("org.postgresql.Driver");
          c = DriverManager.getConnection("jdbc:postgresql:
          //localhost:5432/TempDb","postgres", "123");
      } catch (Exception e) {
         e.printStackTrace();
         System.err.println(e.getClass().getName()+ ": "+e.getMessage());
         System.exit(0);
      }
      System.out.println("Db.Opened successfully");
   }
}
```

- Java program will be used to create a table in previously opened database. Make sure you do not have this table already in your target database.

```
import java.sql.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
```

```
public class PostgreSQLJDBC {
   public static void main( String args[] ) {
      Connection c = null;
      Statement stmt = null;
      try {
         Class.forName("org.postgresql.Driver");
         c = DriverManager .getConnection("jdbc:postgresql:
                                  //localhost:5432/TempDb", "Demo", "123");
         System.out.println("Db.Successfully Opened ");
         stmt = c.createStatement();
         String sql = "CREATE TABLE COMPANY " +
            "(ID INT PRIMARY KEY     NOT NULL," +
            " NAME           TEXT    NOT NULL, " +
            " AGE            INT     NOT NULL, " +
            " ADDRESS        CHAR(50), " +
            " SALARY         REAL)";
         stmt.executeUpdate(sql);
         stmt.close();
         c.close();
      } catch ( Exception e ) {
         System.err.println( e.getClass().getName()+": "+ e.getMessage() );
         System.exit(0);
      }
      System.out.println("Created Table successfully");
   }
}
```

- When a program is compiled and executed, it will create the COMPANY table in **TempDb** database and will display the following two lines:

```
Opened database successfully
Table created successfully
```

INSERT Operation: **[W-18, S-19]**

- Java program shows how we can create records in our COMPANY table

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class PostgreSQLJDBC {
   public static void main(String args[]) {
      Connection c = null;
      Statement stmt = null;
```

```
        try {
            Class.forName("org.postgresql.Driver");
            c =    DriverManager.getConnection
                    ("jdbc:postgresql://localhost:5432/TempDb","Demo", "123");
           c.setAutoCommit(false);
           System.out.println("Db. Opened successfully");
           stmt = c.createStatement();
           String sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "
                      + "VALUES (1, 'Shammi', 21, 'California', 20000.00 );";
           stmt.executeUpdate(sql);
            sql = "INSERT INTO COMPANY(ID,NAME,AGE,ADDRESS,SALARY) " + "VALUES
            (2, 'Yash', 23, 'Texas', 15000.00 );";
           stmt.executeUpdate(sql);
           sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "
                            + "VALUES (3, 'Abhi', 19, 'Norway', 20000.00 );";
           stmt.executeUpdate(sql);
           sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "
                        + "VALUES (4, 'Shashi', 25, 'NewJercy', 65000.00 );";
           stmt.executeUpdate(sql);
           stmt.close();
           c.commit();
           c.close();
        } catch (Exception e) {
           System.err.println( e.getClass().getName()+": "+ e.getMessage() );
           System.exit(0);
        }
        System.out.println("Created Records successfully");
    }
}
```

- When the above program is compiled and executed, it will create given records in COMPANY table and will display the following two lines:

```
Opened database successfully
Records created successfully
```

SELECT Operation:

- The following Java program shows how we can fetch and display records from our COMPANY table created in above example:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class PostgreSQLJDBC {
   public static void main( String args[] ) {
      Connection c = null;
      Statement stmt = null;
      try {
         Class.forName("org.postgresql.Driver");
         c = DriverManager
            .getConnection
                ("jdbc:postgresql://localhost:5432/TempDb", "Demo", "123");
         c.setAutoCommit(false);
         System.out.println("Opened database successfully");
         stmt = c.createStatement();
         ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
         while ( rs.next() ) {
            int id = rs.getInt("id");
            String  name = rs.getString("name");
            int age  = rs.getInt("age");
            String  address = rs.getString("address");
            float salary = rs.getFloat("salary");
            System.out.println( "ID = " + id );
            System.out.println( "NAME = " + name );
            System.out.println( "AGE = " + age );
            System.out.println( "ADDRESS = " + address );
            System.out.println( "SALARY = " + salary );
            System.out.println();
         }
         rs.close();
         stmt.close();
         c.close();
      }  catch ( Exception e ) {
```

```
            System.err.println( e.getClass().getName()+": "+ e.getMessage() );
            System.exit(0);
        }
        System.out.println("Operation done successfully….");
    }
}
```

- When the program is compiled and executed, it will produce the following result:

```
Opened database successfully
ID = 1
NAME = Shammi
AGE = 21
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Yash
AGE = 23
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Abhi
AGE = 19
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Shashi
AGE = 25
ADDRESS = New Jercy
SALARY = 65000.0

Operation done successfully ….
```

UPDATE Operation: **[S-19]**

- The following Java code shows how we can use the UPDATE statement to update any record and then fetch and display updated records from our COMPANY table:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
```

```
public class PostgreSQLJDBC {
   public static void main( String args[] ) {
      Connection c = null;
      Statement stmt = null;
      try {
         Class.forName("org.postgresql.Driver");
         c = DriverManager
            .getConnection("jdbc:postgresql://localhost:5432/TempDb",
            "Demo", "123");
         c.setAutoCommit(false);
         System.out.println("Opened database successfully");
         stmt = c.createStatement();
         String sql = "UPDATE COMPANY set SALARY = 25000.00 where ID=1;";
         stmt.executeUpdate(sql);
         c.commit();
         ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
         while ( rs.next() ) {
            int id = rs.getInt("id");
            String  name = rs.getString("name");
            int age  = rs.getInt("age");
            String  address = rs.getString("address");
            float salary = rs.getFloat("salary");
            System.out.println( "ID = " + id );
            System.out.println( "NAME = " + name );
            System.out.println( "AGE = " + age );
            System.out.println( "ADDRESS = " + address );
            System.out.println( "SALARY = " + salary );
            System.out.println();
         }
         rs.close();
         stmt.close();
         c.close();
      } catch ( Exception e ) {
         System.err.println( e.getClass().getName()+": "+ e.getMessage() );
         System.exit(0);
      }
      System.out.println("Operation done successfully");
   }
}
```

- When the program is compiled and executed, it will produce the following result:

```
Opened database successfully
ID = 1
NAME = Kajol
AGE = 23
ADDRESS = Florida
SALARY = 40000.0
ID = 2
NAME = Yash
AGE = 23
ADDRESS = Texas
SALARY = 15000.0
ID = 3
NAME = Abhi
AGE = 19
ADDRESS = Norway
SALARY = 20000.0
ID = 4
NAME = Shashi
AGE = 25
ADDRESS = New Jercy
SALARY = 65000.0
Operation done successfully ….
```

DELETE Operation: **[S-19]**

- The following Java code shows how we can use the DELETE statement to delete any record and then fetch and display remaining records from our COMPANY table:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class PostgreSQLJDBC6 {
   public static void main( String args[] ) {
      Connection c = null;
      Statement stmt = null;
      try {
         Class.forName("org.postgresql.Driver");
         c = DriverManager.getConnection
                 ("jdbc:postgresql://localhost:5432/TempDb","Demo", "123");
         c.setAutoCommit(false);
```

```
            System.out.println("Opened database successfully");
            stmt = c.createStatement();
            String sql = "DELETE from COMPANY where ID = 2;";
            stmt.executeUpdate(sql);
            c.commit();
            ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANY;" );
            while ( rs.next() ) {
               int id = rs.getInt("id");
               String  name = rs.getString("name");
               int age  = rs.getInt("age");
               String  address = rs.getString("address");
               float salary = rs.getFloat("salary");
               System.out.println( "ID = " + id );
               System.out.println( "NAME = " + name );
               System.out.println( "AGE = " + age );
               System.out.println( "ADDRESS = " + address );
               System.out.println( "SALARY = " + salary );
               System.out.println();
            }
            rs.close();
            stmt.close();
            c.close();
         } catch ( Exception e ) {
            System.err.println( e.getClass().getName()+": "+ e.getMessage() );
            System.exit(0);
         }
         System.out.println("Operation done successfully");
      }
   }
```

- When the program is compiled and executed, it will produce the following result:

```
Opened database successfully
ID = 2
NAME = Yash
AGE = 23
ADDRESS = Texas
SALARY = 15000.0
ID = 3
```

```
NAME = Abhi
AGE = 19
ADDRESS = Norway
SALARY = 20000.0
ID = 4
NAME = Shashi
AGE = 25
ADDRESS = New Jercy
SALARY = 65000.0
Operation done successfully …
```

Summary

- JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
- Java Database Connectivity (JDBC) is Java application programming interface that allows the Java programmers to access DBMS through Java code.
- JDBC API provides a standard interface for interacting with any Relational Database Management Systems (RDBMS). JDBC API consists of the JDBC Driver, Connection, Statement, ResultSet components.
- A JDBC driver is set of Java classes that implement JDBC interfaces for interacting with a specific database. Almost all database vendors such as MySQL, Oracle, Microsoft SQL Server, provide JDBC drivers
- In a Java application, you first load a JDBC driver and then establish a connection to the database. Through the Connection object, you can interact with the database e.g., creating a Statement to execute SQL queries against tables. You can open more than one connection to a database at a time.
- To execute a SQL query e.g., SELECT, INSERT, UPDATE, DELETE, etc., you use a Statement object. You create the Statement object through the Connection object. JDBC provides several types of statements for different purposes such as PreparedStatement, CallableStatement.
- After querying data from the database, you get a ResultSet object. The ResultSet object provides a set of API that allows you to traverse result of the query.
- The typical flow of using JDBC is as follows:
  1. First, load the JDBC driver and create a connection to the database.
  2. Then, create a Statement and execute the query to get a ResultSet.
  3. Next, traverse and process the ResultSet .
  4. Close the ResultSet, Statement and Connection.

- JDBC is the Java API for developing Java applications that access relational databases.
- In Order to connect to database JDBC uses JDBC driver. Since JDBC driver acts as a connector between JDBC and proprietary databases JDBC drivers are DB specific and generally provided by the DB vendor itself.
- The JDBC API is comprised of two packages:
  - java.sql - Referred to as the JDBC core API
  - javax.sql - Referred to as the JDBC Optional Package API
- Steps for connecting to DB are:
  - Loading driver
  - Creating connection to DB
  - Creating Statement
  - Executing Query
  - Processing ResultSet
  - Closing connection

Check Your Understanding

1. Which packages contain the JDBC classes?
   (a) java.jdbc and javax.jdbc (b) java.jdbc and java.jdbc.sql
   (c) java.sql and javax.sql (d) java.rdb and javax.rdb
2. Which type of driver provides JDBC access via one or more ODBC drivers?
   (a) Type 1 driver (b) Type 2 driver
   (c) Type 3 driver (d) Type 4 driver
3. 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)
4. Which of the following statements is false as far as different type of statements is concern in JDBC?
   (a) Regular Statement (b) Prepared Statement
   (c) Callable Statement (d) Interim Statement
5. The JDBC-ODBC bridge is_____.
   (a) Three tiered (b) Multithreaded
   (c) Best for any platform (d) All of the above
6. Which driver is called as thin-driver in JDBC?
   (a) Type-4 driver (b) Type-1 driver
   (c) Type-3 driver (d) Type-2 driver

7. Which method is used to perform DML statements in JDBC?
   (a) execute() (b) executeQuery()
   (c) executeUpdate() (d) executeResult()
8. The JDBC API is what allows access to a data source from a Java middle tier?
   (a) True (b) False
9. Which method is used for an SQL statement that is executed frequently?
   (a) prepareStatement (b) prepareCall
   (c) createStatement (d) None of the above
10. The JDBC-ODBC bridge is _____.
   (a) Multithreaded (b) Single threaded
   (c) Both of the above (d) None of the above
11. In the following JDBC drivers which is known as fully java driver?
   (a) Native-API driver (b) Network Protocol driver
   (c) Thin driver (d) Both (b) and (c)
12. Which of the following JDBC drivers is known as a partially java driver?
   (a) JDBC-ODBC bridge driver (b) Native-API driver
   (c) Network Protocol driver (d) Thin driver

Answers

1. (c)	2. (a)	3. (a)	4. (d)	5. (b)	6. (a)	7. (c)	8. (a)	9. (a)	10. (a)
11. (d)	12. (b)								

Practice Questions

Q.I Answer the following questions in short:

1. What is the use of SetAutoCommit()?
2. Write short note on: JDBC API Packages.
3. What is database?
4. What is meant by JDBC driver?
5. What is the use of JDBC API?
6. What is ResultsetMetaData?
7. What is DatabaseMetaData?

Q.II Answer the following questions:

1. Differentiate between statement and prepared statement interface.
2. Explain various type of JDBC drivers. Discuss advantages and disadvantages of each.
3. What is a difference between a connection and statement?
4. Explain how to create the URL.

5. Explain the significance of each:
   (i) ResultSet
   (ii) ResultSetMetaData
   (iii) DatabaseMetaData.
6. Which to use and when?
   (i) execute()
   (ii) executeUpdate()
   (iii) executeQuery()
   (iv) Statement
   (v) CallabaleStatement
   (vi) PreparedStatement.
7. By considering PreparedStatement interface, explain following methods of it:
   (i) setString()
   (ii) setInt()
8. On what record is the cursor initially positioned in the resultset?
9. Write a small program to open a connection to a database.
10. Describe JDBC-ODBC bridge in detail.
11. Explain architecture of JDBC in detail.
12. How to connect an Java application to PostgresSQL? Explain with example.

Q.III Define the Term:

1. Statement
2. ResultSet
3. Connection
4. DriverManager
5. Callable Statement
6. Prepared Statement

Previous Exam Questions

Winter 2018

1. Which type of statement is used to execute a call to a database store procedure? **[1 M]**
   (i) Prepared Statement
   (ii) Statement
   (iii) Callable Statement
   (iv) All kinds of Statements

Ans. (iii)

2. Create a student table with fields (roll number, name, percentage). Write a jdbc program for the following :
   (i) Insert student details.
   (ii) Display all details of student table. **[5 M]**

Ans. Refer to Section 5.6.

3. What is ResultSet? Explain the types of ResultSet with propersyntax. **[4 M]**

Ans. Refer to Section 5.5.1, 5.5.2.

Summer 2019

1. Create a student table with fields (roll_no, name, percentage). Write a JDBC program to insert, update and delete, display all student details. **[5 M]**

Ans. Refer to Section 5.6.

2. What is JDBC? Explain types of jdbc? **[4 M]**

Ans. Refer to Section 5.1, 5.2.1.

■■■

6... Servlets & JSP

Learning Objectives...

- To understand Basic Concepts of Servlet.
- To study life cycle of servlet.
- To learn about handling get and post request.
- To learn Session Tracking in Servlets.
- To understand basic concepts of JSP.
- To study life cycle of JSP.
- To learn scripting elements.
- To study JSP directives.

6.1 INTRODUCTION TO SERVLETS [W-18, S-19]

- Servlets are small programs that execute on the server side of a web connection. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server.
- In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers co-operate to provide content to a user. Consider a request for a static web page.
- A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content.
- The Multipurpose Internet Mail Extensions (MIME) is used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.
- Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices, availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

- In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information.
- It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response.
- Servlet technology 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.

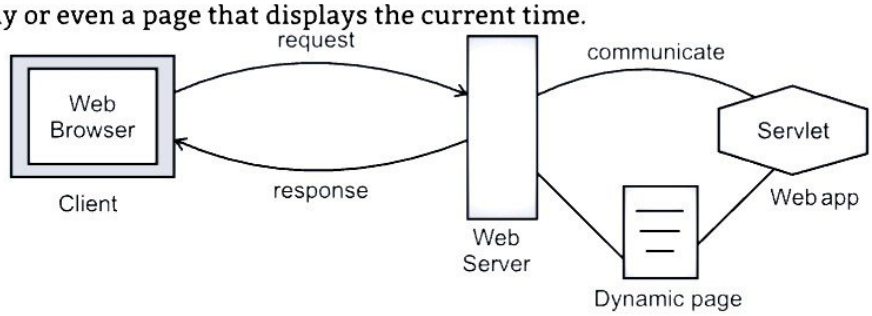


Fig. 6.1: Client Server Communication

- As Servlet Technology uses Java, web applications made using Servlet are secured, scalable and robust.

Advantages of using Servlets:

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.

6.1.1 Hierarchy of Servlets

- Servlet API consists of two important packages that encapsulate all the important classes and interface, namely javax.servlet and javax.servlet.http.

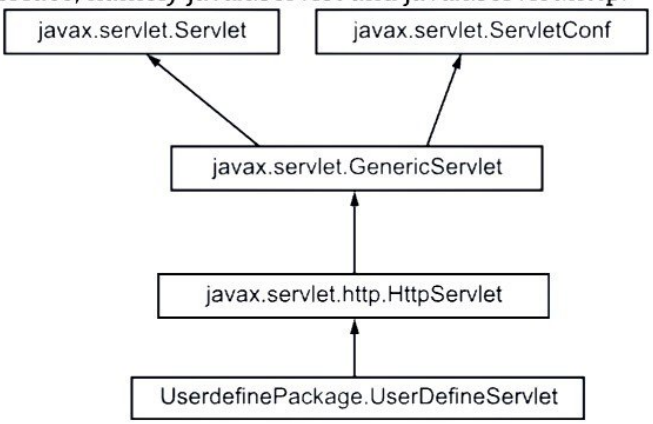


Fig. 6.2: Hierarchy of servlet

- The root interface of the servlet class is Servlet interface hierarchy. All Servlets need to either directly or indirectly implement the Servlet interface.
- The GenericServlet class of the Servlet API implements the Servlet interface. In addition to the Servlet interface, the GenericServlet class implements the ServletConfig interface of the Servlet API and the Serializable interface of the standard java.io package.
- The object of the ServletConfig interface is used by the Web container to pass the configuration information to a servlet when a servlet is initialized.
- To develop a servlet that communicates using HTTP, we need to extend the HttpServlet class in our servlet. The HttpServlet class extends the GenericServlet class and provides built-in HTTP functionality.
- **javax.servlet.Servlet** Interface contains following methods:
  1. `public void destroy()`
  2. `public ServletConfig getServletConfig()`
  3. `public String getServletInfo()`
  4. `public String getServletInfo()`
- **javax.servlet.ServletConfig** Interface contains following methods:
  1. `public String getInitParameter(String param)`
  2. `public Enumeration getInitParameterNames()`
  3. `public ServletContext getServletContext()`

6.1.2 Types of Servlet [S-19]

- There are two main servlet types, Generic and HTTP:

1. Generic Servlets:

- Extend javax.servlet.GenericServlet.
- Generic Servlet are protocol independent. They contain no inherent HTTP support or any other transport protocol.
- **GenericServlet Class:** GenericServlet is an abstract class that provides implementation of most of the basic servlet methods.
- **Methods of GenericServlet Class:**
  1. `public void init(ServletConfig)`
  2. `public abstract void service(ServletRequest request,ServletResposne response)`
  3. `public void destroy()`
  4. `public ServletConfig getServletConfig()`
  5. `public String getServletInfo()`
  6. `public ServletContext getServletContext()`
  7. `public String getInitParameter(String name)`

```
8. public Enumeration getInitParameterNames()
9. public String getServletName()
10. public void log(String msg)
11. public void log(String msg, Throwable t)
```

HttpServlet Class:

- HttpServlet is also an abstract class. This class gives implementation of various service() methods of Servlet interface.
- To create a servlet, we should create a class that extends HttpServlet abstract class. The Servlet class that we will create, must not override service() method. Our servlet class will override only the doGet() and/or doPost() methods.
- The service() method of HttpServlet class listens to the Http methods (GET, POST etc) from request stream and invokes doGet() or doPost() methods based on Http Method type.

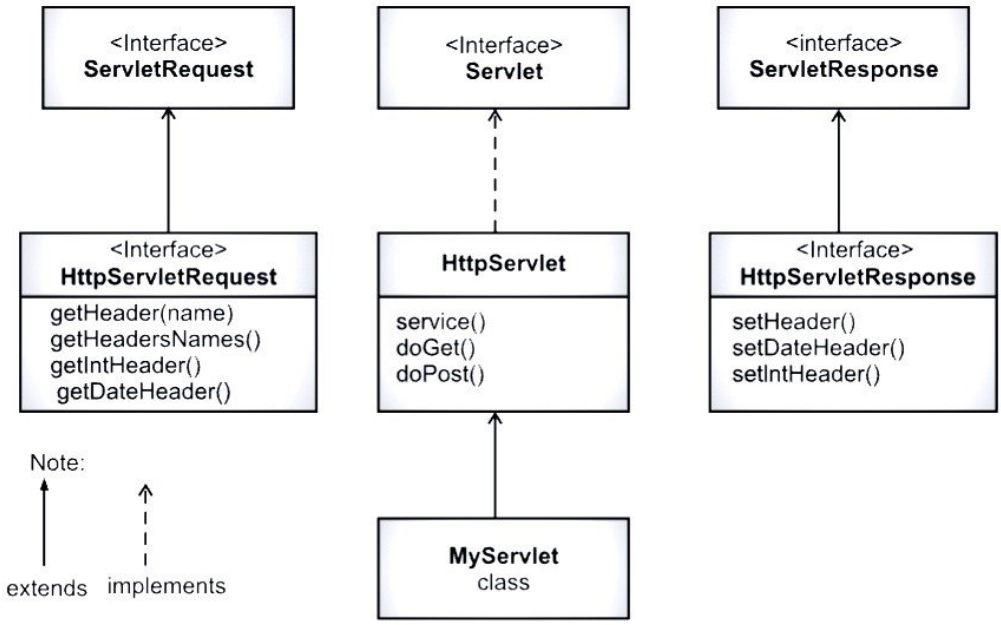


Fig. 6.3: HTTP Servlet Request & Response

2. HTTP Servlets:

- It extends javax.servlet.HttpServlet.
- They have built-in HTTP protocol support and are more useful in a Sun Java System Web Server environment.
- For both servlet types, you implement the constructor method init() and the destructor method destroy() to initialize or deallocate resources.
- All servlets must implement a service() method, which is responsible for handling servlet requests.
- For generic servlets, simply override the service method to provide routines for handling requests.

- HTTP servlets provide a service method that automatically routes the request to another method in the servlet based on which HTTP transfer method is used.
- HTTP servlets, override doPost() to process POST requests, doGet() to process GET requests.
- GenericServlet is protocol independent implementation of Servlet interface whereas HttpServlet is HTTP protocol specific implementation.
- Most of the times we use servlet for creating web application and that's why we extend HttpServlet class.
- HttpServlet class extends GenericServlet and also provide some other methods specific to HTTP protocol.

6.2 LIFE CYCLE OF A SERVLET [W-18]

- It can be defined as, the entire process from its creation till the destruction. The following are the paths followed by a servlet.
  - The servlet is initialized by calling the **init()** method.
  - The servlet calls **service()** method to process a client's request.
  - The servlet is terminated by calling the **destroy()** method.
  - Finally, servlet is garbage collected by the garbage collector of the JVM.

1. **init() Method:**
   - The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets.
   - The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.
   - When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.
   - The init method definition looks like this:

```
public void init() throws ServletException
{
 // Initialization code...
}
```

2. **service() Method:**
   - The service() method is the main method to perform the actual task. The servlet container.
     (i.e. web server) calls the service() method to handle requests coming from the client (browsers) and to write the formatted response back to the client.

- Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.
- Here, is the **syntax** of this method:

```
public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException
{
}
```

- The service () method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate.

(i) doGet() Method:

- A GET request results from a normal request for a URL or from an HTML form that has no method specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
   // Servlet code...
}
```

(ii) doPost() Method:

- A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
   // Servlet code...
}
```

3. destroy() Method:

- The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
- After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
public void destroy()
{
   // Finalization code...
}
```

- The Fig. 6.4 depicts a typical servlet life cycle scenario.
  - First the HTTP requests coming to the server are delegated to the servlet container.
  - The servlet container loads the servlet before invoking the service() method.
  - Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.

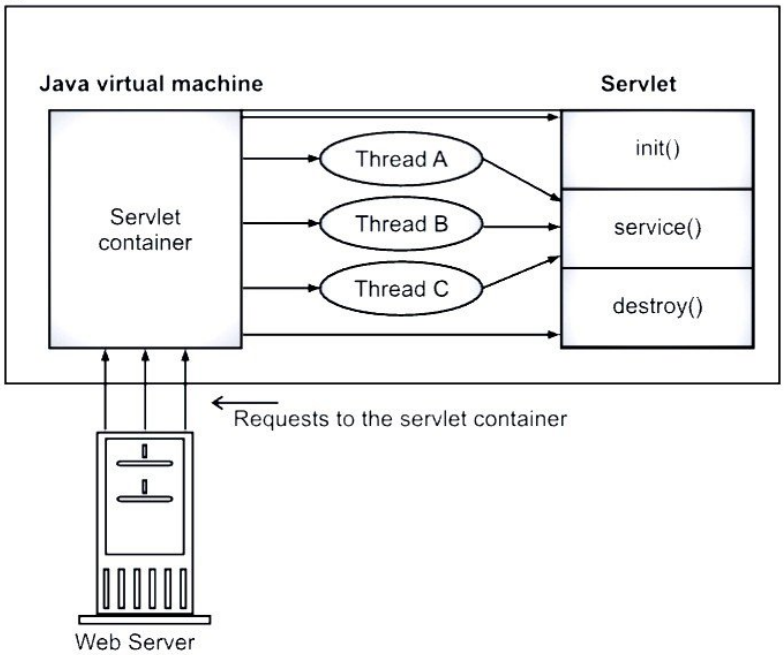


Fig. 6.4: Life cycle of Servlet

- Service method dispatches the requests such as DELETE, GET, HEAD, OPTIONS, POST, PUT, and TRACE.
- A GET request is dispatched to the doGet(HttpServletRequest request, HttpServletResponse response) method.
- A POST request is dispatched to the doPost(HttpServletRequest request, HttpServletResponse response) method. These are the two methods you will usually override.
- doGet and doPost typically do the same thing, so usually you do the real work in one, and have the other just call it.

```
public void doGet(HttpServletRequest request, HttpServletResponse
response)
{
   doPost(request, response);
}
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
```

- This method services a GET request.

Table 6.1: Classes and Interfaces of javax.servlet

Interfaces	Classes
Servlet	ServletInputStream
ServletContext	ServletOutputStream
ServletConfig	ServletRequestWrapper
ServletRequest	ServletResponseWrapper
ServletResponse	ServletRequestEvent
ServletContextListener	ServletContextEvent
RequestDispatcher	ServletRequestAttributeEvent
SingleThreadModel	ServletContextAttributeEvent
Filter	ServletException
FilterConfig	UnavailableException
FilterChain	GenericServlet

Table 6.2: Classes and Interface of javax.servlet.http

Classes	Interfaces
HttpServlet	HttpServletRequest
HttpServletResponse	HttpSessionAttributeListener
HttpSession	HttpSessionListener
Cookie	HttpSessionEvent

A Simple Servlet:

- To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:
  1. Create and compile the servlet source code. Then, copy the servlet's class file to the proper directory, and add the servlet's name and mappings to the proper **web.xml** file.
  2. Start Tomcat.
  3. Start a web browser and request the servlet.
- Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code:

- To begin, create a file named **HelloServlet.java** that contains the following program:

```
import java.io.*;
import javax.servlet.*;
public class HelloServlet extends GenericServlet
```

```
{
    public    void    service(ServletRequest    request,      ServletResponse
    response)throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>Hello!");
    pw.close();
    }
}
```

6.3 TOMCAT CONFIGURATION

Step 1 : Download Apache Tomcat from the link
https://tomcat.apache.org/download-80.cgi

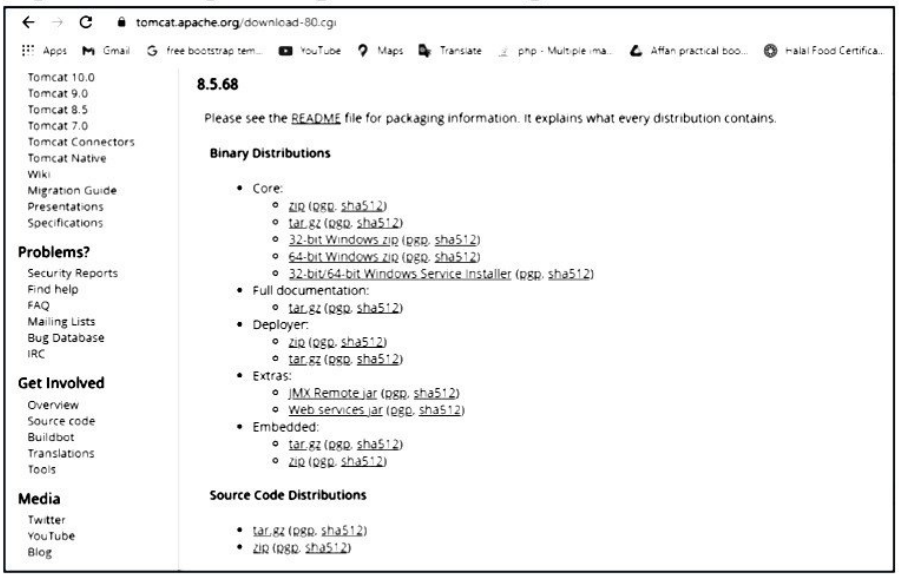


Fig. 6.5

Step 2 : Download and Setup JAVA variables globally.

- Setup JAVA_HOME variable path as C:\Program Files\Java\jdk1.8.0_131 path may change in your computer.
- Setup JRE_HOME variable path as C:\Program Files\Java\jre1.8.0_131 path may change in your computer.
- Setup CATALINA_HOME variable path as C:\Tomcat8 which is my Tomcat installation path.

Step 3 : After successful installation, go to BIN folder directly under Tomcat folder. You will find two batch files with names **startup.bat** and **shutdown.bat**.

C:\Tomcat8\bin>startup.bat

C:\Tomcat8\bin>shutdown.bat

```
C:\WINDOWS\system32\cmd.exe

C:\Tomcat8\bin>startup.bat
Using CATALINA_BASE:   "C:/Tomcat8"
Using CATALINA_HOME:   "C:/Tomcat8"
Using CATALINA_TMPDIR: "C:/Tomcat8\temp"
Using JRE_HOME:        "C:\Program Files\Java\jdk1.8.0_131"
Using CLASSPATH:       "C:/Tomcat8\bin\bootstrap.jar;C:/Tomcat8\bin\tomcat-juli.jar"
C:\Tomcat8\bin>shutdown.bat
Using CATALINA_BASE:   "C:/Tomcat8"
Using CATALINA_HOME:   "C:/Tomcat8"
Using CATALINA_TMPDIR: "C:/Tomcat8\temp"
Using JRE_HOME:        "C:\Program Files\Java\jdk1.8.0_131"
Using CLASSPATH:       "C:/Tomcat8\bin\bootstrap.jar;C:/Tomcat8\bin\tomcat-juli.jar"
C:\Tomcat8\bin>_
```

Fig. 6.6

Step 4 : Go to 'http://localhost:8080'. You should see the below page. It means you successfully installed Tomcat Server. You should start the server before opening the webpage.

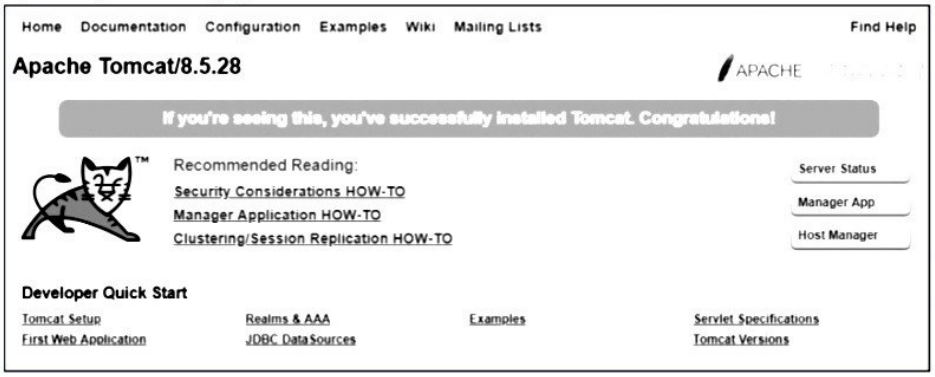


Fig. 6.7

Step 5 : Go to Tomcat/conf/tomcat-users.xml

Add Usernames, Passwords and Roles required. Without configuring these Usernames, You will be able to access Server Status, Manager App and Host Manager options on the Tomcat Home Page.

```
<role rolename="manager-gui"/>
<user username="admin" password="admin" roles="manager-gui"/>
<role rolename="admin-gui"/>
<user username="tomcat" password="tomcat" roles="admin-gui"/>
```

6.4 HANDLING GET AND POST REQUEST (HTTP)

- Servlet API provides two important interfaces javax.servlet.ServletRequest and javax.servlet.http.HttpServletRequest to encapsulate client request. Implementations of these interfaces provide important information about client request to a servlet.

1. Handling HTTPRequest:

- When a browser requests for a web page, it sends lot of information to the web server which cannot be read directly because this information travel as a part of header of HTTP request.
- The following Methods will be useful in all like doGet, doPost service.

Table 6.3: Methods of ServletRequest

Methods	Description
`Object getAttribute(String name)`	This method returns attribute set on request object by name.
`Enumeration getAttributeName()`	This method returns an Enumeration containing the names of the attributes available in this request.
`int getContentLength()`	This method returns size of request body.
`int getContentType()`	This method returns media type of request content.
`ServletInputStream getInputStream()`	This method returns an input stream for reading binary data.
`String getParameter(String name)`	This method returns value of parameter by name.
`String getLocalAddr()`	This method returns the Internet Protocol(IP) address of the interface on which the request was received.
`Enumeration getParameterNames()`	This method returns an enumeration of all parameter names.
`String[] getParameterValues(String name)`	This method returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.
`ServletContext getServletContext()`	This method returns the servlet context of current request.

contd. ...

`String getServerName()`	This method returns the host name of the server to which the request was sent.
`int getServerPort()`	This method returns the port number to which the request was sent.
`boolean isSecure()`	This method returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.
`void removeAttribute(String name)`	This method removes an attribute from this request.
`void setAttribute(String name, Object o)`	This method stores an attribute in this request.

HttpServletRequest Interface:

- HttpServletRequest interface adds the methods that relates to the HTTP protocol.
- The HttpServletRequest interface enables a servlet to obtain information about a client request.

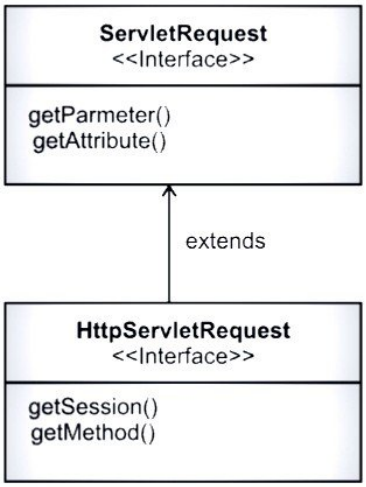


Fig. 6.8: HttpServletRequest Interface

Table 6.4: Methods of HttpServletRequest

Methods	Description
`String getContextPath()`	This method returns the portion of the request URL that indicates the context of the request.
`Cookies getCookies()`	This method returns an array containing all of the Cookie objects the client sent with this request.

contd. ...

`String getQueryString()`	This method returns the query string that is contained in the request URL after the path.
`HttpSession getSession()`	This method returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.
`String getMethod()`	This method returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.
`Part getPart(String name)`	This method gets the Part with the given name.
`String getPathInfo()`	This method returns any extra path information associated with the URL the client sent when it made this request.
`String getServletPath()`	This method returns the part of this request's URL that calls the servlet.

Simple Servlet Method:

```
import java.io.*;
import javax.servlet.*;
public class SimpleGenericServlet extends GenericServlet
{
   public void service (ServletRequest request, ServletResponse response)
   throws ServletException, IOException
   {  response.setContentType("text/plain");
      PrintWriter out = response.getWriter();
      out.println("Hello India"); out.close();
   }
}
```

2. Handling HTTPResponse:

- Servlet API provides two important interfaces ServletResponse and HttpServletResponse to assist in sending response to client.
- The HttpServletResponse interface enables a servlet to formulate an HTTP response to a client.

Table 6.5: Methods of ServletResponse

Methods	Description
`PrintWriter getWriter()`	This method returns a PrintWriter object that can send character text to the client.
`void setBufferSize(int size)`	This method sets the preferred buffer size for the body of the response.
`void setContentLength(int len)`	This method sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.

contd. ...

`void setContentType(String type)`	This method sets the content type of the response being sent to the client before sending the respond.
`void setBufferSize(int size)`	This method sets the preferred buffer size for the body of the response.
`boolean isCommitted()`	This method returns a boolean indicating if the response has been committed.
`void setLocale(Locale loc)`	This method sets the locale of the response, if the response has not been committed yet.

HttpServletResponse Interface:

- HttpServletResponse interface adds the methods that relates to the HTTP response.

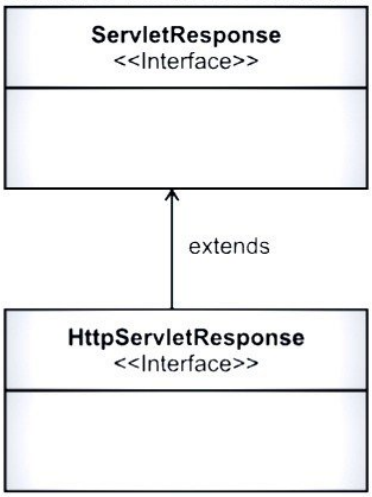


Fig. 6.9: HttpServletResponse Interface

Table 6.6: Methods of HttpServletResponse

Methods	Description
`void addCookie(Cookie cookie)`	This method adds the specified cookie to the response.
`void sendRedirect(String location)`	This method sends a temporary redirect response to the client using the specified redirect location URL and clears the buffer.
`String getHeader(String name)`	This method gets the value of the response header with the given name.
`void setHeader(String name, String value)`	This method sets a response header with the given name and value.
`void setStatus(int sc)`	This method sets the status code for this response.
`void sendError(int sc, String msg)`	This method sends an error response to the client using the specified status and clears the buffer.

GET Method:

- The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? (question mark) symbol.
- The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's Location:box. Never use the GET method if you have password or other sensitive information to pass to the server. The GET method has size limitation: only 1024 characters can be used in a request string.

Program 6.1: Program for GetMethod.

```
<html>
   <body>
      <form action = "HelloForm" method = "GET">
         First Name: <input type = "text" name = "first_name">
         <br />
         Last Name: <input type = "text" name = "last_name" />
         <input type = "submit" value = "Submit" />
      </form>
   </body>
</html>

//Servlet Program
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloForm extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse
   response)
   throws ServletException, IOException
   {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      String docType =
         "<!doctype html public \"-//w3c//dtd html 4.0 " +
         "transitional//en\">\n";
```

```
        out.println(docType +
          "<html>\n" +
            "<body bgcolor = \"#f0f0f0\">\n" +
              "<h1 align = \"center\">" + title + "</h1>\n" +
              "<ul>\n" +
                "  <li><b>First Name</b>: "
                + request.getParameter("first_name") + "\n" +
                "  <li><b>Last Name</b>: "
                + request.getParameter("last_name") + "\n" +
              "</ul>\n" +
            "</body>" +
          "</html>"
        );
    }
}
```

Output:

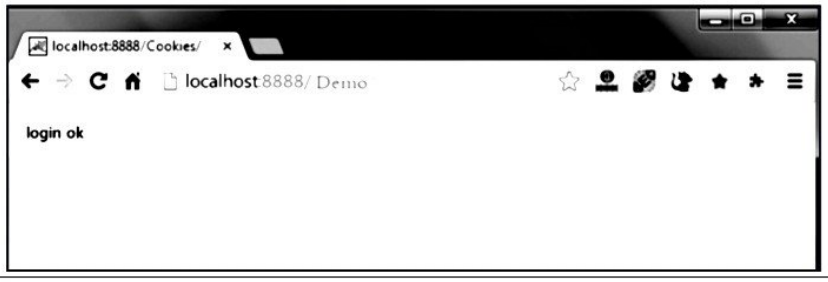
POST Method:

- A generally more reliable method of passing information to a backend program is the POST method. This packages the information in exactly the same way as GET method, but instead of sending it as a text string after a ? (question mark) in the URL it sends it as a separate message. This message comes to the backend program in the form of the standard input which you can parse and use for your processing. Servlet handles this type of requests using doPost() method.

Program 6.2: Program for doPost Method.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginServlet extends HttpServlet
{
   protected void doPost(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException
   {
      String user = req.getParameter("user");
      String pass = req.getParameter("password");
      if ("kamil".equals(user) && "kamil".equals(pass))
      {
         response(resp, "login ok");
      } else
      {
         response(resp, "invalid login");
      }
   }
   private void response(HttpServletResponse resp, String msg) throws
   IOException
   {
      PrintWriter out = resp.getWriter();
      out.println("<html>");
      out.println("<body>");
      out.println("<t1>" + msg + "</t1>");
      out.println("</body>");
      out.println("</html>");
   }
}
```

Ouptut:



6.5 HANDLING A DATA FROM HTML TO A SERVLET

6.5.1 Reading Form Data from Servlet

- Handling form data represented in HTML page is a very common task in web development. A typical scenario is the user fills in fields of a form and submits it. The server will process the request based on the submitted data, and send response back to the client.
- The following figure depicts that workflow with Java servlet on the server side:

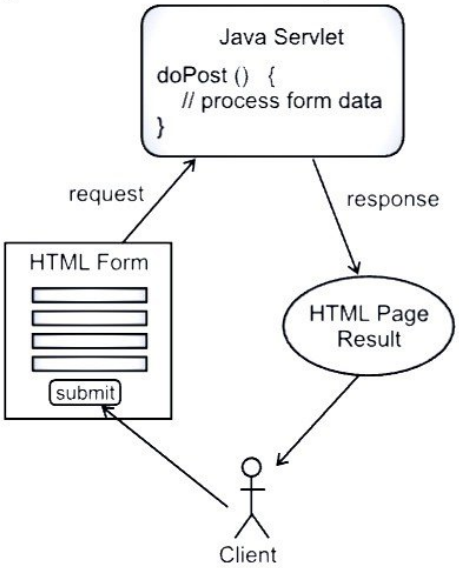


Fig. 6.10: Workflow with Java servlet on the server side

- To create a form in HTML we need to use the following tags:
  - <form>: to create a form to add fields in its body.
  - <input>, <select>, <textarea>...: to create form fields like text boxes, dropdown list, text area, check boxes, radio buttons,... and submit button.
- To make the form works with Java servlet, we need to specify the following attributes for the <form> tag:
  - **method**="post": to send the form data as an HTTP POST request to the server. Generally, form submission should be done in HTTP POST method.
  - **action**="URL of the servlet": specifies relative URL of the servlet which is responsible for handling data posted from this form.
- For example, following is HTML code of a login form:

```
<form name="loginForm" method="post" action="loginServlet">
Username: <input type="text" name="username"/> <br/>
Password: <input type="password" name="password"/> <br/>
<input type="submit" value="Login" />
</form>
```

- This form would look like this in browser:

Fig. 6.11

- On the server side, we need to create a Java servlet which is mapped to the URL: loginServlet, as specified in the form's action attribute. Following is code of the servlet:

```
public class LoginServlet extends HttpServlet
{
   protected void doPost(HttpServletRequest request,
   HttpServletResponse response)
   throws ServletException, IOException
   {
      // code to process the form...
   }
}
```

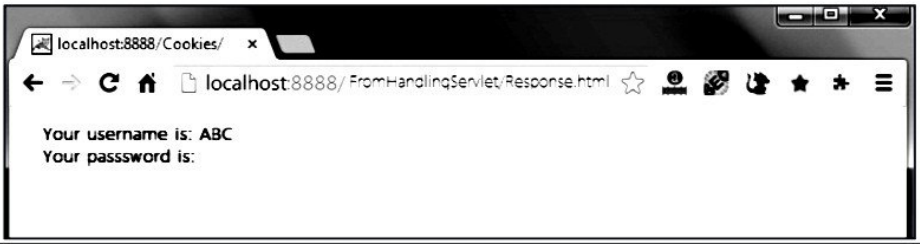
- When the user submits the login form above, the servlet's doPost() method will be invoked by the servlet container. Typically we will do the following tasks inside doPost() method:
  - Read values of the fields posted from the form via the request object (implementation of javax.servlet.http.HttpServletRequest interface).

- Do some processing, e.g. connecting to database to validate the username and password.
- Return response back to the user via the respone object (implementation of javax.servlet.http.HttpServletResponse interface).

Program 6.3: Servlet Class to process the login form:

```
package net.codejava.servlet;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class LoginServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws
    ServletException, IOException
    {
        // read form fields
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        System.out.println("username: " + username);
        System.out.println("password: " + password);
        // do some processing here...
        // get response writer
        PrintWriter writer = response.getWriter();
        // build HTML code
        String htmlRespone = "<html>";
        htmlRespone += "<h2>Your username is: " + username + "<br/>";
        htmlRespone += "Your password is: " + password + "</h2>";
        htmlRespone += "</html>";
        // return response
        writer.println(htmlRespone);
    }
}
```

Output:



6.5.2 Servlet - Database Communication

- We are going to describe how to connect to the database in servlet. We have been provides some classes and API with which we will make use of the database connection. Database plays an important role in any application which requires to store the data and to fetch the data.

Program 6.4: Program for Database Connection using Servlet.

```
import java.io.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ConnectorJDBC extends HttpServlet{
protected void doGet(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException
{
   try
   {
      doProcess(request,response);
   }  catch(SQLException e)
   {
      e.printStackTrace();
   }
}
protected   void   doPost(HttpServletRequest   request,   HttpServletResponse
response)
```

```
throws ServletException, IOException
{
   try
   {
      doProcess(request,response);
   }  catch(SQLException e)
   {
      e.printStackTrace();
   }
}
private void doProcess(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException , SQLException
{
   Connection conn = null;
   Statement stmt = null;
   ResultSet rs = null;
   PrintWriter out = response.getWriter();
   try
   {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      conn = DriverManager.getConnection
                      ("jdbc:mysql://localhost:3306/kamil","root","root");
      stmt = conn.createStatement();
      rs = stmt.executeQuery("select * from userinformation");
      out.print("<html>");
      out.print("<head>");
      out.print("<title>Hello Connect Database</title>");
      out.print("</head>");
      out.print("<name>");
      out.print("<h1>Name from database</h1>");
      out.print("<br/>");
      System.out.println("Get value from table userinformation.");
      while(rs.next())
      {
         out.println("<tr>");
         out.print("Name : " + rs.getString("name"));
```

```
            out.print("\t\t\t");
            out.print("Gender : " + rs.getString("gender"));
            out.print("Address : " + rs.getString("address"));
            out.print("</tr><br/>");
        }
        out.print("</body>");
        out.print("</html>");
        // out.flush();
        }
        catch(Exception e)
        {
        an e.printStackTrace();
        }  finally
        {
            if(stmt != null)
            {
                stmt.close();
            }
            if(conn != null)
            {
                conn.close();
            }
        }
    }
}
```

Output:

6.6 SESSION TRACKING

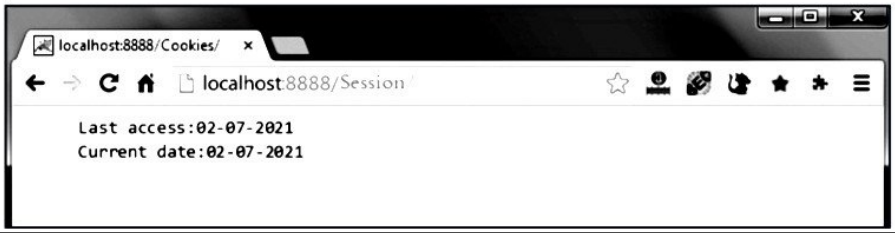
- Session is used to store everything that we can get from the client from all the requests the client makes.
- The basic concept behind session is, whenever a user starts using our application, we can save a unique identification information about him, in an object which is available throughout the application, until it's destroyed. So wherever the user goes, we will always have his information and we can always manage which user is doing what. Whenever a user wants to exit from your application, destroy the object with his/her information.
- HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism. A session can be created via the getSession( ) method of HttpServletRequest.
- An HttpSession object is returned. This object can store a set of bindings that associate names with objects. The setAttribute( ), getAttribute( ), getAttributeNames( ), and removeAttribute( ) methods of HttpSession manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular client.
- The following servlet illustrates how to use session state. The getSession( ) method gets the current session. A new session is created if one does not already exist. The getAttribute( ) method is called to obtain the object that is bound to the name "date". That object is a Date object that encapsulates the date and time when this page was last accessed.
- A Date object encapsulating the current date and time is then created. The setAttribute( ) method is called to bind the name "date" to this object.

Program 6.5: To display last access and current date using session.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DateServlet extends HttpServlet
{
   public void doGet(HttpServletRequest request,
   HttpServletResponse response)
   throws ServletException, IOException
   {
      //Get the HttpSession object.HttpSession hs = request.getSession(true);
      // Get writer.response.setContentType("text/html");
```

```
        PrintWriter pw = response.getWriter();
        pw.print("<B>");
        // Display date/time of last access.
        Date date = (Date)hs.getAttribute("date");
        if(date != null)
        {
        pw.print("Last access: " + date + "<br>");
        }
        // Display current date/time.
        date = new Date();
        hs.setAttribute("date", date);
        pw.println("Current date: " + date);
    }
}
```

Output:



6.6.1 Cookies and HTTP Session

Cookie:

- Cookie is a class from javax.servlet.http package. Cookie stores small strings of data. A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.
- Cookies stay all through the session. Programmer can store the session data with Cookie object. It is another style of tracking session data.
- Servlet Cookies are created on the server (by Servlet container) and sent to the client's browser for temporary storage.
- Servlet Cookies can be sent to and from between client and server in their conversation. Client is not aware of the entire process of Cookie management as everything is done by the server and client is kept completely transparent.
- Servlet Cookies are used mainly for tracking different types of client's information. That is, the Web server can store data on the client browser in the form of Cookie objects.

Uses of a Cookie:

1. To store User name and Password.
2. To write Shopping cart in e-Commerce application.
3. Advertisement on client browser.
4. Customization of Web sites etc.

- The Cookie object created on the server is attached to response object and sent to client. The Cookie carries implicitly (as response header) the server information like name, path, host and connection type etc. and the combination of this information is used by server to know cookie is coming from which client.

Types of Cookies:

1. **Session Cookies:**
   - Session cookies get expired and deleted from browser when the browser closes, thereby, they do not carry any expiration time constraint parameter.
   - They are stored temporarily (just for the usage of session) in the RAM memory.
2. **Persistent Cookies:**
   - As the name indicates, they persist long time on the client browser.
   - The time of expiration can be set programmatically for each Cookie.
   - These Cookies are stored on hard disk.
   - The Cookie gets deleted automatically when the time or age expires.

- Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:
  - **AddCookie.htm:** Allows a user to specify a value for the cookie named MyCookie.
  - **AddCookieServlet.java:** Processes the submission of AddCookie.htm.
  - **GetCookiesServlet.java:** Displays cookie values.
- The HTML source code for **AddCookie.htm** is shown below.
- This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1" method="post" action=
     "http://localhost:8080/servlets-examples/servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type=textbox name="data" size=25 value="">
<input type=submit value="Submit">
</form>
</body>
</html>
```

- The source code for AddCookieServlet.java is shown in the following listing. It gets the value of the parameter named "data". It then creates a Cookie object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the addCookie( ) method. A feedback message is then written to the browser.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class AddCookieServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
    {
        // Get parameter from HTTP request.
        String data = request.getParameter("data");
        // Create cookie.
        Cookie cookie = new Cookie("MyCookie", data);
        // Add cookie to HTTP response.
        response.addCookie(cookie);
        // Write output to browser.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>MyCookie has been set to");
        pw.println(data);
        pw.close();
    }
}
```

- The source code for GetCookiesServlet.java is shown in the following listing. It invokes the getCookies( ) method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP

response. Observe that the getName( ) and getValue( ) methods are called to obtain this information.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookiesServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
    {
        // Get cookies from header of HTTP request.
        Cookie[] cookies = request.getCookies();
        // Display these cookies.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");
        for(int i = 0; i < cookies.length; i++)
        {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            pw.println("name = " + name + "; value = " + value);
        }
        pw.close();
    }
}
```

Output:

HttpSession:

- HttpSession object is used to store entire session with a specific client. We can store, retrieve and remove attribute from HttpSession object. Any servlet can have access to HttpSession object throughout the getSession() method of the HttpServletRequest object.

- The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user.
- You would get HttpSession object by calling the public method getSession() of HttpServletRequest, as below:

```
HttpSession session = request.getSession();
```

- You need to call request.getSession() before you send any document content to the client.

Table 6.7: Methods of HttpSession object

Sr. No.	Method and Description
1.	`public Object getAttribute(String name)` This method returns the object bound with the specified name in session, or null if no object is bound under the name.
2.	`public Enumeration getAttributeNames()` This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3.	`public long getCreationTime()` This method returns the time when session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4.	`public String getId()` This method returns a string containing the unique identifier assigned to session.
5.	`public long getLastAccessedTime()` This method returns the last accessed time of the session, in the format of milliseconds since midnight January 1, 1970 GMT
6.	`public int getMaxInactiveInterval()` This method returns the maximum time interval (seconds), that the servlet container will keep the session open between client accesses.
7.	`public void invalidate()` This method invalidates session and unbinds any objects bound to it.
8.	`public boolean isNew()` This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9.	`public void removeAttribute(String name)` This method removes the object bound with the specified name from this session.
10.	`public void setAttribute(String name, Object value)` This method binds an object to session, using the name specified.
11.	`public void setMaxInactiveInterval(int interval)` This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

- Following table compares Session and Cookie:

Table 6.8: Comparision between session and cookie

Sr. No.	Session	Cookie
1.	Data on server-side.	Data on client side.
2.	Unlimited side of data as per as server capability.	Limited support for data handling.
3.	It can store any type of data.	Only text.
4.	Age of data is not fixed.	Fixed.
5.	Destroy after session timeout or logout.	Remains on client machine.
6.	Less data traveling over the network.	All cookie need to travel each time client sends request to server.
7.	More secure mechanism to session tracking	Less secure.

HttpSession Example:

- Following example describes how to use the HttpSession object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
// Extend HttpServlet class
public class SessionTrack extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse
   response)
   throws ServletException, IOException
   {
      // Create a session object if it is already not  created.
      HttpSession session = request.getSession(true);
      // Get session creation time.
      Date createTime = new Date(session.getCreationTime());
      // Get last access time of this web page.
      Date lastAccessTime = new Date(session.getLastAccessedTime());
      String title = "Welcome Back to my website";
```

```
Integer visitCount = new Integer(0);
String visitCountKey = new String("visitCount");
String userIDKey = new String("userID");
String userID = new String("ABCD");
// Check if this is new comer on your web page.
if (session.isNew())
{
   title = "Welcome to my website";
   session.setAttribute(userIDKey, userID);
}
else
{
   visitCount = (Integer)session.getAttribute(visitCountKey);
   visitCount = visitCount + 1;
   userID = (String)session.getAttribute(userIDKey);
}
session.setAttribute(visitCountKey,  visitCount);
// Set response content type
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String docType ="<!doctype html public \"-//w3c//dtd html 4.0 " +
"transitional//en\">\n";
out.println(docType +"<html>\n" +"<head><title>" + title +
"</title></head>\n" +"<body bgcolor = \"#f0f0f0\">\n" +
"<h1 align = \"center\">" + title + "</h1>\n" +
"<h2 align = \"center\">Session Infomation</h2>\n" +
"<table border = \"1\" align = \"center\">\n" +
"<tr bgcolor = \"#949494\">\n" +
"  <th>Session info</th><th>value</th>
</tr>\n" +

"<tr>\n" +
"  <td>id</td>\n" +
"  <td>" + session.getId() + "</td>
</tr>\n" +

"<tr>\n" +
"  <td>Creation Time</td>\n" +
```

```
            "  <td>" + createTime + "  </td>
            </tr>\n" +

            "<tr>\n" +
            "  <td>Time of Last Access</td>\n" +
            "  <td>" + lastAccessTime + "  </td>
            </tr>\n" +

            "<tr>\n" +
            "  <td>User ID</td>\n" +
            "  <td>" + userID + "  </td>
            </tr>\n" +

            "<tr>\n" +
            "  <td>Number of visits</td>\n" +
            "  <td>" + visitCount + "</td>
            </tr>\n" +
            "</table>\n" +
            "</body>
            </html>"
            );
        }
    }
```

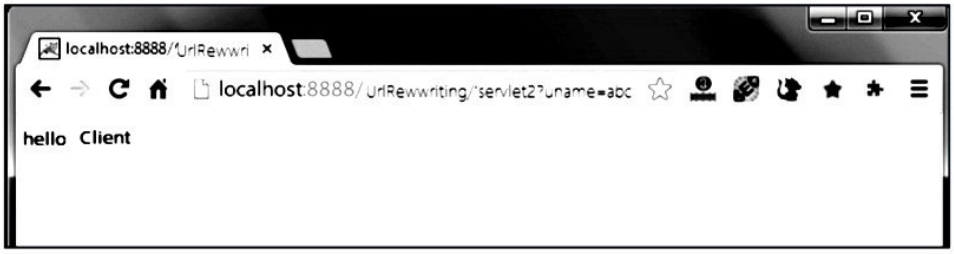
- Compile the above servlet **SessionTrack** and create appropriate entry in web.xml file. Now running http://localhost:8080/SessionTrack would display the result.

Program 6.6: Program to display Hello Client.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloClientServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<HTML><HEAD><TITLE>Hello Client!</TITLE>"+
        "</HEAD><BODY>Hello Client!</BODY></HTML>");
        out.close();
    }
```

```
    public String getServletInfo()
    {
        return "HelloClientServlet //// ";
    }
}
```

Output:



Program 6.7: Servlet counts and displays the number of times it has been accessed since the last server reboot. **[W-18]**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SimpleCounter extends HttpServlet
{
    int count = 0;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        count++;
        out.println("This servlet has been Accessed " + count + " Times.");
    }
}
```

Output:

Program 6.8: Servlet counts and displays the number of times it has been accessed, and saves the count to a file in its destroy() method to make the count persistent.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DCounter extends HttpServlet
{
    int count;
    public void init() throws ServletException
    {
        // Try to load the initial count from our saved persistent state
        FileReader fileReader = null;
        BufferedReader bufferedReader = null;
        try
        {
            fileReader = new FileReader("DCounter.initial");
            bufferedReader = new BufferedReader(fileReader);
            String initial = bufferedReader.readLine();
            count = Integer.parseInt(initial);
            return;
        }
        catch (FileNotFoundException ignored) { }  // no saved state
        catch (IOException ignored) { }            // problem during read
        catch (NumberFormatException ignored) { }  // corrupt saved state
        finally
        {
            // Close the file
            try
            {
                if (bufferedReader != null)
                {
                    bufferedReader.close();
                }
            }
            catch (IOException ignored) { }
        }
        // Check for an init parameter
        String initial = getInitParameter("initial");
```

```
        try
        {
            count = Integer.parseInt(initial);
            return;
        }
        catch (NumberFormatException ignored) { }
        // null or non-integer value
        // Default to an initial count of "0"
        count = 0;
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        count++;
        out.println("This servlet has been accessed " + count + " times.");
    }
    public void destroy()
    {
        super.destroy();  //  optional
        saveState();
    }
    public void saveState()
    {
        // Try to save the accumulated count
        FileWriter fileWriter = null;
        PrintWriter printWriter = null;
        try
        {
            fileWriter = new FileWriter("DCounter.initial");
            printWriter = new PrintWriter(fileWriter);
            printWriter.println(count);
            return;
        }
        catch (IOException e) { // problem during write }
        finally
        {
            // Close the file
```

```
            if (printWriter != null)
            {
                printWriter.close();
            }
        }
    }
}
```

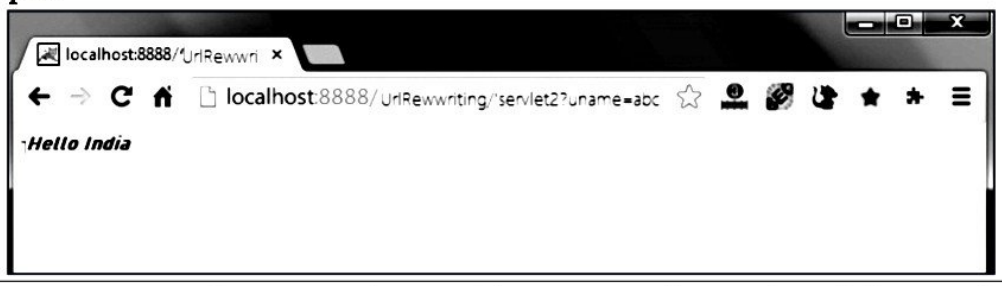
Output:

Program 6.9: Servlet generates and returns a GIF image. The graphic says "Hello India!".

```
import java.io.*;
import java.awt.*;
import javax.servlet.*;
import javax.servlet.http.*;
import Acme.JPM.Encoders.GifEncoder;
public class HelloIndiaGraphics extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        ServletOutputStream out = res.getOutputStream();  // binary output!
        Frame frame = null;
        Graphics g = null;
        try
        {
            // Create an unshown frame
            frame = new Frame();
            frame.addNotify();
            // Get a graphics region, using the Frame
            Image image = frame.createImage(400, 60);
            g = image.getGraphics();
            // Draw "Hello India!"
            g.setFont(new Font("Serif", Font.ITALIC, 48));
```

```
            g.drawString("Hello India!", 10, 50);
            // Encode the off-screen image into a GIF and send it to the
            client res.setContentType("image/gif");
            GifEncoder encoder = new GifEncoder(image, out);
            encoder.encode();
        }
        finally
        {
            // Clean up resources
            if (g != null) g.dispose();
            if (frame != null) frame.removeNotify();
        }
    }
}
```

Output:



Program 6.10: Servlet uses session tracking to count the number of times a client has accessed it. The servlet also displays all the bindings for the current session.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class STracker extends HttpServlet
{
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        // Get the current session object, create one if necessary
        HttpSession session = req.getSession();
        // Increment the hit count for this page. The value is saved
        // in this client's session under the name "tracker.cnt".
        Integer count = (Integer)session.getAttribute("tracker.cnt");
```

```
        if (count == null)
        count = new Integer(1);
        else
        count = new Integer(count.intValue() + 1);
        session.setAttribute("tracker.cnt", count);
        out.println("<HTML><HEAD><TITLE>STracker</TITLE></HEAD>");
        out.println("<BODY><H1>Session Tracking Demo</H1>");
        // Display the hit count for this page
        out.println("You've visited this page " + count +((count.intValue()
        == 1) ? " time." : " times."));
        out.println("<P>");
        out.println("<H2>Here is your session data:</H2>");
        Enumeration enum = session.getAttributeNames();
        while (enum.hasMoreElements())
        {
           String name = (String) enum.nextElement();
           out.println(name + ": " + session.getAttribute(name) + "<BR>");
        }
        out.println("</BODY></HTML>");
    }
}
```

Output:

Program 6.11: Write java program to accept user name, password from HTML and pass to servlet and compare with DB and display appropriate message.

```
Login.html
<form action="servlet1" method="post">
User:<input type="text" name="username"/><br/><br/>
Password:<input type="password" name="userpass"/><br/><br/>
<input type="submit" value="Submit"/>
</form>
// FirstServlet.java
```

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class FirstServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
            String n=request.getParameter("username");
            String p=request.getParameter("userpass");
            if(Login.validate(n, p))
            {
                RequestDispatcher rd=request.getRequestDispatcher("servlet2");
                rd.forward(request,response);
            }
            else
            {
                out.print("Sorry username or password error");
                RequestDispatcher rd=request.getRequestDispatcher("index.html");
                rd.include(request,response);
            }
            out.close();
    }
}
Login.java
import java.sql.*;
public class Login
{
    public static boolean validate(String name,String pass)
    {
        boolean status=false;
```

```
        try
        {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con=DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
            PreparedStatement ps=con.prepareStatement(
            "select * from userreg where name=? and pass=?");
            ps.setString(1,name);
            ps.setString(2,pass);
            ResultSet rs=ps.executeQuery();
            status=rs.next();
        }
        catch(Exception e){System.out.println(e);}
        return status;
    }
}
```

6.7 INTRODUCTION TO JSP

- Java Server Pages (JSP) is a technology which is used to develop web pages by inserting Java code into the HTML pages by making special JSP tags. The JSP tags which allow java code to be included into it are <% ----java code----%>.
- It can consist of either HTML or XML (combination of both is also possible) with JSP actions and commands.
- It can be used as HTML page, which can be used in forms and registration pages with the dynamic content into it.
- Dynamic content includes some fields like dropdown, checkboxes, etc. whose value will be fetched from the database. This can also be used to access JavaBeans objects.
- We can share information across pages using request and response objects. JSP can be used for separation of the view layer with the business logic in the web application.
- **JSP** technology is used to create web application just like Servlet technology.

Why use JSP?

- In JSP, the execution is much faster compared to other dynamic languages.
- It is much better than Common Gateway Interface (CGI).
- JSP are always compiled before it's processed by the server as it reduces the effort of the server to create process.
- JSP are built over Java Servlets API. Hence, it has access to all Java APIs, even it has access to JDBC EJB and other components of java.
- JSP are used in MVC architecture (which will be covered in MVC architecture topic) as view layer.

- The request is processed by a view layer which is JSP and then to servlet layer which is java servlet and then finally to a model layer class which interacts with the database.

  Example:

```
<html>
   <body>  Hello World!
      <br> Current time is
         <%= new java.util.Date() %>
   </body>
</html>
```

Advantages of JSP:

1. **Separation of Static and Dynamic Contents:** JSP enables the separation of static contents from dynamic contents. The dynamic contents are generated via programming logic and inserted into the static template. This greatly simplifies the creation and maintenance of web contents.
2. **Reuse of Components and Tag Libraries:** The dynamic contents can be provided by reusable components such as JavaBean, Enterprise JavaBean (EJB) and tag libraries - you do not have to re-inventing the wheels.
3. **Java's Power and Portability:** JSPs are internally compiled into Java Servlets.
4. **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.

- JSPs are internally compiled into Java Servlets. Anything that can be done using JSPs can also be accomplished using Java servlets. However, it is important to note that servlets and JSPs are complementary technologies, NOT replacement of each other. Servlet can be viewed as "**HTML inside Java**", which is better for implementing business logic - as it is Java dominant.
- JSP, on the other hand, is "**Java inside HTML**", which is superior for creating presentation - as it is HTML dominant. In a typical Model-View-Control (MVC) application, servlets are often used for the Controller (C), which involves complex programming logic.
- JSPs are often used for the View (V), which mainly deals with presentation. The Model (M) is usually implemented using JavaBean or EJB.

Apache TOMCAT server:

- JSPs, like servlets, are server-side programs run inside a HTTP server. To support JSP/servlet, a Java-capable HTTP server is required. Tomcat Server (@ http://tomcat.apache.org) is the official reference implementation (RI) for Java servlet and JSP, provided free by Apache (@ http://www.apache.org) - an open-source software foundation.

- The web server needs a JSP engine, i.e, a container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. This tutorial makes use of Apache which has built-in JSP container to support JSP pages development.
- A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs.
- Fig. 6.12 shows the position of JSP container and JSP files in a Web application.

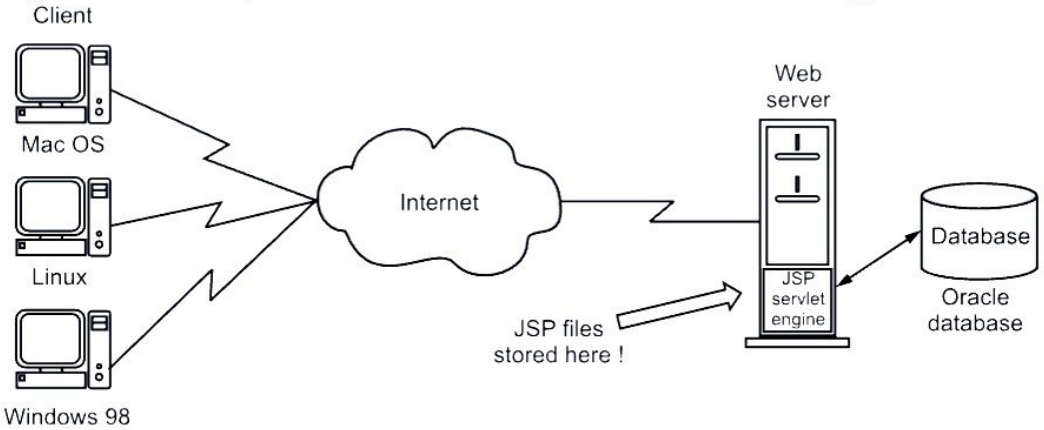


Fig. 6.12: Position of JSP container and JSP files in a Web application

JSP Processing:

- The following steps explain how the web server creates the Webpage using JSP:
  1. As with a normal page, your browser sends an HTTP request to the web server.
  2. The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of **.html**.
  3. The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code. This code implements the corresponding dynamic behavior of the page.
  4. The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
  5. A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format. The output is furthur passed on to the web server by the servlet engine inside an HTTP response.
  6. The web server forwards the HTTP response to your browser in terms of static HTML content.
  7. Finally, the web browser handles the dynamically-generated HTML page inside the HTTP response exactly as if it were a static page.

- All the above mentioned steps can be seen in the Fig. 6.13.

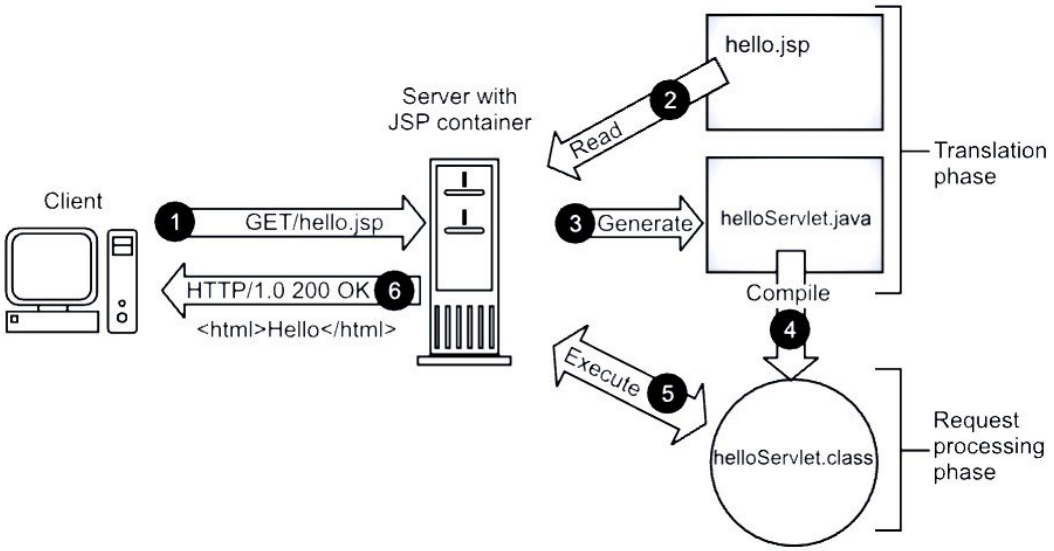


Fig. 6.13: JSP Processing

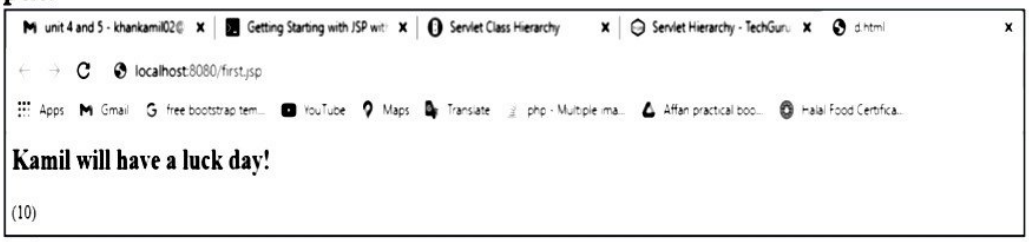
- Typically, the JSP engine checks to see whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with the other scripting languages (such as PHP) and therefore faster.
- So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet.

6.7.1 Simple JSP Program

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95)
   {
      %>
         <h2>Kamil will have a luck day!</h2><p>(<%= num %>)</p>
      <%
    }
```

```
    else
    {
       %>
          <h2>Kamil Well, life goes on ... </h2><p>(<%= num %>)</p>
       <%
     }
    %>
    <a href="<%= request.getRequestURI() %>"><h3>Try Again</h3></a>
</body>
</html>
```

Output:



6.8 LIFE CYCLE OF A JSP

- When first time a JSP page is requested, the necessary servlet code is generated and loaded into the servlet container. Until the JSP page gets rendered fully, the servlet code handles the requests coming from the browser. Whenever there is a change in the JSP page, the JSP compiler regenerates the servlet code.
- JSP lifecycle phases:
  1. **Translation:** JSP container verifies the JSP page code and parses it to generate the servlet source code. For your note, if the JSP page name is <Login.jsp>, the generated servlet class name would be like Login_jsp, and the Java file name would be Login_jsp.java.
  2. **Compilation:** JSP container compiles the JSP page and creates a class file in this phase.
  3. **Class Loading:** At this stage, the container reads the JSP class into application memory.
  4. **Instantiation:** This phase guides the container to execute the no-args constructor of the generated JSP class and loads it into memory to instantiate it.
  5. **Initialization:** Container calls the JSP class init method and initializes the servlet config with the parameters specified in the deployment descriptor. This phase makes sure the JSP is fit to respond to the client requests.
  6. **Request Processing:** This is the phase where the JSP page creates threads to processes a request. Every new thread leads to the creation of ServletRequest and ServletResponse object followed by a call to the JSP service method.
  7. **Destroy:** This is the terminal phase of JSP life cycle where the JSP class gets unloaded from the memory. This stage comes in the picture when the application is undeployed, or the server is shut down.

Life Cycle of JSP:

- A JSP page is converted into Servlet in order to service requests. The translation of a JSP page to a Servlet is called Lifecycle of JSP.
- JSP Lifecycle is exactly same as the Servlet Lifecycle, with one additional first step, which is, translation of JSP code to Servlet code.
- Following are the JSP Lifecycle steps:

  Step 1 : Translation of JSP to Servlet code.
  Step 2 : Compilation of Servlet to bytecode.
  Step 3 : Loading Servlet class.
  Step 4 : Creating servlet instance.
  Step 5 : Initialization by calling jspInit() method.
  Step 6 : Request Processing by calling _jspService() method.
  Step 7 : Destroying by calling jspDestroy() method.

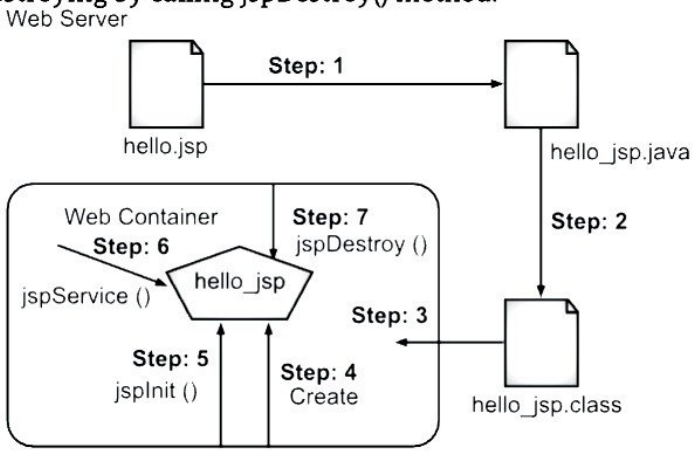


Fig. 6.14: JSP Lifecycle

- When a browser asks for a Java Server Page, the Java Server Page engine first checks to see whether it needs to compile the page. If the page not compiled, or if the Java Server Page has been updated since it was last compiled, the Java Server Page engine compiles the page.
- The compilation process involves three steps:
  1. Java Server Page Parsing.
  2. Convert the Java Server Page into a servlet.
  3. Compiling the servlet.

JSP Initialization:

- When a container loads a Java Server Page it invokes the jspInit() method before servicing any requests. If you need to perform Java Server Page -specific initialization, override the jspInit() method:

```
public void jspInit()
{
   // Initialization code...
}
```

JSP Execution:

- This phase of the Java Server Page life cycle shows all interactions with requests until the JSP is destroyed. Whenever a browser requests a Java Server Page and the page has been loaded and initialized, the Java Server Page engine invokes the _jspService() method in the Java Server Page.
- The _jspService() method takes an HttpServletRequest and an HttpServletResponse as its parameters as follows:

```
void _jspService(HttpServletRequest request, HttpServletResponse
   response) {
// Service handling code...
}
```

- The _jspService() method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e, GET, POST, DELETE, etc.

JSP Destroy:

- In this phase of the JSP life cycle represents when a JSP is being removed from use by a container.
- The jspDestroy() method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.
- The jspDestroy() method has the following form:

```
public void jspDestroy()
{
   // Your cleanup code goes here.
}
```

What happens to a JSP when it is translated into Servlet?

- The code written inside <% %> is JSP code.

```
<html>
    <head>
        <title>My First JSP Page</title>
    </head>
   <%
       int cnt = 0;
   %>
  <body>
        Page Cnt is:
        <% out.println(++cnt); %>
  </body>
</html>
```

- The above JSP page(hello.jsp) becomes this Servlet

```
public class hello_jsp extends HttpServlet
{
   public void _jspService(HttpServletRequest request, HttpServletResponse
   response)
   throws IOException,ServletException
   {
      PrintWriter out = response.getWriter();
      response.setContenType("text/html");
      out.write("<html><body>");
      int cnt=0;
      out.write("Page cnt is:");
      out.print(++cnt);
      out.write("</body></html>");
   }
}
```

- As JSP developer, you do not have to worry about how a JSP page is converted to a Servlet, as it is done automatically by the web container.

6.9 IMPLICIT OBJECTS [W-18, S-19]

- JSP provide access to some implicit object which represent some commonly used objects for servlets that JSP page developers might need to use.
- For example, you can retrieve HTML form parameter data by using request variable, which represent the HttpServletRequest object.

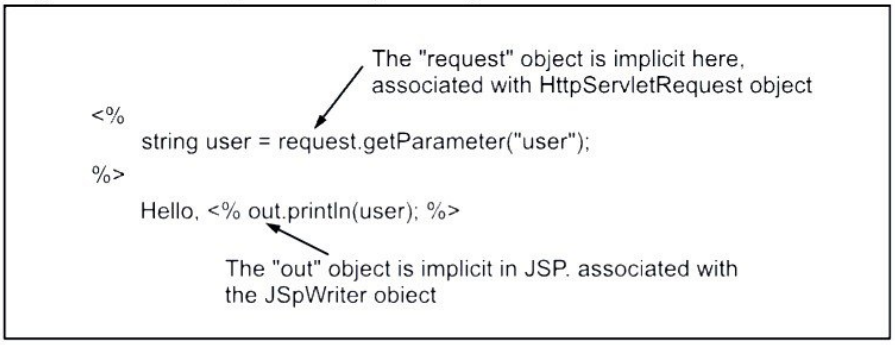


Fig. 6.15: Implicit object

- JSP Implicit Objects are the Java objects available in each page and can be used directly without explicitly declaration.
- Implicit Objects in JSP. These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables.

- Example: When you will create an application where in User sessions have to be created session will come into picture, request is used when you have form submissions in your application etc.
- A JSP page consists of HTML tags and JSP tags. The jsp pages are easier to maintain than servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tag etc.
- The Server **Syntax** is:

```
PrintWriter out=response.getWriter();
i/JSP Syntax
<%out.print("Welcome to W3adda.com);%>
```

- Example: In this example we are simply displaying our message.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>JSP Page</title>
</head>
<body>
<-% out.print("Welcome to W3adda.com");%>
</body>
</html>
```

Following are the JSP implicit objects:

1. out Implicit object:

- For writing any data to the buffer, JSP provides an implicit object named out. It is the object of JspWriter. In case of servlet you need to write: `PrintWriter out=response.getWriter();`
- But in JSP, you don't need to write this code.
- Example of out implicit object which displaying date and time:

index.jsp

```
<html>
<body>
<% out.print("Today is:"+java.util.Calendar.getInstance().getTime()); %>
</body>
</html>
```

2. request Implicit Object:

- The JSP request is an implicit object of type HttpServletRequest i.e. created for each jsp request by the web container. It can be used to get request information such as parameter, header information, remote address, server name, server port, content type, character encoding etc.
- It can also be used to set, get and remove attributes from the jsp request scope.

- Example: Let's see the simple example of request implicit object where we are printing the name of the user with welcome message.

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
```

3. response Implicit Object:

- In JSP, response is an implicit object of type HttpServletResponse. The instance of HttpServletResponse is created by the web container for each jsp request.
- It can be used to add or manipulate response such as redirect response to another resource, send error etc.
- Let's see the example of response implicit object where we are redirecting the response to the Google.

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
response.sendRedirect("http://www.google.com");
%>
```

4. config Implicit Object:

- In JSP, config is an implicit object of type ServletConfig. This object can be used to get initialization parameter for a particular JSP page. The config object is created by the web container for each jsp page.
- Generally, it is used to get initialization parameter from the web.xml file.
- Example of config implicit object:

index.html

```
<form action="welcome">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

web.xml

```
<web-app>
<servlet>
<servlet-name>sonoojaiswal</servlet-name>
<jsp-file>/welcome.jsp</jsp-file>
<init-param>
<param-name>dname</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</init-param>
</servlet>
<servlet-mapping>
<servlet-name>sonoojaiswal</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>
</web-app>
```

welcome.jsp

```
<%
out.print("Welcome "+request.getParameter("uname"));
String driver=config.getInitParameter("dname");
out.print("driver name is="+driver);
%>
```

5. Application Implicit Object:

- In JSP, application is an implicit object of type ServletContext.
- The instance of ServletContext is created only once by the web container when application or project is deployed on the server.
- This object can be used to get initialization parameter from configuaration file (web.xml). It can also be used to get, set or remove attribute from the application scope.
- This initialization parameter can be used by all jsp pages.
- Example of application implicit object:

index.html:

```
<form action="welcome">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

web.xml

```
<web-app>
<servlet>
<servlet-name>sonoojaiswal</servlet-name>
```

```
<jsp-file>/welcome.jsp</jsp-file>
</servlet>
<servlet-mapping>
<servlet-name>sonoojaiswal</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>
<context-param>
<param-name>dname</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</context-param>
</web-app>
```

welcome.jsp

```
<%
out.print("Welcome "+request.getParameter("uname"));
String driver=application.getInitParameter("dname");
out.print("driver name is="+driver);
%>
```

6. session Implicit Object:

- In JSP, session is an implicit object of type HttpSession.The Java developer can use this object to set,get or remove attribute or to get session information.
- Program example of session implicit object:

index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

welcome.jsp

```
<html>
<body>
<%
String name=request.getParameter("uname");
out.print("Welcome "+name);
session.setAttribute("user",name);
<a href="second.jsp">second jsp page</a>
%>
</body>
</html>
```

second.jsp

```
<html>
<body>
<%
String name=(String)session.getAttribute("user");
out.print("Hello "+name);
%>
</body>
</html>
```

7. pageContext Implicit Object:

- In JSP, pageContext is an implicit object of type PageContext class.The pageContext object can be used to set,get or remove attribute from one of the following scopes:
  - page
  - request
  - session
  - application
- In JSP, page scope is the default scope.
- Example of pageContext implicit object:

index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

welcome.jsp

```
<html>
<body>
<%
String name=request.getParameter("uname");
out.print("Welcome "+name);
pageContext.setAttribute("user",name,PageContext.SESSION_SCOPE);
<a href="second.jsp">second jsp page</a>
%>
</body>
</html>
```

second.jsp

```
<html>
<body>
<%
String
   name=(String)pageContext.getAttribute("user",PageContext.SESSION_SCO
   PE);
out.print("Hello "+name);
%>
</body>
</html>
```

8. page Implicit Object:

- In JSP, page is an implicit object of type Object class.This object is assigned to the reference of auto generated servlet class. It is written as:

```
Object page=this;
```

- For using this object it must be cast to Servlet type.
- For example:

```
<% (HttpServlet)page.log("message"); %>
```

- Since, it is of type Object it is less used because you can use this object directly in jsp.For example:

```
<% this.log("message"); %>
```

9. exception Implicit Object:

- In JSP, exception is an implicit object of type java.lang.Throwable class. This object can be used to print the exception. But it can only be used in error pages.It is better to learn it after page directive. Let's see a simple example:
- Example of exception implicit object:

error.jsp

```
<%@ page isErrorPage="true" %>
<html>
<body>
Sorry following exception occured:<%= exception %>
</body>
</html>
```

6.10 SCRIPTING ELEMENTS

6.10.1 Scriptlets [W-18]

- A scriptlet is a set of one or more Java language statements intended to be used to process an HTTP request.
- The **syntax** of a scriptlet is as follows:

```
<% statement; [statement; ...] %>
```

- The JSP compiler simply includes the contents of scriptlet verbatim in the body of the _jspService() method. A JSP page may contain any number of scriptlets. If multiple scriptlets exist, they are each appended to the _jspService() method in the order in which they are coded. This being the case, a scriptlet may contain an open curly brace that is closed in another scriptlet.
- Consider the following JSP page, which produces a Fahrenheit to Celsius temperature conversion table:

```
<%@ page import="java.text.*" %>
<TABLE BORDER=0 CELLPADDING=3>
<TR>
<TH>Degrees<BR>Fahrenheit</TH>
<TH>Degrees<BR>Celsius</TH>
</TR>
<%
   NumberFormat fmt = new DecimalFormat("###.000");
   for (int f = 32; f <= 212; f += 20) {
   double c = ((f - 32) * 5) / 9.0;
   String cs = fmt.format(c);
%>
<TR>
   <TD ALIGN="RIGHT"><%= f %></TD>
   <TD ALIGN="RIGHT"><%= cs %></TD>
</TR>
<%
}
%>
</TABLE>
```

6.10.2 Declarations

- Like scriptlets, declarations contain Java language statements, but with one big difference: scriptlet code becomes part of the _jspService() method, whereas declaration code is incorporated into the generated source file outside the _jspService() method.
- The **syntax** of a declaration is as follows:

```
<%! statement; [statement; ...] %>
```

- Declaration sections can be used to declare class or instance variables, methods, or inner classes. Unlike scriptlets, they have no access to the implicit objects described in the next section. If you use a declaration section to declare a method that needs to use the request object, for example, you need to pass the object as a parameter to the method.

- The following shows an example of a JSP page that uses a declaration section:

```
<%@ page
   errorPage="ErrorPage.jsp"
   import="java.io.*,java.util.*"
%>
<%
   Enumeration enames;
   Map map;
   String title;
   // Print the request headers
   map = new TreeMap();
   enames = request.getHeaderNames();
   while (enames.hasMoreElements())
   {
      String name = (String) enames.nextElement();
      String value = request.getHeader(name);
      map.put(name, value);
   }
   printTable(out, map, "Request Headers");
   // Print the session attributes
   map = new TreeMap();
   enames = session.getAttributeNames();
   while (enames.hasMoreElements())
   {
      String name = (String) enames.nextElement();
      String value = "" + session.getAttribute(name);
      map.put(name, value);
   }
   printTable(out, map, "Session Attributes");
%>
```

6.10.3 Expressions

- JSP provides a simple means for accessing the value of a Java variable or other expression and merging that value with the HTML in the page. The syntax is:

  `<% = exp%>`

  Where `exp` is any valid Java expression. The expression can have any data value, as long as it can be converted to a string. This conversion is usually done simply by generating an `out.print()` statement.
- For example, the current time is, `<%= new java.util.Date() %>`

- Following example shows a JSP Expression:

```
<html>
   <head><title>A Comment Test</title></head>
   <body>
      <p>Today's date: <%= (new java.util.Date()).toLocaleString()%></p>
   </body>
</html>
```

6.10.4 Comments

- The JSP specification provides two means of including comments in a JSP page one for hidden comments only visible in the JSP page itself and one for comments included in the HTML or XML output generated by the page.
- The **syntax** is as follows:

```
<%- - This is a hidden JSP comment - -%>and the latter looks like this:
<!- - This is included in the generated HTML - ->
```

- Following example shows the JSP Comments:

```
<html>
   <head><title>A Comment Test</title></head>
   <body>
      <h2>A Test of Comments</h2>
      <%-- This comment will not be visible in the page source --%>
   </body>
</html>
```

6.11 JSP DIRECTIVES

- Directives are instructions to the JSP container that describe what code should be generated.
- JSP directives provide directions and instructions to the container, telling it how to handle certain aspects of the JSP processing.
- They have the general **form/syntax**:

```
<%@ directive-name [attribute="value" attribute="value" ...] %>
```

- Zero or more spaces, tabs, and newline characters can be after the opening <%@ and before the ending %>, and one or more whitespace characters can be after the directive name and between attributes/value pairs. The only restriction is that the opening <%@ tag must be in the same physical file as the ending %> tag.
- The JSP specification describes three standard directives available in all compliant.
- JSP environments:
  - page
  - include
  - taglib

- Although the specification declares that no custom directives can be used in the JSP environment, this leaves open the possibility that user-defined directives may be included in a later specification. The next sections provide an overview of each of these directives.

6.11.1 Page Directive

- The page directive is used to specify attributes for the JSP page as a whole. It has the following syntax:

```
<%@ page [attribute="value" attribute="value" ...] %>
```

- Following table lists out the attributes associated with the page directive:

Table 6.9: Attributes of Page Directive

Attribute	Purpose
buffer	Specifies a buffering model for the output stream.
autoFlush	Controls the behavior of the servlet output buffer.
contentType	Defines the character encoding scheme.
errorPage	Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
isErrorPage	Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
extends	Specifies a superclass that the generated servlet must extend.
import	Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
info	Defines a string that can be accessed with the servlet's *getServletInfo()* method.
isThreadSafe	Defines the threading model for the generated servlet.
language	Defines the programming language used in the JSP page.
session	Specifies whether or not the JSP page participates in HTTP sessions
isELIgnored	Specifies whether or not the EL expression within the JSP page will be ignored.
isScriptingEnabled	Determines if the scripting elements are allowed for use.

6.11.2 Include Directive

- The include directive merges the contents of another file at translation time into the .jsp source input stream, much like a #include C preprocessor directive.
- The **syntax** is:

```
<%@ include file="filename" %>
```

where filename is an absolute or relative pathname interpreted according to the current servlet context.

- Examples would be,

```
<%@ include file="/header.html" %>
<%@ include file="/doc/legal/disclaimer.html" %>
<%@ include file="sortmethod" %>
```

- In this example, we are including the content of the header.html file. To run this example you must create an header.html file.

```
<html>
<body>
 <%@ include file="header.html" %>
   Today is: <%= java.util.Calendar.getInstance().getTime() %>
</body>
</html>
```

6.11.3 Taglib Directive

- The taglib directive makes custom actions available in the current page through the use of a tag library. The syntax of the directive is

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

Table 6.10: Attributes of taglib Directive

Attribute	Value
tagLibrary	The URL of a Tag Library Descriptor.
tagPrefix	A unique prefix used to identify custom tags used later in the page.

- 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.

```
<html>
<body>
 <%@ taglib uri="http://www.javatpoint.com/tags" prefix="mytag" %>
 <mytag:currentDate/>
</body>
</html>
```

6.12 MIXING SCRIPLETS AND HTML

- We have already seen how to use the "out" variable to generate HTML output from within a scriptlet. For more complicated HTML, using the out variable all the time loses some of the advantages of JSP programming. It is simpler to mix scriptlets and HTML.
- Suppose you have to generate a table in HTML. This is a common operation, and you may want to generate a table from a SQL table, or from the lines of a file. But to keep our example simple, we will generate a table containing the numbers from 1 to N. Not very useful, but it will show you the technique.

- Here is the JSP fragment to do it:

```
<TABLE BORDER=2>
<%
   for ( int i = 0; i < n; i++ )
   {
      <%
      <TR>
      <TD>Number</TD>
      <TD><%= i+1 %></TD>
      </TR>
      %>
   }
%>
</TABLE>
```

- You would have to supply an int variable "n" before it will work, and then it will output a simple table with "n" rows.
- The important things to notice are how the %> and <% characters appear in the middle of the "for" loop, to let you drop back into HTML and then to come back to the scriptlet.
- Another example of mixing scriptlets and HTML is shown below -- here it is assumed that there is a boolean variable named "hello" available. If you set it to true, you will see one output, if you set it to false, you will see another output.

```
<%
   if ( hello )
   {
      %>
      <P>Hello, world
      <%
   } else
   {
      %>
      <P>Goodbye, world
      <%
   }
%>
```

Example: Program to Generate a table containing the numbers from 1 to N.

- JSP fragment to do it:

```
<TABLE BORDER=2>
<%
for ( int i = 0; i < n; i++ )
{
   %>
   <TR>
      <TD> Number </TD>
      <TD> <%= i+1 %> </TD>
   </TR>
   <%
 }
 %>
 </TABLE>
```

- Any control expressions such as a "while" or a "for" loop or an "if" expression will control the HTML also.
- If the HTML is inside a loop, it will be emitted once for each iteration of the loop.
- Another example of mixing scriptlets and HTML is shown below — here it is assumed that there is a boolean variable named "hello" available. If you set it to true, you will see one output, if you set it to false, you will see another output.

```
<%
   if ( hello )
   {
      %>
      <P>Hello, world
      <%
   } else
   {
      %>
      <P>Goodbye, world
      <%
   }
 %>
```

Summary

- Servlets can maintain state in session variables across many server transactions by using HTTP cookies, or URL rewriting.
- To deploy and run a servlet, a web container must be used. A web container (also known as a servlet container) is essentially the component of a web server that interacts with the servlets.
- The web container is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.
- A Servlet is an object that receives a request and generates a response based on that request.
- Servlet has no static main() method. Therefore, a servlet must execute under the control of an external container.
- Servlet containers, sometimes referred to as servlet engines, execute and manage servlets. The servlet container calls servlet methods and provides services that the servlet needs while executing.
- The servlet container provides the servlet with easy access to properties of the HTTP request, such as its headers and parameters. When a servlet is called, such as when it is specified by URL, the Web server passes the HTTP request to the servlet container.
- Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server.
- Servlets can be created using the javax.servlet and javax.servlet.http packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects.
- A servlet life cycle can be defined as the entire process from its creation till the destruction. Three methods are central to the life cycle of a servlet. These are init( ), service( ), and destroy().
- The servlet is initialized by calling the init() method. The servlet calls service() method to process a client's request. The servlet is terminated by calling the destroy() method.
- The GET method sends the encoded user information appended to the page request. A generally more reliable method of passing information to a backend program is the POST method.
- Servlet API provides two important interfaces javax.servlet.ServletRequest and javax.servlet.http.HttpServletRequest to encapsulate client request.
- The javax.servlet package contains a number of interfaces and classes that establish the framework in which servlets operate.
- The ServletRequest interface enables a servlet to obtain information about a client request.
- The ServletResponse interface enables a servlet to formulate a response for a client.

- The ServletRequest interface includes methods that allow you to read the names and values of parameters that are included in a client request.We will develop a servlet that illustrates their use. The example contains two files. Aweb page is defined in PostParameters.htm, and a servlet is defined in PostParametersServlet.java.
- The service() method is overridden to process client requests. The getParameterNames( ) method returns an enumeration of the parameter names.
- The javax.servlet.http package contains a number of interfaces and classes that are commonly used by servlet developers.
- The HttpServlet class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are doDelete( ), doGet( ), doHead( ), doOptions( ), doPost( ), doPut( ), and doTrace( ).
- Cookies are small pieces of information that are sent in response from the web server to the client. Cookies are the simplest technique used for storing client state.
- Cookies are stored on client's computer. They have a lifespan and are destroyed by the client browser at the end of that lifespan.

Check Your Understanding

1. 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
2. A JSP is transformed into a(n):
   (a) Java applet (b) Java servlet
   (c) Either a or b above (d) Neither a nor b above
3. In Java servlet method init( ) is called _____ times.
   (a) 1 (b) 2
   (c) 0 (d) multiple
4. How constructor can be used for a servlet?
   (a) Initialization
   (b) Constructor function
   (c) Initialization and Constructor function
   (d) Setup() method
5. Can servlet class declare constructor with ServletConfig object as a argument?
   (a) True (b) False
6. Which of the following code is used to get an attribute in a HTTP Session object in servlets?
   (a) session.getAttribute(String name)
   (b) session.alterAttribute(String name)
   (c) session.updateAttribute(String name)
   (d) session.setAttribute(String name)

7. When destroy() method of filter is called?
   (a) The destroy() method is called only once at the end of the life cycle of a filter.
   (b) The destroy() method is called after the filter has executed doFilter method.
   (c) Both of the above.
   (d) The destroyer() method is called after the filter has executed.
8. Which of the following is true about servlets?
   (a) Servlets execute within the address space of web server.
   (b) Servlets are platform-independent because they are written in java.
   (c) Servlets can use the full functionality of the Java class libraries.
   (d) Servlets execute within the address space of web server, platform independent and uses the functionality of java class libraries.
9. How is dynamic interception of requests and responses to transform the information done?
   (a) servlet container (b) servlet config
   (c) servlet context (d) servlet filter

Answers

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

Practice Questions

Q.I Answer the following questions in short:

1. What is Servlet?
2. Differentiate between Get and Post.
3. Explain Generic Servlet and HttpServlet.
4. Differentiate between Session and cookie.
5. What are the different types of scripting elements?
6. What are implicit objects of JSP?

Q.II Answer the following questions:

1. Explain the servlet types.
2. With the help of diagram explain Servlet life cycle.
3. Explain session tracking.
4. Explain types of cookies.
5. Explain HTTPRequest and HTTPResponse.
6. Explain URL rewriting.
7. Explain hidden form field.
8. What are JSP Lifecycle methods?
9. What are advantages of JSP over Servlets?

Q.III Define the term:

1. Session
2. Cookies
3. Servlet
4. URL rewriting
5. JSP Declaration
6. JSP Expression
7. Implicit Object

Previous Exam Questions

Winter 2018

1. What is scriptlet in JSP? **[1 M]**

Ans. Refer to Section 6.10.1.

2. What is a servlet? Explain the lifecycle of servlet in detail. **[5 M]**

Ans. Refer to Section 6.1, 6.2.

3. State the purpose of implicit objects. Explain any four implicit objects. **[5 M]**

Ans. Refer to Section 6.9.

4. Write a servlet program which counts how many times a user has visited a web page (use session). **[4 M]**

Ans. Refer to Program 6.7.

Summer 2019

1. Write any two implicit object in JSP? **[1 M]**

Ans. Refer to Section 6.9.

2. What is servlet? Explain the types of servlet in detail. **[5 M]**

Ans. Refer to Section 6.1.

3. Why is JSP fly compilation? Explain implicit object in JSP? **[5 M]**

Ans. Refer to Section 6.9.

4. Write java program to accept user name, password from HTML and pass to servlet and compare with DB and display appropriate message. **[4 M]**

Ans. Refer to Program 6.11.

■■■

Bibliography

1. https://www.tutorialspoint.com

2. https://www.javatpoint.com

■■■