NEW SYLLABUS
CBCS PATTERN

T.Y. B.B.A. (C.A.)
SEMESTER - V

Core Java

Dr. Ms. Manisha Bharambe
Mrs. Manisha Suryawanshi
Kamil Ajmal Khan

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

SPPU New Syllabus

A Book Of

Core Java

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

[Course Code CA-503 : Credits-03]

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 Garvare 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 ₹ 360.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N4959

CORE JAVA **ISBN 978-93-5451-305-3**

First Edition : September 2021

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

Polyplate

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

DISTRIBUTION CENTRES

PUNE

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

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

MUMBAI

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

DISTRIBUTION BRANCHES

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

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

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

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

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

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

marketing@pragationline.com | www.pragationline.com

Also find us on www.facebook.com/niralibooks

Preface ...

We take an opportunity to present this book entitled as **"Core Java"** to the students of Fifth Semester - B.B.A.(C.A). The object of this book is to present the subject matter in a most concise and simple manner. The book is written strictly according to the New Syllabus (CBCS Pattern).

The book covers theory of Java Fundamentals, Classes, Objects and Methods, Inheritance, Package and Collection, File and Exception Handling, Applet, AWT, Event and Swing Programming.

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

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 Mr. Akbar Shaikh, 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. Java Fundamentals **[Lectures 8]**

- 1.1 Introduction to Java
- 1.2 Basics of Java: Data Types, Variable, Expression, Operators, Constant
- 1.3 Structure of Java Program
- 1.4 Execution Process of Java Program
- 1.5 JDK Tools
- 1.6 Command Line Arguments
- 1.7 Array and String
  - 1.7.1 Single Array and Multidimensional Array
  - 1.7.2 String, String Buffer
- 1.8 Built In Packages and Classes
  - 1.8.1 java.util: Scanner, Date, Math etc.
  - 1.8.2 java.lang

2. Classes, Objects and Methods **[Lectures 8]**

- 2.1 Class and Object
- 2.2 Object Reference
- 2.3 Constructor: Constructor Overloading
- 2.4 Method: Method Overloading, Recursion, Passing and Returning Object Form Method
- 2.5 New Operator, this and static keyword, finalize() Method
- 2.6 Nested Class, Inner Class and Anonymous Inner Class

3. Inheritance, Package and Collection **[Lectures 10]**

- 3.1 Overview of Inheritance
- 3.2 Inheritance in Constructor
- 3.3 Inheriting Data Members and Methods
- 3.4 Multilevel Inheritance - Method Overriding Handle Multilevel Constructors
- 3.5 Use of Super and Final Keyword
- 3.6 Interface
- 3.7 Creation and Implementation of an Interface, Interface Reference
- 3.8 Interface Inheritance
- 3.9 Dynamic Method Dispatch
- 3.10 Abstract Class
- 3.11 Comparison between Abstract Class and Interface
- 3.12 Access Control
- 3.13 Packages
  - 3.13.1 Packages Concept
  - 3.13.2 Creating User Defined Packages
  - 3.13.3 Java Built Inpackages
  - 3.13.4 Import Statement, Static Import

3.14 Collection
- 3.14.1 Collection Framework.
- 3.14.2 Interfaces: Collection, List, Set
- 3.14.3 Navigation: Enumeration, Iterator, ListIterator
- 3.14.4 Classes: LinkedList, ArrayList, Vector, HashSet

4. File and Exception Handling [Lectures 8]

Exception

4.1 Exception and Error
4.2 Use of try, catch, throw, throws and finally
4.3 Built in Exception
4.4 Custom exception
4.5 Throwable Class.

File Handling

4.6 Overview of Different Stream (Byte Stream, Character stream)
4.7 Readers and Writers class
4.8 File Class
4.9 File Input Stream, File Output Stream
4.10 Input Stream Reader and Output Stream Writer

Class

4.11 FileReader and FileWriter Class
4.12 Buffered Reader Class

5. Applet, AWT, Event and Swing Programming [Lectures 14]

Applet

5.1 Introduction
5.2 Types Applet
5.3 Applet Lifecycle
- 5.3.1 Creating Applet
- 5.3.2 Applet Tag

5.4 Applet Classes
- 5.4.1 Color
- 5.4.2 Graphics
- 5.4.3 Font

AWT

5.5 Components and container used in AWT
5.6 Layout Managers
5.7 Listeners and Adapter Classes
5.8 Event Delegation Model

Swing

5.9 Introduction to Swing Component and Container Classes
5.10 Exploring Swing Controls - JLabel and Image Icon, JText Field, The Swing Buttons JButton, JToggle Button, JCheck Box, JRadio Button, JTabbed Pane, JScroll Pane, JList, JTable, JComboBox, Swing Menus, Dialogs, JFileOpen, JColorChooser

■■■

Contents ...

1.	Java Fundamentals	1.1 – 1.50
2.	Classes, Objects and Methods	2.1 – 2.32
3.	Inheritance, Package and Collection	3.1 – 3.72
4.	File and Exception Handling	4.1 – 4.40
5.	Applet, AWT, Event and Swing Programming	5.1 – 5.88
	Bibliography	B.1 – B.1

■■■

1... Java Fundamentals

Learning Objectives...

- To understand Basic Concepts of Java.
- To learn Structure of Java.
- To study JDK Tools.
- To study Command Line Arguments.
- To learn about Array and String.
- To study Built In Packages and Classes.

1.1 INTRODUCING JAVA

- Java is a class-based, object-oriented, 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 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++.
- It is a general-purpose programming language, intended to let application developers write once, run anywhere. Meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM).
- Java enables computer programmers to write computer instructions using English based commands, instead of having to write in numeric codes.
- Like English language, Java has a set of rules (syntax) that determine how the instructions are written. Once, a program has been written, the high-level instructions are translated into numeric codes that computers can understand and execute.

1.1.1 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 recommend 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.

1.1.2 Features of Java [W-18]

- Following are list of buzzwords:
  1. **Simple:** Java is a simple language. There are various concepts that makes the Java as 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 truly 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 multithreaded 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:** 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, Java is an extremely dynamic language, at runtime, the Java environment can extends itself by linking in classes that may be located on remote servers on a network, for example, the Internet.

1.2 BASICS OF JAVA - DATA TYPES, VARIABLE, EXPRESSION, OPERATORS, CONSTANT

1.2.1 Data Types

- A computer has to process various types of data. Data type is a term that refers to the kind of data used in a program. Every programming language comes with a set of data types ready to use.
- Java is a strongly typed language and this means that every variable and every expression has a data type known at compile time.

1. The Primitive Types:

- Various primitive data types are listed below:

(i) byte:

- Byte data type is an 8-bit signed two's complement integer.
- Range is –128 to 127.
- Default value is 0.
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100, byte b = -50

(ii) short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 and maximum value is 32,767.
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int.
- Default value is 0.
- Example: short s = 10000, short r = – 20000

(iii) int:

- int data type is a 32-bit signed two's complement integer.
- Range is - 2,147,483,648.. to 2,147,483,647.
- int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

(iv) long:

- long data type is a 64-bit signed two's complement integer.
- Range is -9,223,372,036,854,775,808... to 9,223,372,036,854,775,807.
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: long a = 100000L, int b = -200000L.

(v) float:

- float data type is a single-precision 32-bit IEEE 754 floating point.
- float is mainly used to save memory in large arrays of floating point numbers.
- default value is 0.0f.
- float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

(vi) double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

(vii) boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

(viii) char:

- char data type is a single 16-bit unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

2. Reference Data Types:

- Reference data types are made by the logical grouping of primitive data types.
- These are called reference data types because they contain the address of a value rather than the value itself.
- Arrays, classes, enum etc., are the example of reference data type.

1.2.2 Variables

- Variables are the identifier of the memory location, which used to save data temporarily for later use in the program. During execution of a program, values can be stored in a variable, and the stored value can be changed.
- While naming Java variables we have following naming rules.
  1. Variable names are case-sensitive.
  2. Variable name can be an unlimited-length sequence of Unicode letters and digits.

3. Variable name must begin with either a letter or the dollar sign "$", or the underscore character "_".
4. No white space is permitted in variable names.
5. Variable name must not be a keyword or reserved word.
6. Do not begin with a digit.
7. A name can be of any realistic length.

Program 1.1: Program for variables.

```
public class AddTwoNumbers
{
   public static void main(String args[])
   {
      int num1 = 50;
      int num2 = 15;
      int sum;
      sum = num1 + num2;
      System.out.println("Sum of Numbers: "+sum);
   }
}
```

Output:

```
Sum of Numbers: 65
```

1.2.3 Expression

- Expressions are essential building blocks of any Java program, usually created to produce a new value, although sometimes an expression assigns a value to a variable.
- An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.
- Expression evaluation in Java is based upon the following concepts:

1. **Type conversion:** It has two types based on how the conversion is performed:
   (i) Implicit conversion: It is also known as automatic conversion. This type of conversion is performed automatically by Java due to performance reasons. Implicit conversion is not performed at all times. For example, Java will automatically convert a value of *byte* into *int* type in expressions since they are both compatible and *int* is larger than *byte* type.
   (ii) Explicit conversion: It is also known as **Type Casting**. There may be situations where you want to convert a value having a type of size less than the destination type size. In such cases Java will not help you. You have do it on your own explicitly. That is why this type of conversion is known as explicit conversion or casting as the programmer does this manually.

Example:

```
int a=10;
byte b = (int) a; //integer value to be converted into a byte type
```

Type promotion rules of Java for expressions are listed below:

- All *char, short* and *byte* values are automatically promoted to *int* type.
- If at least one operand in an expression is a *long* type, then the entire expression will be promoted to *long*.
- If at least one operand in an expression is a *float* type, then the entire expression will be promoted to *float*.
- If at least one operand in an expression is a *double* type, then the entire expression will be promoted to *double*.

2. **Operator Precedence:** Operator precedence determines the order in which the operators in an expression are evaluated.

Table 1.1: Operator Precedence

Operators	Precedence	Associativity
equality	== !=	left to right
bitwise AND	&	left to right
bitwise exclusive OR	^	left to right
bitwise inclusive OR	|	left to right

Program 1.2: Program for operator precedence.

```
class Precedence
{
   public static void main(String[] args)
   {
      int a = 10, b = 5, c = 1, result;
      result = a-++c-++b;
      System.out.println(result);
   }
}
```

Output: 2

- The operator precedence of prefix ++ is higher than that of - subtraction operator.

3. **Associativity of Operators in Java:** If an expression has two operators with similar precedence, the expression is evaluated according to its associativity (either left to right, or right to left).

Example:

(i) d=a+b+c

Since + operator is left associative therefore a+b will solve first and then the result will be added to c.

(ii) d=a^b^c

Since exponent operator is right associative therefore b^c will soved first and then a^ result of b^c .

1.2.4 Operators

- Operators in Java can be classified into 6 types:

1. Unary operators:

+	Unary plus operator; indicates positive value.
–	Unary minus operator; negates an expression.
++	Increment operator; increments a value by 1.
– –	Decrement operator; decrements a value by 1.
!	Logical complement operator; inverts the value of a Boolean.

Program 1.3: Program for unary operator.

```
public class Unary
{
   public static void main(String args[])
   {
      int a=20;
      System.out.println(a++);
      System.out.println(++a);
      System.out.println(a--);
     System.out.println(--a);
   }
}
```

Output:

```
20
22
22
20
```

2. Arithmetic operators:

+	Additive operator (also used for String concatenation)
–	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

Program 1.4: Program for Arithmetic operator.

```
public class Arithmetic
{
   public static void main(String args[])
   {
      int a=20;
      int b=10;
      System.out.println(a+b);
      System.out.println(a-b);
      System.out.println(a*b);
      System.out.println(a/b);
   }
}
```

Output:

```
30
10
200
2
```

3. Bitwise and Bit Shift Operators

~	Unary bitwise complement
<<	Signed left shift
>>	Signed right shift
>>>	Unsigned right shift
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR

Program 1.5: Program for Bitwise and Bit Shift Operator.

```
public class Bitwise
{
   public static void main(String args[])
   {
      int a=20;
      int b=10;
      int c=5;
      System.out.println(20<<2);
      System.out.println(20>>2);
      System.out.println(a<b&a<c);
   }
}
```

Output:

```
80
5
false
```

4. **Logical operator:** The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

logical AND	&&
logical OR	||

Program 1.6: Program for Logical Operator.

```
public class Logical
{
   public static void main(String args[])
   {
      int a=10;
      int b=5;
      int c=20;
      System.out.println(a<b&&a<c);
      System.out.println(a<b||a<c);
   }
}
```

Output:

```
false
true
```

5. **Ternary Operator:** Java ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

   Syntax: variable = (expression)? expressionIsTrue : expressionIsFalse;

Program 1.7: Program for ternary operator.

```
public class Ternary
{
   public static void main(String args[])
   {
      int a=2;
      int b=5;
      int min=(a<b)?a:b;
      System.out.println(min);
   }
}
```

Output:

```
2
```

6. **Equality and Relational Operators:**

==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Program 1.8: Program for equality and relational Operator.

```
public class Euqlity
{
   public static void main(String[] args)
   {
      int x = 10;
      int y = 20;
      if(x== y)
         System.out.println("x == y");
      if(x != y)
         System.out.println("x != y");
      if(x > y)
         System.out.println("x > y");
      if(x < y)
         System.out.println("x < y");
      if(x <= y)
         System.out.println("x <= y");
   }
}
```

Output:

```
x != y
x < y
x <= y
```

7. **Assignment Operator:** Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Table 1.2: Assignment Operator, Usage and Effect

Operator	Usage	Effect
+=	a+=b;	a=a+b;
-=	a-=b;	a=a-b;
*=	a*=b;	a=a*b;
/=	a/=b;	a=a/b;
%=	a%=b;	a=a%b;
&=	a&=b;	a=a&b;
|=	a|=b;	a=a|b;
^=	a^=b;	a=a^b;
<<=	a<<=b;	a=a<<b;
>>=	a>>=b;	a=a>>b;
>>>=	a>>>=b;	a=a>>>b;

Program 1.9: Program for assignment Operator.

```
public class Assignment
{
   public static void main(String[] args)
   {
      int a=10;
      int b=20;
      a+=4;//a=a+4 (a=10+4)
      b-=4;//b=b-4 (b=20-4)
      System.out.println(a);
      System.out.println(b);
   }
}
```

Output:

```
14
16
```

1.2.5 Constant

- A value which is fixed and does not change during the execution of a program is called **Constants** in java. In other words, Java constants are fixed (known as immutable) data values that cannot be changed. Java supports various types of constants.

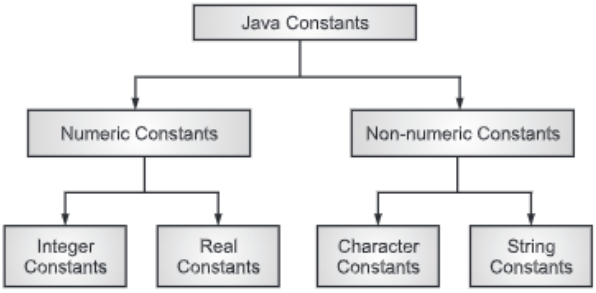


Fig. 1.1: Classification of Java Constants

1. **Integer Constants:**

- An integer constant is a sequence of digits without a decimal point. For example, 10 and -200 are integer constants. There are three types of integer constants. They are as follows:
  - **Decimal integer:** For example: 24, -55, 0
  - **Octal integer:** Example: 0123,024
  - **Hexadecimal integer:** Example: 0x23, ox5B

2. **Real Constants:**

- Real constants consist of a sequence of digits with fractional parts or decimal points. These constants are also called floating-point constants. The valid examples of real constants are 2.3, 0.0034, –0.75, 56.7, etc.

3. **Character Constants:**

- A single character constant (or simply character constant) is a single character enclosed within a pair of single quote. The example of single-character constants are: '5', 'x', ';', ' ', etc

4. **String Constants:**

- A string constant is a sequence of characters within a pair of double-quotes. The characters can be alphabets, special characters, digits, and blank spaces. The valid examples of string constants are: "Hello Java", "1924", "?...!", "2+7", "X", etc.

Backslash Character Constants:

- Java provides some special backslash character constants that are used in output methods. For example, the symbol 'n' which stands for newline character.
- There are various types of backslash character constants that are supported by java. The list is given in the below table.

Table 1.3: Backslash Character Constant

Constants	Meaning
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\''	single quote
'\"'	double quote
\\	backslash

1.2.6 Types of Comments

- Comments are non-executable statements and are ignored by the Java compiler. The comments increases the readability of the programs.
- Comments are an integral part of any program. They help the person reading the code (often you) better understand the intent and functionality of the program.
- Java language provides three styles of comments as given below:

1. **Line Comments:**
   - It start 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 Comments:**
   - Java's multi-line or slash-star or traditional comment is a piece of text enclosed in slash-star (/*) and star-slash (*/).
   - Java's multi-line 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 Comments:**
   - Javadoc comments are used to document the new classes you create as a programmer i.e. provide information about the class for users of the class to read.
   - Java documentation comments is given by /**.....*/.

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

Program 1.10: 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.
   }
}
```

1.3 STRUCTURE OF JAVA PROGRAM

Java Program Structure: [W-18; S-19]

- Java 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.

Documentation Section	<----- Suggested
Package Statements	<----- Optional
Import Statements	<----- Optional
Interface Statements	<----- Optional
Class Definitions	<----- Optional
Main method class	<----- Essential

Fig. 1.2: 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;

3. **Import Statement:** Similar to #include statement in C.

   For example, import employee.test;

   Import statement instructs the interpreter to load the test class contained in package employee. Using import statement; we can access to 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 feature.
5. **Class Definition:** Java program may contain many/multiple class definition. 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.

First Java Program:

```
1. public class MyProgram
2. {
3. public static void main(String[] args)
4. {
5. System.out.print("My First Java Program");
6. }
7. }
```

Output: My First Java Program

Explanation:

- The basic unit of a Java program is a class. A class called "`MyProgram`" is defined via the keyword "`class`" in Lines 2–7. The braces {.....} encloses the body of the class.

```
public class MyProgram
{
   ......
}
```

- In Java, the name of the source file must be the same as the name of the class with a mandatory file extension of ".`java`". Hence, this file must be saved as "`MyProgram.java`", case-sensitive.

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

- Lines 3-6 defines the so-called `main()` method, which is the entry point for program execution. Again, the braces {.....} encloses the body of the method, which contains programming statements.
- In Line 5, the programming statement `System.out.println("My First Java Program")` is used to print the string "`My First Java Program`" to the display console. A string is surrounded by a pair of double quotes and contain texts. The text will be printed as it is, without the double quotes. A programming statement ends with a semi-colon (;).

1.4 EXECUTION PROCESS OF JAVA PROGRAM

- **Example:**

```
public class MyFirstJavaProgram
{
   /* This is my first java program.
   * This will print 'Hello World' as the output
   */
   public static void main(String []args)
   {
      System.out.println("Hello World"); // prints Hello World
   }
}
```

- Let's look at how to compile, and run the program.

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 MyFirstJavaProgram.java
```

- Javac compiler creates a file named '`MyFirstJavaProgram.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 MyFirstJavaProgram
```

Output:

```
C:\> javac MyFirsthelloProgram.java
C:\> java MyFirsthelloProgram
Hello World
```

1.5 JDK TOOLS

- The JDK is a software development environment which is used to develop java applications and applets.
- JDK (Java Development Kit) provides environment to develop and run java applications.
- Java Development Kit has a collection of tools for Java Development. JDK contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools. JRE is part of JDK but can be used independently to run any byte code (compiled java program). It can be called as JVM implementation.
- The JDK comes with a set of tools that are used for developing and running Java program. It includes:
  1. **Javac:** It is a Java compiler.
  2. **Java:** It is a java interpreter.
  3. **Javah:** It is for java C header files.
  4. **Jdb:** It is Java debugger.
  5. **AppletViewer:** It is used for viewing the applet.
  6. **Javap:** Java disassembler, which convert byte code into program description.
  7. **Javadoc:** It is used for creating HTML document.

1.6 COMMAND LINE ARGUMENTS

- 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
```

Example: The following Program illustrate that how to take input as two numbers from the user and display the addition with two input methods: BufferedReader and Command Line arguments.

```
import java.io.*;
import java.util.Scanner;
class demo
{
   public static void main(String arg[]) throws Exception
   {
      System.out.println("addition of two numbers using three input
      methods");
      //input using BufferedReader
      int x,y;
      BufferedReader b=new BufferedReader(new InputStreamReader
      (System.in));
      System.out.println("enter two numbers");
      x=Integer.parseInt(b.readLine());
      y=Integer.parseInt(b.readLine());
      System.out.println("sum is=" +(x+y));
      // Input using Command line arguments
      int a=Integer.parseInt(arg[0]);
      int d=Integer.parseInt(arg[1]);
      int sum1=0;
      sum1=a+d;
      System.out.println(sum1);
   }
}
```

- When we accepting input from console we need to BufferedReader and scanner class.
- 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.
- 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.

1.7 ARRAY AND STRING

1.7.1 Array [W-18]

- 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.

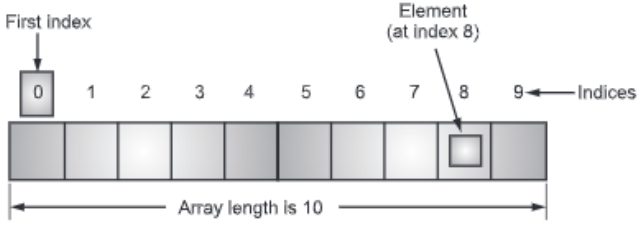


Fig. 1.3: Arrays

1.7.1.1 Single-Dimensional (1D) Array

- An array is a sequence of variables of the same data type. 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, type 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 1.11: 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 1.12: 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]);
            }
        }
    }
}
```

Output:

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

1.7.1.2 Multi-Dimensional (2D) Array

- In Java, multi-dimensional arrays are actually arrays of arrays. These arrays look and act like regular multi-dimensional arrays.
- 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];
```

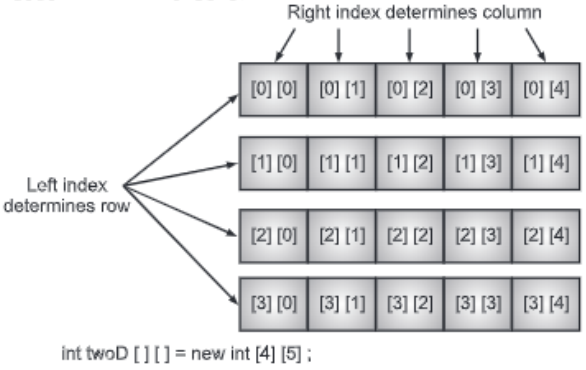


Fig. 1.4: 2D Array Declaration

- 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 [exp1][exp2];
```

Where, exp1 and exp2 are expressions that evaluate to integers and specify, respectively, the number of rows and the number of columns in the array.

- 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 1.13: 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 1.14: 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]);
         }
      }
      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]);
           }
           System.out.println();
        }
     }
  }
```

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 1.15: 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 1.16: 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 1.17: 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 1.18: 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 1.19: 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
```

1.7.2 String

1.7.2.1 String (Basic Functions)

- 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, and
  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. '+' & '+=" 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 = "Kal" + "pa" + "na";`

  Output: Kalpana
- In the above example, a method append() is called which creates the new String object having "Kal" 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.

String Class Methods:

- 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 1.4: String Class Methods

Sr. No.	Methods	Description
1.	`char charAt(int index)`	This method returns char at (value for) the particular index.
2.	`int length()`	This method returns string length.
3.	`static String format(String format, Object... args)`	This method returns formatted string.
4.	`static String format(Locale l, String format, Object... args)`	This method returns formatted string with given locale.

contd. ...

5.	`String substring(int beginIndex)`	This method returns substring for given begin index.
6.	`String substring(int beginIndex, int endIndex)`	This method returns substring for given begin index and end index.
7.	`boolean contains(CharSequence s)`	This method returns true or false after matching the sequence of char value.
8.	`static String join(CharSequence delimiter, CharSequence... elements)`	This method returns a joined string with delimiter.
9.	`static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)`	This method returns a joined string.
10.	`boolean equals(Object another)`	This method checks the equality of string with object.
11.	`boolean isEmpty()`	This method checks if string is empty.
12.	`String concat(String str)`	This method concatenates specified string.
13.	`String replace(char old, char new)`	This method replaces all occurrences of specified char value.
14.	`String replace(CharSequence old, CharSequence new)`	This method replaces all occurrences of specified CharSequence.
15.	`String trim()`	This method returns trimmed string omitted leading and trailing spaces.
16.	`String split(String regex)`	This method returns splitted string matching regex.
17.	`String split(String regex, int limit)`	This method returns splitted string matching regex and limit.
18.	`String intern()`	This method returns interned string.
19.	`int indexOf(int ch)`	This method returns specified char value index.
20.	`int indexOf(int ch, int fromIndex)`	This method returns specified char value index starting with given index.
21.	`int indexOf(String substring)`	This method returns specified substring index.

contd. ...

22.	`int indexOf(String substring, int fromIndex)`	This method returns specified substring index starting with given index.
23	`String toLowerCase()`	This method returns string in lowercase.
24.	`String toLowerCase(Locale l)`	This method returns string in lowercase using specified locale.
25.	`String toUpperCase()`	This method returns string in uppercase.
26.	`String toUpperCase(Locale l)`	This method returns string in uppercase using specified locale.

String Class Constructors:

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

Table 1.5: String Class Constructor

Sr. No.	Constructor	Description
1.	`String()`	This constructor creates an empty string.
2.	`String(String value)`	This constructor creates a new string that is a copy of the given string.
3.	`String(char[] value)`	This constructor constructs a new string based on the character array.
4.	`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.
5.	`String(byte[] value)`	This constructor creates a new string by converting the given array of bytes.
6.	`String(byte[] value, int offset, int length)`	This constructor creates a new string by converting the given sub of array of bytes.
7.	`String(StringBuffer buffer)`	This constructor creates a new string based on a StringBuffer value.
8.	`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.
9.	`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.

1.7.2.2 StringBuffer

- 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.

- The StringBuffer defines three types of constructors which we use to initialize the string value.
  1. StringBuffer()
  2. StringBuffer (int size)
  3. StringBuffer (String str)
- The first type is the default constructor as it has no parameters. It reserves room for 16 characters without reallocation. The second, type accepts an integer argument which explicitly sets the size of buffer. In third case, 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.
- The current length of a StringBuffer can be found via length() method and the total allocated capacity can be found through capacity() method.
- The general forms are:

```
int length();
```

  and

```
int capacity();
```

- 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 startIndex, int endIndex)
```

- The startIndex specifies the beginning index and endIndex tells us about stopping point.
- **StringBuffer Class Methods:** Here, is the list of important methods supported by StringBuffer class:
  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 1.20: 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: [W-18]

- 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.

1.8 BUILT IN PACKAGES AND CLASSES

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.

1.8.1 java.util:- Scanner, Date, Math

- Java.util Package contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
- Following are the Important Classes in Java.util package:

1. Scanner:

- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings. It is the easiest way to read input in a Java program though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.
- **Methods of scanner class**:
  1. `nextBoolean()` : Reads a boolean value from the user.
  2. `nextByte()` : Reads a byte value from the user.
  3. `nextDouble()` : Reads a double value from the user.
  4. `nextFloat()` : Reads a float value from the user.
  5. `nextInt()` : Reads a int value from the user.
  6. `nextLine()` : Reads a String value from the user.
  7. `nextLong()` : Reads a long value from the user.
  8. `nextShort()` : Reads a short value from the user.
- To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the `NextInt()` method, which is used to read integer value from user:

Program 1.21: The following Program illustrate that how to take input as two numbers from the user and display the addition with three input methods: Scanner.

```
import java.io.*;
import java.util.Scanner;
public class demo
{
   public static void main(String arg[]) throws Exception
   {
      Scanner s=new Scanner(System.in);
```

```
        System.out.println("enter two numbers");
        int l=s.nextInt();
        int n=s.nextInt();
        int sum=0;
        sum=l+n;
        System.out.println("sum is="+sum);
    }
}
```

Output:

```
enter two numbers
5
5
sum is=10
```

2. Date:

- Java does not have a built-in Date class, but we can import the java.time package to work with the date and time API. The package includes many date and time classes. For example:
- Date and Time classes:
  1. `LocalDate` : Represents a date (year, month, day (yyyy-MM-dd))
  2. `LocalTime` : Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns))
  3. `LocalDateTime` : Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns)
  4. `DateTimeFormatter` : Formatter for displaying and parsing date-time objects

Program 1.22: To display current date and time with formatting.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class Main
{
    public static void main(String[] args)
    {
        LocalDateTime myDateObj = LocalDateTime.now();
        System.out.println("Before formatting: " + myDateObj);
        DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-
        yyyy HH:mm:ss");
        String formattedDate = myDateObj.format(myFormatObj);
        System.out.println("After formatting: " + formattedDate);
    }
}
```

Output:

```
Before formatting: 2021-08-03T07:40:54.347413
After formatting: 03-08-2021 07:40:54
```

- The "T" in the example above is used to separate the date from the time. You can use the `DateTimeFormatter` class with the of `Pattern()` method in the same package to format or parse date-time objects.

3. Math:

- java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.
- Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.
- If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.
- For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Math Methods:

- The **java.lang.Math** class contains various methods for performing basic numeric operations such as the logarithm, cube root, and trigonometric functions etc. The various java math methods are as follows:

Table 1.6: Basic Math methods

Method	Description
`Math.abs()`	It will return the Absolute value of the given value.
`Math.max()`	It returns the Largest of two values.
`Math.min()`	It is used to return the Smallest of two values.
`Math.round()`	It is used to round of the decimal numbers to the nearest value.
`Math.sqrt()`	It is used to return the square root of a number.
`Math.cbrt()`	It is used to return the cube root of a number.
`Math.pow()`	It returns the value of first argument raised to the power to second argument.
`Math.signum()`	It is used to find the sign of a given value.
`Math.ceil()`	It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer.

contd. ...

`Math.copySign()`	It is used to find the Absolute value of first argument along with sign specified in second argument.
`Math.nextAfter()`	It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.
`Math.nextUp()`	It returns the floating-point value adjacent to d in the direction of positive infinity.
`Math.nextDown()`	It returns the floating-point value adjacent to d in the direction of negative infinity.
`Math.floor()`	It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value.
`Math.floorDiv()`	It is used to find the largest integer value that is less than or equal to the algebraic quotient.
`Math.random()`	It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
`Math.rint()`	It returns the double value that is closest to the given argument and equal to mathematical integer.
`Math.hypot()`	It returns sqrt(x^2 +y^2) without intermediate overflow or underflow.
`Math.ulp()`	It returns the size of an ulp of the argument.
`Math.getExponent()`	It is used to return the unbiased exponent used in the representation of a value.
`Math.IEEEremainder()`	It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value.
`Math.addExact()`	It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long.
`Math.subtractExact()`	It returns the difference of the arguments, throwing an exception if the result overflows an int.
`Math.multiplyExact()`	It is used to return the product of the arguments, throwing an exception if the result overflows an int or long.
`Math.incrementExact()`	It returns the argument incremented by one, throwing an exception if the result overflows an int.
`Math.decrementExact()`	It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long.
`Math.negateExact()`	It is used to return the negation of the argument, throwing an exception if the result overflows an int or long.
`Math.toIntExact()`	It returns the value of the long argument, throwing an exception if the value overflows an int.

Program 1.23: Pragram for math class.

```
public class JavaMathExample1
{
    public static void main(String[] args)
    {
        double x = 28;
        double y = 4;
        System.out.println("Maximum number of x and y is:" +Math.max(x, y));
        System.out.println("Square root of y is: " + Math.sqrt(y));
        System.out.println("Power of x and y is: " + Math.pow(x, y));
        System.out.println("Logarithm of x is: " + Math.log(x));
        System.out.println("Logarithm of y is: " + Math.log(y));
        System.out.println("log10 of x is: " + Math.log10(x));
        System.out.println("log10 of y is: " + Math.log10(y));
        System.out.println("log1p of x is: " +Math.log1p(x));
        System.out.println("exp of a is: " +Math.exp(x));
       System.out.println("expm1 of a is: " +Math.expm1(x));
    }
}
```

Output:

```
Maximum number of x and y is: 28.0
Square root of y is: 2.0
Power of x and y is: 614656.0
Logarithm of x is: 3.332204510175204
Logarithm of y is: 1.3862943611198906
log10 of x is: 1.4471580313422192
log10 of y is: 0.6020599913279624
log1p of x is: 3.367295829986474
exp of a is: 1.446257064291475E12
expm1 of a is: 1.446257064290475E12
```

1.8.2 java.lang

- Provides classes that are fundamental to the design of the Java programming language. The most important classes are object, which is the root of the class hierarchy, and class, instances of which represent classes at run time.

- **Following are the Important Classes in Java.lang package:**
  1. **`Boolean`:** The Boolean class wraps a value of the primitive type boolean in an object.
  2. **`Byte`:** The Byte class wraps a value of primitive type byte in an object.
  3. **`Character – Set 1, Set 2`:** The Character class wraps a value of the primitive type char in an object.
  4. **`Character.Subset`:** Instances of this class represent particular subsets of the Unicode character set.
  5. **`Character.UnicodeBlock`:** A family of character subsets representing the character blocks in the Unicode specification.
  6. **`Class – Set 1, Set 2`:** Instances of the class Class represent classes and interfaces in a running Java application.
  7. **`ClassLoader`:** A class loader is an object that is responsible for loading classes.
  8. **`ClassValue`:** Lazily associate a computed value with (potentially) every type.
  9. **`Compiler`:** The Compiler class is provided to support Java-to-native-code compilers and related services.
  10. **`Double`:** The Double class wraps a value of the primitive type double in an object.
  11. **`Enum`:** This is the common base class of all Java language enumeration types.
  12. **`Float`:** The Float class wraps a value of primitive type float in an object.
  13. **`InheritableThreadLocal`:** This class extends Thread Local to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values.
  14. **`Integer`:** The Integer class wraps a value of the primitive type int in an object.
  15. **`Long`:** The Long class wraps a value of the primitive type long in an object.
  16. **`Math – Set 1, Set 2`:** The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
  17. **`Number`:** The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.
  18. **`Object`:** Class Object is the root of the class hierarchy.
  19. **`Package`:** Package objects contain version information about the implementation and specification of a Java package.
  20. **`Process`:** The ProcessBuilder.start() and Runtime.exec methods create a native process and return an instance of a subclass of Process that can be used to control the process and obtain information about it.
  21. **`ProcessBuilder`:** This class is used to create operating system processes.
  22. **`ProcessBuilder.Redirect`:** Represents a source of subprocess input or a destination of subprocess output.

23. **`Runtime`**: Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running.
24. **`RuntimePermission`:** This class is for runtime permissions.
25. **`SecurityManager`:** The security manager is a class that allows applications to implement a security policy.
26. **`Short`**: The Short class wraps a value of primitive type short in an object.
27. **`StackTraceElement`**: An element in a stack trace, as returned by Throwable.getStackTrace().
28. **`StrictMath - Set1, Set2`:** The class StrictMath contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
29. **`String - Set1, Set2`:** The String class represents character strings.
30. **`StringBuffer`:** A thread-safe, mutable sequence of characters.
31. **`StringBuilder`:** A mutable sequence of characters.
32. **`System`**: The System class contains several useful class fields and methods.
33. **`Thread`**: A thread is a thread of execution in a program.
34. **`ThreadGroup`**: A thread group represents a set of threads.
35. **`ThreadLocal`**: This class provides thread-local variables.
36. **`Throwable`:** The Throwable class is the superclass of all errors and exceptions in the Java language.
37. **`Void`**: The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the Java keyword void.

Program 1.24: Write a Java program to accept 'n' numbers through command line and store all prime numbers into an array and display elements of array. **[W-18]**

```
import java.io.*;
public class ArrayDemo2
{
   public static void main (String args[]) throws IOException
   {
      int j,s,i,k=0,m=0,n;
      int r[]=new int[10] ;
      int prime[]=new int[10] ;
      int perfect[]=new int[10] ;
      BufferedReader    br=new    BufferedReader(new    InputStreamReader
      (System.in));
      System.out.println("Enter the Number of Elements :=>");
      n=Integer.parseInt(br.readLine());
```

```
        for (i=0;i<n;i++)
        {
           System.out.println("Enter the "+i+" Number of the Array :=> ");
           r[i]= Integer.parseInt(br.readLine());
        }
        for(j=0;j<10;j++)
        {
           int flag=0;
           if(r[j]==0||r[j]==1)
           {
                  continue;
           }
           else
           {
                  for(i=2;i<r[j];i++)
                  {
                         if(r[j]%i==0)
                         flag=1;
                  }
           }
           if(flag==0)
           {
                  prime[k]=r[j];
                  k++;
           }
        }
        System.out.println("\nPrime Numbers are");
        for(i=0;i<k;i++)
        {
           System.out.print(prime[i]+" ");
        }
     }
  }
```

Output:

```
Enter the Number of Elements :=>
3
Enter the 0 Number of the Array :=>
3
```

```
Enter the 1 Number of the Array :=>
2
Enter the 2 Number of the Array :=>
4
Prime Numbers are
3
2
```

Program 1.25: Java program to count vowels, consonants, digits, and spaces of given word. [W-18]

```
public class Main
{
  public static void main(String[] args)
  {
    String line = "This website is aw3som3.";
    int vowels = 0, consonants = 0, digits = 0, spaces = 0;
    line = line.toLowerCase();
    for (int i = 0; i < line.length(); ++i)
    {
      char ch = line.charAt(i);
      // check if character is any of a, e, i, o, u
      if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u')
      {
        ++vowels;
      }
      // check if character is in between a to z
      else if ((ch >= 'a' && ch <= 'z'))
      {
        ++consonants;
      }
      // check if character is in between 0 to 9
      else if (ch >= '0' && ch <= '9')
      {
        ++digits;
      }
      // check if character is a white space
      else if (ch == ' ')
      {
        ++spaces;
      }
    }
```

```
        System.out.println("Vowels: " + vowels);
        System.out.println("Consonants: " + consonants);
        System.out.println("Digits: " + digits);
        System.out.println("White spaces: " + spaces);
      }
   }
```

Output:

```
Vowels: 7
Consonants: 11
Digits: 2
White spaces: 3
```

Program 1.26: Write a java program to accept n names of cities from users and display them in descending order. **[S-19]**

```
import java.util.*;
class City
{
   //declaration of array
   String a[];
   int n;
   City()
   {
      Scanner s=new Scanner(System.in);
      System.out.print("Enter how many city you want to enter : ");
      n=s.nextInt();
      //redeclartion of array
      a=new String[n];
      //To accept values
      for(int i=0;i<n;i++)
      {
         System.out.print("Enter {i+1} element: ");
         a[i]=s.next();
      }
   }
   //To display values
   void display()
   {
      String temp="";
```

```
        for(int i=0;i<n;i++)
        {
           for(int j=i+1;j<n;j++)
           {
           if(a[i].compareTo(a[j])>0)
           {
              temp=a[i];
              a[i]=a[j];
              a[j]=temp;
           }
           }
        }
        System.out.println("Sorted Cities are ");
        for(int i=0;i<n;i++)
        {
           System.out.print(a[i]+" ");
        }
     }
  }
  //To create object
  public class TestCity
  {
     public static void main(String args[])
     {
        City obj=new City();
        obj.display();
     }
  }
```

Output:

```
Enter how many city you want to enter : 5
Enter {i+1} element: Pune
Enter {i+1} element: Satara
Enter {i+1} element: Kolhapur
Enter {i+1} element: Nagpur
Enter {i+1} element: Pimpri
Sorted Cities are
Kolhapur Nagpur Pimpri Pune Satara
```

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 defines eight primitive types of data: byte, short, int, long, char, float, double, and Boolean.
- 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.
- Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.
- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings.
- A package is a collection of classes and interfaces.

Practice Questions

1. 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
2. Which of these classes is not included in java.lang?
   (a) Byte (b) Integer
   (c) Array (d) Class
3. Which of these is a super class of wrappers Double & Integer?
   (a) Long (b) Digits
   (c) Float (d) Number
4. Which of the following package stores all the simple data types in java?
   (a) lang (b) java
   (c) util (d) java.packages
5. 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;

6. Which Scanner class method is used to read integer value from the user?
   (a) next() (b) nextInteger()
   (c) nextInt() (d) readInt()
7. What is the output of this program?

```
class increment
{
   public static void main(String args[])
   {
       int g = 3;
       System.out.print(++g * 8);
   }
}
```

   (a) 25 (b) 24
   (c) 32 (d) 33
8. Which is the correct syntax to declare Scanner class object?
   (a) Scanner objectName= Scanner();
   (b) Scanner objectName= new Scanner();
   (c) Scanner objectName= Scanner(System.in);
   (d) Scanner objectName= new Scanner(System.in);
9. How to format date from one form to another?
   (a) SimpleDateFormat (b) DateFormat
   (c) SimpleFormat (d) DateConverter
10. How to format date from one form to another?
    (a) SimpleDateFormat (b) DateFormat
    (c) SimpleFormat (d) DateConverter

Answers

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

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 is called portable?
5. How to compile Java program?
6. How to run Java program?

Q.II Answer the following Questions.

1. Explain atleast five features of Java.
2. Why Java needs compiler and interpreter?
3. What is a variable? What rules can be applied to have a valid variable name?
4. Explain primitive data types in detail.
5. Explain reference or non-primitive data types.
6. What is array?
7. How a comment is added in java program?
8. What are the types of array?
9. Explain java.lang package.
10. Explain java.util package.
11. Explain scanner, math, date class in java.
12. Predict the output:

```
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. Array
2. String
3. Package
4. Variable
5. Constant
6. Type casting

Previous Exam Questions

Winter 2018

1. Describe any two features of Java programming language. **[2 M]**

Ans. Refer to Section 1.1.2.

2. What are differences between String and StringBuffer class? **[2 M]**

Ans. Refer to Section 1.7.2.2.

3. "Import statement is not essential in java." True/False. Justify. **[2 M]**

Ans. Refer to Section 1.3.

4. Explain Array in Java. How does it differ from C++? **[4 M]**

Ans. Refer to Section 1.7.1.

5. Write a Java program to accept 'n' numbers through command line and store all prime numbers into an array and display elements of array. **[4 M]**

Ans. Refer to Program 1.24.

6. Write a Java Program that displays the number of non-vowels in the given word. **[4 M]**

Ans. Refer to Program 1.25.

Summer 2019

1. Does the order of public static void matter in main method. **[2 M]**

Ans. Refer to Section 1.3.

2. Write a java program to accept n names of cities from users and display them in descending order. **[4 M]**

Ans. Refer to Program 1.26.

■■■

2... Classes, Objects and Methods

Learning Objectives...

- To understand Basic Concepts of Objects and Classes.
- To study Constructors and Methods.
- To learn Nested Classes, Inner Class, Anonymous Inner Class in Java.
- To learn New Operator, this and static keyword, finalize() method.

2.1 CLASS AND OBJECT [S-19]

- 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.
- Java is not fully object oriented programming language because it makes use of eight data types such as boolean, int, char, double, long, short, float and byte which are not object.

2.1.1 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 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.

- 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;
}
```

Program 2.1: 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
```

2.1.2 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. 2.1 shows a real world example of class and objects. Fig. 2.1 shows class Car has objects like Audi, Nissan, Volvo.

Fig. 2.1: 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:** `<classname> objectname;`

  Where, *classname* is the name of the already defined class and the *objectname* 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 Box class.

```
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
```

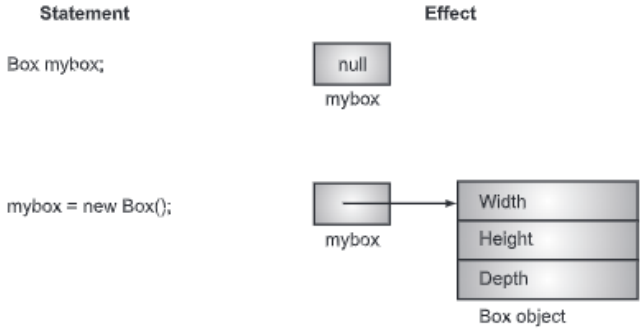


Fig. 2.2: Object Instantitation

(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,

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

Program 2.2: Program for declaration and instantiation of object.

```
class MyObject
{
   public void getMessage()
   {
      System.out.println("Object Created and Functioned Called");
   }
   public static void main(String args[])
   {
      MyObject x = new MyObject(); //x is declared and initialized with an
      instance of MyObject
      x.getMessage();
   }
}
```

Output:

```
Object Created and Functioned Called
```

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 2.1: 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{ ... }`	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.

2.2 OBJECT REFERENCE

- The objects are constructed and allocated in memory independently from the declarations of variables. Specifically:
  1. Objects denoted by a literal (such as literals of type String, e.g., "foo", "ciao", etc.) are allocated in memory at compilation time.
  2. All other objects must be constructed and allocated through an explicit statement.
- A variable whose type is a class contains a reference to an object of the class (i.e., the address of the memory location where the object is allocated).
- A variable is a reference to a memory location in which an object is stored. To represent variables and their values we use the following graphical notation.

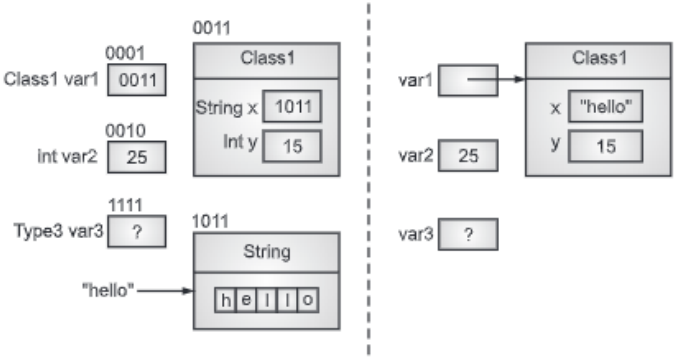


Fig. 2.3: Reference to object of variable

2.3 CONSTRUCTOR

- 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 2.3: 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
```

2.3.1 Types of Constructor

- Constructors can be classified into two types, Default Constructors and Parameterized Constructors as shown in Fig. 2.4.

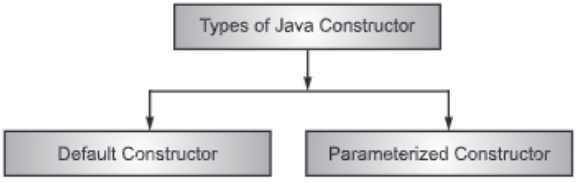


Fig. 2.4: Types of Constructor

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:** `ConstructorName(){}`

Program 2.4: 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 1
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 2.5: 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
```

2.3.2 Constructor Overloading

- Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.
- The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.
- Constructors having the same name with different parameter list is called as Constructor overloading.

Program 2.6: 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
```

2.4 METHOD

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.
- The only required elements of a method declaration are the method's return type, name, a pair of parentheses (), and a body between braces {}.
- More generally, method declarations have following components, in order:
  1. **Modifiers:** Such as public, private, default, protected.
  2. **The return type:** The data type of the value returned by the method, or void if the method does not return a value.
  3. **The method name:** The rules for field names apply to method names as well, but the convention is a little different.
  4. **The parameter list in parenthesis:** A comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses (). If there are no parameters, you must use empty parentheses.
  5. **The method body, enclosed between braces:** The method's code, including the declaration of local variables, goes here.

     Syntax:

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

Program 2.7: Program for the method.

```
public class Main
{
    public String getName(String st)
    {
        System.out.println("Kamil Ajmal Khan");
    }
    public static void main(String[] args)
    {
        Main Obj=new Main();
        Obj.getName();
    }
}
```

Output:

```
Kamil Ajmal Khan
```

2.4.1 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 2.8: 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 2.2: Difference between Constructor and Method

Sr. No.	Constructor	Method
1.	Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
2.	Constructor name must be same as the class name.	Method name may or may not be same as class name.
3.	Constructor is invoked implicitly.	Method is invoked explicitly.
4.	The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
5.	Constructor must not have return type.	Method must have return type.

2.4.2 Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

How a particular problem is solved using recursion?

- The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

What is the difference between direct and indirect recursion?

- A function `fun` is called direct recursive if it calls the same function `fun`. A function `fun` is called indirect recursive if it calls another function say `fun_new` and `fun_new` calls `fun` directly or indirectly.

  Direct recursion:

```
void directRecFun()
{
    // Some code....
    directRecFun();
    // Some code...
}
```

  Indirect recursion:

```
void indirectRecFun1()
{
    // Some code...
    indirectRecFun2();
    // Some code...
}
void indirectRecFun2()
{
    // Some code...
    indirectRecFun1();
    // Some code...
}
```

Program 2.9: Write java program to find factorial of given number.

```
class Factorial
{
   static int factorial(int n)
   {
      if (n == 0)
         return 1;
      else
         return(n * factorial(n-1));
   }
```

```
    public static void main(String args[])
    {
        int i, fact=1;
        int number=4;//It is the number to calculate factorial
        fact = factorial(number);
        System.out.println("Factorial of "+number+" is: "+fact);
    }
}
```

Output:

```
Factorial of 4 is: 24
```

2.4.3 Passing and Returning Object form Method

- When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference.
- Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.
- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.

Program 2.10: Program to passing & returning object as parameter in method.

```
class ObjectPassDemo
{
    int a, b;
    ObjectPassDemo(int i, int j)
    {
        a = i;
        b = j;
    }
    boolean equalTo(ObjectPassDemo o)
    {
        return (o.a == a && o.b == b);
    }
}
```

```
public class Test
{
    public static void main (String args[])
    {
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

Output:

```
ob1 == ob2: true
ob1 == ob3: false
```

2.5 new OPERATOR, this and static KEYWORD, finalize() METHOD

2.5.1 'new' Operator

- The new operator is used in Java to create new objects. It can also be used to create an array object.
- In other words, it instantiates a class by allocating memory for a new object and returning a reference to that memory. We can also use the new keyword to create the array object.
- **Syntax:**

```
NewExample obj = new NewExample();
```

Program 2.11: Program for new operator.

```
public class Main
{
   public static void main(String[] args)
   {
      double[] myList = new double[] {1.9, 2.9, 3.4, 3.5};
      for (int i = 0; i < myList.length; i++)
      {
         System.out.println(myList[i] + " ");
      }
      double total = 0;
      for (int i = 0; i < myList.length; i++)
      {
         total += myList[i];
      }
```

```
        System.out.println("Total is: " + total);
        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
```

2.5.2 '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.
- The 'this' is used inside the method or constructor to refer its own object i.e., 'this' is always a reference to the object of the current class' type.
- **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. 2.5 shows use of 'this' keyword.

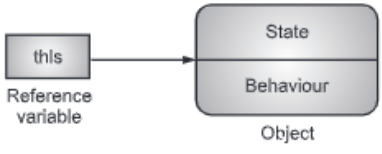


Fig. 2.5: this keyword

- **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 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 2.12: 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
```

2.5.3 'static' Keyword

- The static keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.
- Static variables are also known as class variables. Local variables cannot be declared static.

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 when the first time the class is loaded into memory.

Program 2.13: Program 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
```

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 2.14: Java program to use of static 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 2.15: Program using static method which maintain 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("Customerl 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: 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). The method *modifyRate()* is made static in the class as we are only using static field rate in it. Only static method can access a static field.

2.5.4 finalize() Method [W-18]

- 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 2.16: 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.
```

2.6 NESTED CLASS, INNER CLASS, AND ANONYMOUS INNER CLASS

2.6.1 Nested Class

- It is possible to put the definition of one class inside the definition of another class. The class which is defined inside is called nested class.
- Inner class is a part of nested class. Non-static nested classes are known as inner classes.

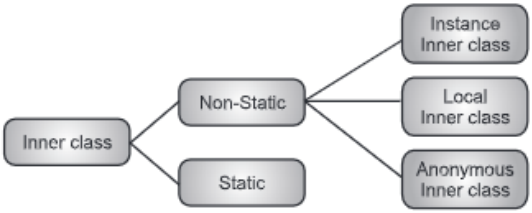


Fig. 2.6: Types of Nested Classes

Types of Nested Classes:

- There are two types of nested classes, Non-static and Static nested classes. The Non-static nested classes are also known as Inner Classes.

(i) Non-static Nested Class:

- The nested class for which static modifier is not applied, such a class is known as non-static nested class.
- This class can also be refer as inner class, because it is fully within the scope of its enclosing class.
- It has access to all the variables and methods of its outer class or enclosing class.
- The general **form/syntax** of non-static definition is as follows:

```
public class OutClass
{
   //Members of outclass
   public class InClass //Nested class
   {
      //Members of inclass
   }
   //More members of outclass
}
```

- You can't create any object of *InClass*, without creating any object of *OutClass*. For example, an object of Outclass can be created as:

```
OutClass Outobj = new OutClass();
```

- After creating an object of *OutClass*, an object of *InClass* will not be automatically created. To refer a nested class type, we must have to use the name of enclosing class as a qualifier.
- For example, to create an object of *InClass*:

```
OutClass.InClass InObj = OutObj.new InClass();
```

- That means an object of the nested class is created in the context of an object of enclosing class.
- But non-static methods that are members of enclosing class can refer the nested class without any qualifier.
- The main purpose of nested classes is for grouping a set of related classed under the single enclosing class.

- Following program, illustrates the creation of an object and calling method of inner class inside outer class. It also shows, how to create and access an object of inner class outside the outer class.

Program 2.17: Program for Non-static nested class.

```
class OutClass
{
   int a = 5;
   void outmethod()
   {
      //creation of an object of inner class
      InClass InObj = new InClass();
      InObj.InMethod();
   }
   class InClass
   {
      int b = 10;
      void Inmethod()
      {
         //Member of outerclass can refer here
         System.out.println("a of OutClass =" + a);
         System.out.println("b of InClass =" + b);
      }
   } //end of inclass
}//end of outclass
Class TestNested1
{
   public static void main(String args[])
   {
      //create object of outclass
      Outclass Outobj=new Outclass();
      //Call to method of outclass
      OutObj.Outmethod();
      //Creation of InnerClass Object
      OutClass.InClass InObj = OutObj.new InClass();
      //Calling a method of inner class
      InObj.Inmethod();
   }
}
```

Output:

```
a  of OutClass = 5
b  of InClass = 10
a  of OutClass = 5
b  of Inclass = 10
```

(ii) Static Nested Class:

- A class which is defined as a static inside other enclosing class, such a class is called as Static Nested Class.
- This type of nested class, cannot refer any member of its enclosing class directly, that means it must access members of enclosing class through an object. This is shown in following program.

Program 2.18: Program for Static Nested Class.

```
class Outclass
{
   int a = 5;
   public static class InClass
   {
      int b = 10;
      void Inmethod()
      {
         //Member of enclosing class
         //Should refer through its object
         OutClass.OutObj=new OutClass();
         System.out.println("a of Outclass=" + OutObj.a);
         System.out.println("b of Inclass=" + b);
      }
   }//end of Inclass
}//end of Outclass
class TestNested 2
{
   public static void main(String args[])
   {
      //Nested class object can be created
      //Without creating enclosing classobject
      Outclass.Inclass Inobj=new Outclass.Inclass();
      Inobj.Inmethod();
   }
}
```

Output:

```
a of Outclass = 5
b of Inclass = 10
```

2.6.2 Inner Classes [W-18]

- Inner classes are syntactic feature that was added to Java in JDK 1.1. They provide a convenient way for a developer to define classes within the body of another class.
- Java inner class or nested class is a class i.e. declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- Additionally, it can access all the members of outer class including private data members and methods.
- **Syntax:**

```
class Java_Outer_class
{
   //code ...
   class Java_Inner_class
   {
      //code ...
   }
}
```

Advantages of Inner Classes:

1. Inner classes can access all members of the outer class including private members.
2. Inner classes help you hide the implementation of a class completely.
3. Inner classes provides a shorter way of writing listeners.
4. Inner classes helps us to write optimized code.
5. It requires less code to write.

Program 2.19: Program for Inner Class.

```
class TestMemberOuter1
{
   private int data=30;
   class Inner
   {
      void msg(){System.out.println("data is "+data);}
   }
   public static void main(String args[])
   {
      TestMemberOuter1 obj=new TestMemberOuter1();
      TestMemberOuter1.Inner in=obj.new Inner();
      in.msg();
   }
}
```

Output:

```
data is 30
```

2.6.3 Anonymous Inner Class

- It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.
- A class that have 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 2.20: Program for anonymous inner class.

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

Output:

```
nice fruits
```

Summary

- 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 initialize an object when it is declared. This process is referred as initialization.

- Inner classes are syntactic feature that was added to Java in JDK 1.1. They provide a convenient way for a developer to define classes within the body of another class.
- Java inner class or nested class is a class i.e. declared inside the class or interface.
- An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

Check Your Understanding

1. Which is true about an anonymous inner class?
   (a) It can extend exactly one class and implement exactly one interface.
   (b) It can extend exactly one class and can implement multiple interfaces.
   (c) It can extend exactly one class or implement exactly one interface.
   (d) It can implement multiple interfaces regardless of whether it also extends a class.
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. What would be the behaviour if this() and super() used in a method?
   (a) Runtime error (b) Compile time error
   (c) Throws exception (d) Runs successfully
6. What is false about constructor?
   (a) Constructors cannot be synchronized in Java
   (b) Java does not provide default copy constructor
   (c) Constructor can have a return type
   (d) "this" and "super" can be used in a constructor
7. Abstract class cannot have a constructor.
   (a) True (b) False
8. What is the process of defining more than one method in a class differentiated by method signature?
   (a) Function overriding (b) Function overloading
   (c) Function doubling (d) None of the mentioned
9. Which method can be defined only once in a program?
   (a) main method (b) finalize method
   (c) static method (d) private method
10. The new operator is used to create a:
   (a) function object (b) class object
   (c) method object (d) all of above
11. Which of these methods must be made static?
   (a) main() (b) delete()
   (c) run() (d) finalize()
12. Which of these is the method which is executed first before execution of any other thing takes place in a program?
   (a) main method (b) finalize method
   (c) static method (d) private method
13. Which of these data type can be used for a method having a return statement in it?
   (a) void (b) int
   (c) float (d) both int and float

Answers

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

Practice Questions

Q.1 Answer the following Questions in short.

1. What is a method?
2. What is method overloading?
3. What is inner class?

4. Enlist types of inner classes.
5. What is anonymous class?
6. Enlist types constructors.
7. Define Parameterized constructor.
8. Define Default constructor.
9. What is Recursion?
10. What is use of new operator?
11. What is constructor overloading?

Q.II Long Answers the following Questions.

1. Explain the significance of each of the following:
   (i) finalize() (ii) this
   (iii) static (iv) new operator
2. Discuss object reference 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 methods.
7. Explain various types constructors.
8. Create a class circle, consider pi and radius. Calculate the area of a circle and display it.
9. Explain accepting input using Command line argument.
10. Explain anonymous class in detail.
11. Create a class distance with variable feet and inches, method get_val(), show_dist(). Use constructor. Create two objects and display their distance.

Q.III Define the terms.

1. this keyword
2. static keyword
3. class
4. object
5. Constructor
6. Finalize ()

Previous Exam Questions

Winter 2018

1. Write any two advantages of inner class. **[2 M]**

Ans. Refer to Section 2.6.2.

2. Explain the following keywords: Finalized. **[4 M]**

Ans. Refer to Section 2.5.4.

Summer 2019

1. Justify True/False Java is not fully object oriented. **[2 M]**

Ans. Refer to Section 2.1.

■■■

3...

Inheritance, Package and Collection

Learning Objectives...

- To understand Basic Concepts of Inheritance.
- To study Types of inheritance.
- To learn Inheriting Data members and Methods.
- To study Use of super and final keyword.
- To study abstract class.
- To study Interfaces.
- To learn about Packages.
- To study Creating, Accessing and using Packages.
- To study about Collection.

3.1 OVERVIEW OF INHERITANCE

- Object Oriented Programming (OOP) allows you to create new class based on a class that has been already defined using inheritance.
- It is an important feature of the OOP's because it supports the hierarchical classifications, reusability of class and defined to specialization.
- 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.
- 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. 3.1 illustrating a class called Food, which has two subclasses called Pizza and Pasta

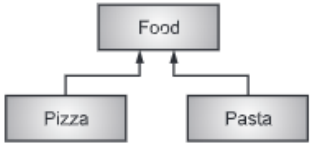


Fig. 3.1: Example of Inheritance

- The class Pizza and Pasta inherits from the class Food. The Food class is the superclass of Pizza and Pasta. Pizza and Pasta are subclasses of Food.

Advantages of Inheritance:

1. Reusability of code.
2. Easy and simple to understand since model structure is provided.
3. Code is easily managed by dividing into parent and child classes.
4. Coding can be done using existing classes and representing the objects.
5. It's used easily to convert smaller systems into larger systems.

3.1.1 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. Note that we cannot extend a subclass from more than one super class. Direct multiple inheritance is not supported in Java.

Program 3.1: 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.

3.1.2 Types of Inheritance

- Fig. 3.2 shows various types of inheritance in Java.

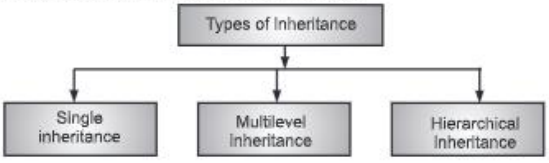


Fig. 3.2: Types of Inheritance

1. Single Inheritance:

- In this one class extends another class at single level.
- In case of single inheritance there is only a sub class and it's parent class. It is also called simple inheritance or one level inheritance.
- Fig. 3.3 shows that class Mango extends only one class which is Fruit. Here, Fruit is a **parent class** of Mango and Mango would be a **child class** of Fruit.

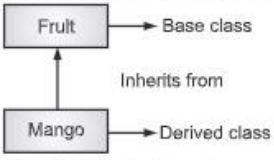


Fig. 3.3: Single Inheritance

- The Fig. 3.3 shows that class Fruit is super class and class Mango is the sub class. Let us consider that we have class 'First' which is inherited by class 'Second', (See Fig. 3.4).

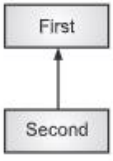


Fig. 3.4: Example of Single Inheritance

Program 3.2: 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/levels 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.

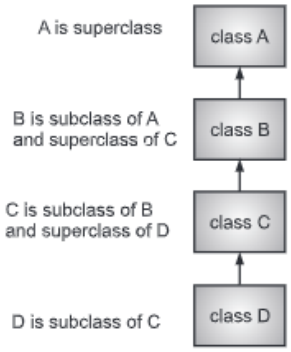


Fig. 3.5: Representation of Multilevel Inheritance

Program 3.3: 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.

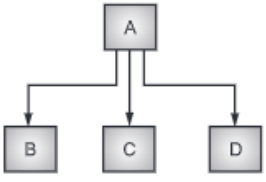


Fig. 3.6: Hierarchical Inheritance

- Fig. 3.7 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.

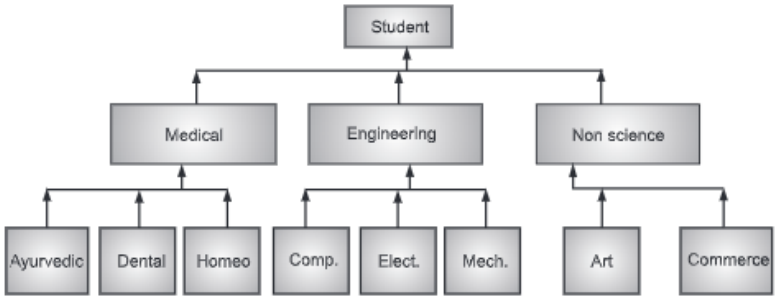


Fig. 3.7: Example of Hierarchical Inheritance

Program 3.4: 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
```

3.2 INHERITANCE IN CONSTRUCTOR

- The constructors never get inherited to any child class in inheritance.
- In Java, the default constructor of a parent class is called automatically by the constructor of its child class. When an object of the child class is created, the parent class constructor is executed, followed by the child class constructor's execution. However, if the parent class contains both default and parameterized constructor, then only the default constructor is called automatically by the child class constructor.

Program 3.5: Program using inheritance, constructor.

```
class Sports
{
    int i;
    Sports()
    {
        System.out.println("Sports Default Constructor Called.....");
    }
```

```
    Sports(int i)
    {
        System.out.println("Sports Parameterized Constructor Called.....");
        this.i = i;
    }
}
class FootBall extends Sports
{
    FootBall()
    {
        System.out.println("\n  FootBall  Class  Default  Constructor  Called
        .....");
    }
}
public class Temp
{
    public static void main(String[] args)
    {
        FootBall obj = new FootBall();
    }
}
```

Output:

```
Sports Default Constructor Called......
FootBall Class Default Constructor Called.....
```

3.3 INHERITING DATA MEMBERS AND METHODS

- In inheritance one class attributes or methods can be inherited in another class. In Java extends keyword is used for inheritance. The class whose properties/ attributes and methods are inherited is called Super class or base class or parent class. Child class/subclass can extend the attributes/methods of Parent class and also can have its own attributes and methods.

Program 3.6: Using Data Members and methods.

```
class DryFruit
{
    int i = 10;
    String s1 = "Almond....."; // Data Members
```

```
    void show() // Member Functions
    {
        System.out.println("\n Enjoy DryFruits ");
        System.out.println("\n \t i = " +i);
        System.out.println("\n \t Dry Friut: = "+ s1);
    }
}
class Pistachio extends DryFruit
{
    int j = 20;
    String s2 = " Pista...."; // Data Members
    void show() // Member Functions
    {
        DryFruit d1 = new DryFruit ();
        d1.show();
        System.out.println("\n \t j = " +j);
        System.out.println("\n \t Dry Friut: = " + s2 );
    }
}
public class Walnut extends Pistachio
{
    int k = 30;
    String s3 = " Walnut...."; // Data Members
    void show() // Member Functions
    {
        Pistachio p1 = new Pistachio();
        p1.show();
        System.out.println("\n\t k = " +k);
        System.out.println("\n \t Dry Friut: = " + s3 );
    }
    public static void main(String[] args)
    {
        Walnut w1 = new Walnut();
        w1.show();
    }
}
```

Output:

```
Enjoy DryFruits
i = 10
Dry Friut: = Almond.....
j = 20
Dry Friut: = Pista....
k = 30
Dry Friut: = Walnut....
```

3.4 MULTILEVEL INHERITANCE – METHOD OVERRIDING HANDLE MULTILEVEL CONSTRUCTORS

- In Multilevel Inheritance one class extends another class which in turn extends another class. Example, Class C is extended from Class B and Class B is extended from Class A. In method overriding we have same function name, parameters/arguments and same prototype in all the inherited classes. We can also have multilevel constructors as shown in the below program.

Program 3.7: Program using multilevel inheritance with multilevel constructors.

```
class DairyMilk
{
    int i;
    DairyMilk(int i)
    {
        this.i = i;
        System.out.println(" Enjoy Dairy Milk.....");
    }
    void display() // Member Functions
    {
        System.out.println("\n Kuch meetha ho jay. DairyMilk ");
    }
}
class DairyMilk_Silk extends DairyMilk
{
    int j;
    DairyMilk_Silk(int i, int j)
    {
        super(i);
        this.j = j;
        System.out.println("Enjoy Dairy Milk Silk....");
    }
```

```
    void display() // Member Functions
    {
        System.out.println("\n Kuch meetha ho jay. DairyMilk_Silk ");
    }
}
class DairyMilk_Hazelnut extends DairyMilk_Silk
{
    int k;
    DairyMilk_Hazelnut(int i, int j, int k)
    {
        super(i, j);
        this.k = k;
        System.out.println("DairyMilk_Hazelnut..... Enjoyed ");
    }
    void display() // Member Functions
    {
        System.out.println("\n Kuch meetha ho jay. DairyMilk_Hazelnut ");
    }
}
public class MultiLevel
{
    public static void main(String arg[])
    {
        System.out.println("\n _____");
        DairyMilk d1 = new DairyMilk (100);
        d1.display();
        System.out.println("\n _____");
        DairyMilk_Silk d2 = new DairyMilk_Silk(200,300);
        d2.display();
        System.out.println("\n _____");
        DairyMilk_Hazelnut d3 = new DairyMilk_Hazelnut(400, 500, 600);
        d3.display();
        System.out.println("\n _____");
    }
}
```

Output:

```
_____________________
Enjoy Dairy Milk.....
Kuch meetha ho jay. DairyMilk

_____________________
Enjoy Dairy Milk.....
Enjoy Dairy Milk Silk....
Kuch meetha ho jay. DairyMilk_Silk

_____________________
Enjoy Dairy Milk.....
Enjoy Dairy Milk Silk....
DairyMilk_Hazelnut..... Enjoyed
Kuch meetha ho jay. DairyMilk_Hazelnut
```

3.5 USE OF 'super' AND 'final' KEYWORD [S-19]

3.5.1 Use of 'super' Keyword

- 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 3.8: 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 3.9: 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.
- **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 3.10: 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.

3.5.2 Use of 'final' Keyword [W-18]

- 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.

3.5.2.1 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 3.11: 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
```

3.5.2.2 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 3.12: 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
```

3.5.2.3 Use of final keyword related to Classes

- If you 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 3.13: 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
```

3.6 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.

3.7 CREATION AND IMPLEMENTATION OF AN INTERFACE, INTERFACE REFERENCE

3.7.1 Defining Interface [W-18, S-19]

- 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);
}
```

3.7.2 Implementation of an 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 3.14: Program of interface implementation. [W-18]

```
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
```

3.7.3 Interface Reference

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 3.15: Following program shows the partial implementation of an interface. **[S-19]**

```
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
```

3.8 INTERFACE INHERITANCE

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 3.16: 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 ,,,,,, ");
      System.out.println("\n ");
   }
   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. 3.8: Multiple Inheritance in Java

Program 3.17: Program to implementation of multiple inheritance using Interface.

```
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
```

3.8.1 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.
- 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 3.18: Program declare 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();
        //Upcasting Here
        message.msg();
    }
}
```

Output:

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

3.9 DYNAMIC METHOD DISPATCH / RUNTIME POLYMORPHISM

Runtime Polymorphism:

- Polymorphism in Java is a concept by which different ways can we used to perform different action. Polymorphism implies many forms which is derived from 2 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 following program, 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 3.19: 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 3.20: 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...
```

Runtime Polymorphism can be categorized in following two parts:

1. **Runtime Polymorphism with Data Member:**

- The method is overridden but not the data members, so run-time polymorphism cannot be done with data members.
- In the program given below, we created a Student class and Monitor class. Both the classes have data members, in other words name and roll number. We are accessing the data members by the reference variable of the parent class that refers to the subclass object.

- Since, we are accessing the data members of the subclass that are not overridden, the data members of the parent class will always be invoked.

Program 3.21: Program using Runtime Polymorphism.

```
class Student
{
   String name="Om";
}
class Monitor extends Student
{
   String name="Yash";
   public static void main(String args[])
   {
      Student s=new Monitor();
      System.out.println(s.name);
   }
}
```

Output: Om

2. Runtime Polymorphism with Multilevel Inheritance:

- In method overriding, a subclass overrides a method with the same signature as that of in its superclass. Reference type check is made during compile time. We can override a method at any level of multilevel inheritance. See the program below to understand the concept.

Program 3.22: Runtime Polymorphism with Multilevel inheritance.

```
class Electronics
{
   void function()
   {
      System.out.println("Made for various purpose");
   }
}
class Phone extends Electronics
{
   void function()
   {
      System.out.println("Display,Voice");
   }
}
```

```
public class Computer extends Phone
{
    void function()
    {
        System.out.println("Calculation,Internetworking");
    }
    public static void main(String args[])
    {
        Electronics e1=new Electronics();
        Electronics e2=new Phone();
        Electronics e3=new Computer();
        e1.function();
        e2.function();
        e3.function();
    }
}
```

Output:

```
Display,Voice
Calculation,Internetworking
Made for various purpose
```

3.10 ABSTRACT CLASS

- 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.
- 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.

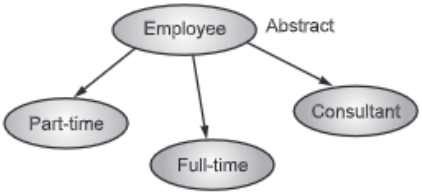


Fig. 3.9: Abstract Class

- 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.

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 3.23: 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
```

Program 3.24: Program using abstract class and methods.

```
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.
- In this example, if you create the instance of `Rectangle` class, `draw()` method of `Rectangle` class will be invoked.

3.11 COMPARISON BETWEEN ABSTRACT CLASS AND INTERFACE

- Abstraction can be achieved using abstract class and interface in which you can declare the abstract methods. You cannot instantiate abstract class and interface.

Table 3.1: Comparison between abstract class and interface [S-19]

Abstract class	Interface
1. In this abstract and non-abstract methods can be used.	It can include only abstract methods. Default and static methods can also be included.
2. In this multiple inheritance is not supported.	It support's multiple inheritance.
3. Final, non-final, static and non-static variables can be used	Only static and final variables can be used.
4. Implementation of interface is provided.	Implementation of abstract class can't be provided.
5. It uses abstract keyword.	It uses interface keyword.
6. It might extend another Java class and implement multiple Java interfaces.	It might extend another Java interface only.
7. "extends" keyword is used to extend an abstract class.	"implements" keyword is used to implement an interface.
8. Class members like private, protected, etc. can be used.	By default members are public.

9. **Example:**
`public abstract class Movie`
`{`
`public abstract void trailer();`
`}`
	Example:
	`public interface Movie`
	`{`
	`void trailer();`
	`}`

3.12 ACCESS CONTROL

- In this access of certain members of a class is restricted to specific parts of a program. Access modifiers can be used to control the access to members of a class. There are four access modifiers in Java. They are:
  1. public
  2. protected
  3. default
  4. private

- If the member (variable or method) is not declared as either public or protected or private, the access modifier for that member will be default.
- Among the four access modifiers, private is the most restrictive access modifier and public is the least restrictive access modifier.
- Syntax for declaring a access modifier is as follows:

```
access-modifier data-type variable-name;
```

- Private integer variable declaration is as follows:

```
private int x;
```

Program 3.25: Program using access modifier.

```
public class Student
{
   private int Rno;
   private String name;
   private float mks;
   public void disp(float mks)
   {
      if(mks < 0)
      {
         System.out.println("Please enter Mks > 0 ");
      }
      else
      {
          // marks = mks;
            System.out.println(" Students Mks = "+ mks);
      }
   }
   public static void main(String args[])
   {
      Student s1  = new Student();
      s1.disp(98);
   }
}
```

Output:

```
Students Mks = 98.0
```

3.13 PACKAGES [W-18, S-19]

- 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 grow 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 are 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".
- 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.

3.13.1 Packages Concept

- 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.

3.13.2 Creating user defined Packages [S-19]

- 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.

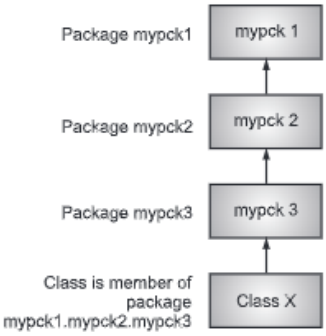


Fig. 3.10: Package Hierarchy

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

3.13.2.1 Accessing and Using Packages

- 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 3.26: Program for package that import the packagename. **[W-18]**

```
//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 3.27: 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 3.28: 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
```

3.13.3 Java Built-in Packages

- In Java group of related classes are put together in a package. E.g. File directory has different folders.
- In Java you can either use built-in packages or user defined packages.
  - **java.lang:** It has classes for primitive types, strings, threads, exceptions and math functions.
  - **java.io:** It has stream classes for Input/Output.
  - **java.awt:** Classes for implementing Graphical User Interface – menus, windows, buttons etc.
  - **java.util:** It has classes such as dates, Calendars, vectors, hash tables, etc.
  - **java.net:** Classes for networking.
  - **java. Applet:** Classes for creating and implementing applets.

3.13.4 Import statement, Static import

Import statement:

- Import statement brings certain classes or the entire packages, into visibility. Class can be referred directly by using only its name once imported.
- Import statements occurs immediately followed by package statement (if exists) and before any class definitions.
- Import statement is as:

```
import packg1[.packg2].(classname|*);
```

- `packg1` is the name of top-level package, and `packg2` is the name of subordinate package inside outer package separated by dot (.). Explicit classname or a star (*), indicates that the Java compiler must import the full package.

```
import java.util.Date;
import java.io.*;
```

- In a Java package, all the basic Java classes are stored. The fundamental language functions are stored in a package inside of the java package i.e., java.lang.

Program 3.29: Program using import statement.

```
import java.lang.Math.*;
public class Sample
{
    public static void main(String args[])
    {
        int i=30, x=3;
        System.out.println("   Sqrt   of   25   =   "   +   Math.sqrt(625));
        System.out.println(" \n Absolute of -2178 = " + Math.abs(-2178));
        System.out.println(" \n Power of 2 = " + Math.pow(2,3));
        // return a power of 2
        System.out.println("\n Exp of x = " + Math.exp(x));
        System.out.println("\n Cosine value of i = " +Math.cos(i));
    }
}
```

Output:

```
Sqrt of 25 = 25.0
Absolute of -2178 = 2178
Power of 2 = 8.0
Exp of x = 20.085536923187668
Cosine value of i = 0.15425144988758405
```

Static import:

- It is used to allow Java program to access any static member of a class directly so less coding is required. Its not needed to qualify it by the class name.

Program 3.30: Program using Static import.

```
import static java.lang.System.*;
public class Happy
{
    public static void main(String args[])
    {
        out.println("\n Be Happy ……");
        out.println("\t\t Always ……");
    }
}
```

Output:

```
Be Happy ……
Always ……
```

3.14 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 3.2: 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
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 3.3: List of interfaces defined by java.util package

Collection	List	Queue
Comparator	Listlterator	RandomAccess
Deque	Map	Set
Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap	SortedSet
Formattable	NavigableSet	
lterator	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. It was added by J2SE 1.2. Before Collections framework, Java had ad hoc 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. It simply provides a better way of doing things.

3.14.1 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. You can use these implementations as it is. 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.

3.14.2 Interfaces: Collection, List, Set

3.14.2.1 Collection Interface

- Collection in java represents a single unit of objects i.e. a group. The Java Collection interface is the root of the collection interface hierarchy.
- The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have.
- Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an UnsupportedOperationException.

Table 3.4: Collection interfaces

Interface	Explanation
`Collection`	Allow you to work with groups of objects; Collection interface is at the top of the collections hierarchy.
`Deque`	It extends Queue that handles a double-ended queue.
`List`	It extends Collection which handles sequences.
`NavigableSet`	Extends SortedSet to handle reclaimation of elements based on closest-match searches.
`Queue`	Extends Collection to handle special types of lists in which elements are to be removed only from the head.
`Set`	Extends Collection to handle sets that contain unique elements.
`SortedSet`	Extends Set to handle sorted sets.

- Methods defined by Collection Interface are given in Table 3.5.

Table 3.5: Methods defined by Collection Interface

Method	Explanation
`boolean add(E obj)`	Adds `obj` to the invoking collection. Returns true if `obj` is inserted to the collection. Returns false if `obj` is already a member of the collection and the collection does not allow duplicates.
`boolean addAll(Collection<? extends E> c)`	Adds all the elements of `c` to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false.
`void clear( )`	Removes all elements from the invoking collection.
`boolean contains(Object obj)`	Returns true if `obj` is an element of the invoking collection. Otherwise, returns false.
`boolean containsAll(Collection<?> c)`	This method returns true if the invoking collection contains all elements of `c`. Otherwise, a method returns false.
`boolean equals(Object obj)`	Returns true if the call upon (invoking) collection and `obj` are equal. Otherwise, it returns false.

contd. ...

`int hashCode( )`	Returns the `hashcode` for the invoking collection.
`boolean isEmpty( )`	Returns true if the invoking collection is empty. Otherwise, returns false.
`Iterator<E> iterator( )`	Returns an iterator for the invoking collection.
`boolean remove(Object obj)`	Removes an instance of `obj` from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
`boolean removeAll(Collection<?> c)`	Removes all elements of `c` from the invoking collection. Returns true if the elements were removed. Otherwise, returns false.
`boolean retainAll(Collection<?> c)`	Removes all elements from the invoking collection except those in c. Returns true if the elements were removed from the invoking collection. Otherwise, returns false.
`int size( )`	Returns the number of elements seized in the invoking collection.
`Object[ ] toArray( )`	Returns an array which contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.

3.14.2.2 List Interface

- 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 3.6: 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.

contd. ...

`boolean addAll(int index, Collection<? extends E> c)`	It adds all elements of `c` into the invoking list at given index passed in index parameter. Any previous elements at or after the point of insertion are shifted up. Thus, elements are not overwritten. Method returns true if the invoking list changes and returns false otherwise.
`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.
`int lastIndexOf(Object obj)`	Returns the index of the last 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.
`ListIterator(E> listIterator(int index)`	Returns an iterator that starts at the specified index 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.
`List<E> subList(int start, int end)`	Returns a list which includes elements from start to (end –1) in the invoking list. Elements in the returned list are also referenced by the invoking object.

3.14.2.3 Set Interface

- 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.

SortedSet Interface:

- SortedSet interface extends Set. The behavior of a collection is that a set is sorted in ascending order. SortedSet defines several methods that make set processing more convenient. SortedSet interface methods are summarized below in the table.

Table 3.7: Methods defined by SortedSet interface

Method	Explanation
`Comparator<? super E> comparator( )`	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
`E first( )`	Returns the first element in the invoking sorted set.
`SortedSet<E> headSet(E end)`	Returns a SortedSet containing those elements less than given end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
`E last( )`	Returns the last element in the invoking sorted set.
`SortedSet<E> subSet(E start, E end)`	Returns a SortedSet which includes those elements between start and (end –1). Elements in the returned collection are also referenced by the invoking object.
`SortedSet<E> tailSet(E start)`	Returns a SortedSet which contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

3.14.3 Navigation: Enumeration, Iterator, ListIterator

3.14.3.1 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 3.31: Program 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]
```

3.14.3.2 ListIterator Interface

- ListIterator extends Iterator. With the help of ListIterator, we can traverse the elements of the list bi-directionally.

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 3.32: 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
```

3.14.3.3 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 3.33: 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());
            }
         }
      }
```

Output:

```
New Elements In Vector ........
1000
2000
3000
true
```

3.14.4 Classes: LinkedList, ArrayList, Vector, HashSet

3.14.4.1 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 elements 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 has 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 3.34: 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.

```
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]
```

3.14.4.2 ArrayList [W-18]

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

Table 3.8: 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 performs fast.	Slow down the performance.
It stores both objects and primitives type.	It converts primitive type to object.
We use for loop or for each loop to iterate over an array.	We use an 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 can be multi-dimensional.	ArrayList is always 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 3.35: 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.

```
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);
```

3.14.4.3 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 3.9: 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 3.10: 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.

contd. ...

`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.
`void ensureCapacity(int size)`	Sets the minimum capacity of the vector to size.
`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.
`int indexOf(Object element, int start)`	Returns the index of the first occurrence of element at or after start. If the object is not in that portion of 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.
`E lastElement( )`	Returns the last element in the vector.
`int lastindexOf(Object element)`	Returns the index of the last occurrence of element. If the object is not in the vector, –1 is returned.
`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.
`boolean removeElement(Object element)`	Removes element from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns true if successful and false if the object is not found.

contd. ...

`void removeElementAt(int index)`	Removes the element at the given location specified by index.
`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 3.36: 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

```
File: VectorDemo.java
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
```

3.14.4.4 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 3.37: 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]
```

Program 3.38: Define an abstract class shape with abstract method area() and volume(). Write a java program to calculate area and volume of cone and cylinder. **[S-19]**

```
import java.util.*;
abstract class Shape
{
   final float PI=3.14f;
   Scanner s=new Scanner(System.in);
   abstract void area();
   abstract void volume();
}
class Cylinder extends Shape
{
   float a,r,h,v;
```

```
    void accept()
    {
        System.out.println("Please enter Radius, Height ");
        r=s.nextFloat();
        h=s.nextFloat();
    }
    void area()
    {
        a = (2*PI*r*h)+(2*PI*r*r);
        System.out.println("Area of Cylinder  :"+a);
    }
    void volume()
    {
        v=PI*r*r*h;
        System.out.println("Volume of Cylinder  :"+v);
    }
}
class Cone extends Shape
{
    float a,r,h,v;
    void accept()
    {
        System.out.println("Please Radius, Height ");
        r=s.nextFloat();
        h=s.nextFloat();
    }
    void area()
    {
        a=PI*r*(r+(float)Math.sqrt(h*h+r*r));
        System.out.println("Area of Cone is :"+a);
    }
    void volume()
    {
        v=PI*r*r*(h/3);
        System.out.println("Volume of Cone is :"+v);
    }
}
```

```
class Temp
{
   public static void main(String args[])
   {
      Cylinder s1=new Cylinder();
      Cone s2=new Cone();
      s1.accept();
      s2.accept();
      s1.area();
      s1.volume();
      s2.area();
      s2.volume();
   }
}
```

Output:

```
java Temp
Please Radius, Height
12
23
Please Radius, Height
67
45
Area of Cylinder : 2637.6
Volume of Cylinder : 10399.68
Area of Cone : 31075.094
Volume of Cone : 211431.9
```

Program 3.39: Write a package for games in java, which have two classes Indoor and Outdoor. Use a function display() to generate the list of players for specific games. (Use parameterized constructor, finalize() method and array of objects). **[W-18, S-19]**

Indoor.java

```
package games;
public class Indoor
{
   protected String player;
   public Indoor(){  }
   public Indoor(String p)
   {
      player = p;
   }
```

```
    public void display()
    {
        System.out.println(player);
    }
    protected void finalize()
    {
        System.out.println("\nTerminating Indoor Class");
    }
}
```

Outdoor.java

```
package games;
public class Outdoor
{
    protected String player;
    public Outdoor(){}
    public Outdoor(String p)
    {
        player = p;
    }
    public void display()
    {
        System.out.println(player);
    }
    protected void finalize()
    {
        System.out.println("\nTerminating Outdoor Class");
    }
}
```

Main

```
package testgame;
import games.*;
import java.util.*;
class Games
{
    public static void main(String args[])
    {
        int n,m,i;
```

```
        Scanner s=new Scanner(System.in);
        System.out.println("How many indoor players you want ");
        n=s.nextInt();
        System.out.println("How many outdoor players you want ");
        m=s.nextInt();
        Indoor in[]=new Indoor[n];
        Outdoor out[]=new Outdoor[m];
        for(i=0;i<n;i++)
        {
           System.out.println("Enter indoor player name :");
           String name=s.next();
           in[i]=new Indoor(name);
        }
        for(i=0;i<m;i++)
        {
           System.out.println("Enter outdoor player name :");
           String name=s.next();
           out[i]=new Outdoor(name);
        }
        System.out.println("Indoor Players : ");
        for(i=0;i<n;i++)
        {
           in[i].display();
        }
        System.out.println("Outdoor Players : ");
        for(i=0;i<m;i++)
        {
           out[i].display();
        }
    }
    protected void finalize()
    {
        System.out.println("Finalize is called");
    }
}
```

Output:

```
java testgame.Games
How many indoor players you want
2
How many outdoor players you want
2
Enter indoor player name :
Shreyas
Enter indoor player name :
Pranay
Enter outdoor player name :
Ira
Enter outdoor player name :
Siddhi
Indoor Players :
Shreyas
Pranay
Outdoor Players :
Ira
Siddhi
```

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

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. abstract class
5. method overriding

Previous Exam Questions

Winter 2018

1. List the three different uses of final keyword. **[2 M]**

Ans. Refer to Section 3.5.2.

2. Can interface be final? Justify. **[2 M]**

Ans. Refer to Section 3.5.2.

3. Explain ArrayList class in Collection Framework. **[4 M]**

Ans. Refer to Section 3.14.4.2.

4. Explain how multiple inheritance is achieved in Java. **[4 M]**

Ans. Refer to Section 3.7.1.

5. Write a note on package in Java. **[4 M]**

Ans. Refer to Section 3.13.

6. Create a package vehicle which will have two classes: Two wheeler and four-wheeler. Two-wheeler with method display (cc, price), four-wheeler with method display (reg-no, reg-year). **[4 M]**

Ans. Refer to Program 3.39.

7. What is Interface? Explain it with example. **[4 M]**

Ans. Refer to Section 3.7.

8. Define an Interface shape with abstract method area( ). Write a Java program to calculate an area of Circle and Rectangle (Use final keyword). **[4 M]**

Ans. Refer to Program 3.14.

Summer 2019

1. How to create and access the package in Java? **[2 M]**

Ans. Refer to Section 3.13.

2. What is Interface? Why are they used in Java? Explain. **[4 M]**

Ans. Refer to Section 3.7.

3. Define an abstract class shape with abstract method area() and volume(). Write a java program to calculate area and volume of cone and cylinder. **[4 M]**

Ans. Refer to Program 3.38.

4. Write a package for games in java, which have two classes Indoor and Outdoor. Use a function display() to generate the list of players for specific games. (Use parameterized constructor, finalize() method and array of objects). **[4 M]**

Ans. Refer to Program 3.39.

5. Define collection. Explain any two classes used with collection. **[4 M]**

Ans. Refer to Section 3.14.

6. How to add user defined package in main class? List the steps with suitable example. **[4 M]**

Ans. Refer to Section 3.13.2.

7. Write similarities and dissimilarities between Abstract Class and Interface. **[4 M]**

Ans. Refer to Section 3.5.

8. Explain the use of super and final keyword with reference to inheritance. **[4 M]**

Ans. Refer to Section 3.11.

■■■

4...

File and Exception Handling

Learning Objectives...

- To learn Exception and Error.
- To understand use of try, catch, throw, throws and finally.
- To study Built in Exception and Custom exception.
- To understand File Handling.
- To learn FileReader and FileWriter class.

4.1 EXCEPTION AND ERROR

4.1.1 Exception [S-19]

- 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. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- 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 then it is 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. So 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
}
```

4.1.2 Error

- The errors which can't be predicted are exceptional conditions in the programs.
- An error is an abnormal condition. eg. StackOverFlowError, IllegalAccessErrors, InternalError etc.
- Three types of Errors are runtime errors, syntax errors, logic errors.

1. Syntax Error:

- Syntax errors or compile errors are detected by the compiler if there are errors in code construction, such as mistyping a keyword, punctuation is omitted, mismatched braces etc.
- **Example:**

```
public class Syntx_err
{
   public static main(String[] args)
   {
       System.out.println("All the Best");
   }
}
```

```
/Syntx_err.java:2: error: invalid method declaration; return type required
public static main(String[] args)
{
   ^
      /Syntx_err.java:3: error: unclosed string literal
   System.out.println("All the Best");
   ^
   2 errors
}
```

2. Runtime Errors:

- Due to Runtime error programs terminates abnormally. It's detected when a program is running if the environment detects an operation that is impossible to carry out. Input mistakes typically cause runtime errors. For example, data-type errors, division by zero.
- **Example:**

```
public class RunTm_err
{
   public static void main(String[] args)
   {
      System.out.println(1012/0);
   }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
RunTm_err.main(RunTm_err.java:4)
```

3. Logical errors:

- It occurs when a program does not perform in desired way since there is logical error.
- **Example:**

```
public class LogicalErr
{
   public static void main(String[] args)
   {
      int num = 1234;
      int reverse = 0;
      int rem ;
      while (num != 0)
      {
```

```
            rem = num  / 10;    // to get correct output use % sign
            reverse = reverse * 10 + rem;
            num /= 10;
         }
         System.out.println("Reverse Num:  "+ reverse);
      }
   }
```

Output:

```
Reverse Num:  124210
```

4.2 USE OF TRY, CATCH, THROW, THROWS AND FINALLY [S-19]

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

1. Try:

- This block of code might rise 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:

- It is used to catch an exception when its 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 4.1: Program 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:

- 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 4.2: 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: [S-19]

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

Output:

```
Salary > 20000 .. Enjoy ….
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 4.4: 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 ...
```

4.3 BUILT IN EXCEPTION

- These exceptions are available in Java Libraries which explains the error situation. Various built-in exceptions are Arithmetic exception, FileNotFoundException, ArrayIndexOutOfBoundException, ClassNotFoundException, IOException, NullPointerException etc.

Program 4.5: Program for built in exception.

```
public class Arr_Err
{
   public static void main(String args[])
   {
      int arr[] = { 66,89,23,54,45,11,59,31 };
      System.out.println(arr[9]);
   }
}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index
9 out of bounds for length 8 at Arr_Err.main(Arr_Err.java:6)
```

4.4 CUSTOM 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, and 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 4.6: Following program creates a custom exception type.

```
class MyException extends Exception
{
   private int detail;
   MyException(int a)
   {
      detail = a;
   }
   public String toString()
   {
      return "MyException[" + detail + "]";
   }
}
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:

```
Error: Main method not found in class MyException, please define the main
method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
Predefined functions in Throwable class are getStackTrace(),
printStackTrace(),toString() etc.
```

4.5 THROWABLE CLASS

- All exception types are subclasses of the built-in class `Throwable`. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException.

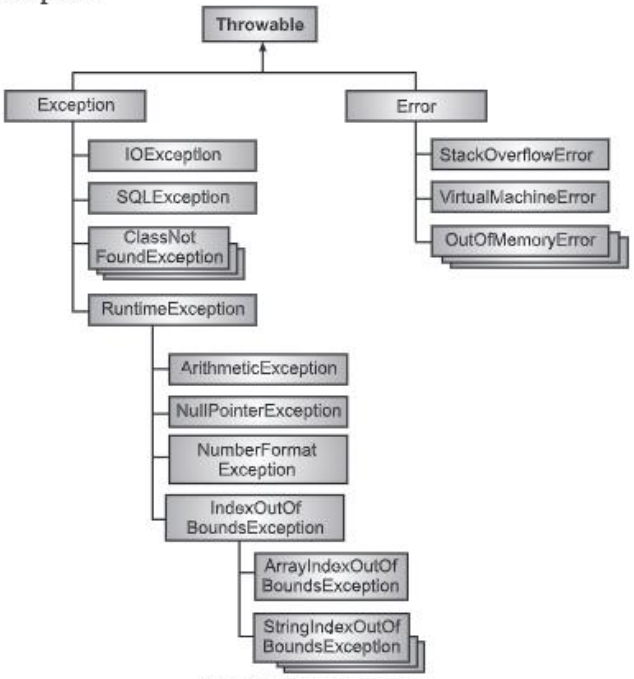


Fig. 4.1: Throwable class

Program 4.7: Program using Throwable.

```
import java.io.*;
public class Th_eg
{
   public static void main(String[] args) throws Exception
   {
      try
      {
         trial();
      }
```

```
        catch (Throwable e)
        {
            System.out.println("\n Inside Throwable Exception:" +
                                                        e.toString());
        }
    }
    public static void trial() throws Exception
    {
        throw new Exception("\n  Inside Trial ..... Exception Thrown .....");
    }
}
```

Output:

```
Inside Throwable Exception: java.lang.Exception:
Inside Trial ..... Exception Thrown .....
```

4.6 OVERVIEW OF DIFFERENT STREAM (BYTE STREAM, CHARACTER STREAM)

- The input or output operation can be performed with files, network connections, memory buffers etc. You will not find any difference between input and output when you write Java programs. They are all treated same by the abstraction stream. These various sources and destinations are handled by the stream.
- A stream is nothing but a logical entity that is linked with the data source or destinations. Data flows through this logical entity from source (input) to destination (output). Input may come from mouse, keyboard, memory etc. whereas output may go to screen, memory, disk etc.
- Java uses a concept of streams to represent ordered sequence of data. A stream in Java is a path along which data flows. It is like a pipe through which water flows. It has a source and destination. The source and the destination may be the physical devices, programs or other streams in the same program. Java streams are classified into two streams namely, input stream and output stream.
- An input stream reads data from the source file and sends it to the program while output stream takes data from the program and sends it to the destination file.

Stream Classes:

- Java's I/O package contains variety of stream classes. These classes can be categorized into two groups based on the data type.
  1. **Byte stream classes** provide support for I/O operations on bytes.
  2. **Character stream classes** provide support for I/O operations on characters.
- Java's stream based I/O has four abstract classes: InputStream, OutputStream, Reader and Writer. InputStream and OutputStream deal with bytes i.e. read/write bytes.
- Reader and Writer are designed for character streams. Character streams read or write characters. Character streams are high level streams. High level streams work with character data and primitive data types which provides meaningful data for programmers.

4.6.1 Byte Streams

- As the name implies, byte streams handle reading and writing of bytes. InputStream and OutputStream are the abstract classes designed for byte streams. Each of these abstract classes has several concrete subclasses which can be used for reading from or writing to various devices like disk files, network connections, memory buffers etc.
- Byte streams are low level streams. Low level streams will work with raw data as stored by the file system. This type is useful for applications that read or write image data.
- All stream methods are synchronized. This implies that methods of these streams return only after reading or writing bytes. They will wait till the data is available and then perform the operation and return.
- You should note that the methods of byte streams return the integer. The lower eight bits of the integer will be used to represent the byte read.

1. **InputStream**:

- It is an abstract class. We cannot create an instance of this class.
- Most of its methods throw an IOException in case of errors.

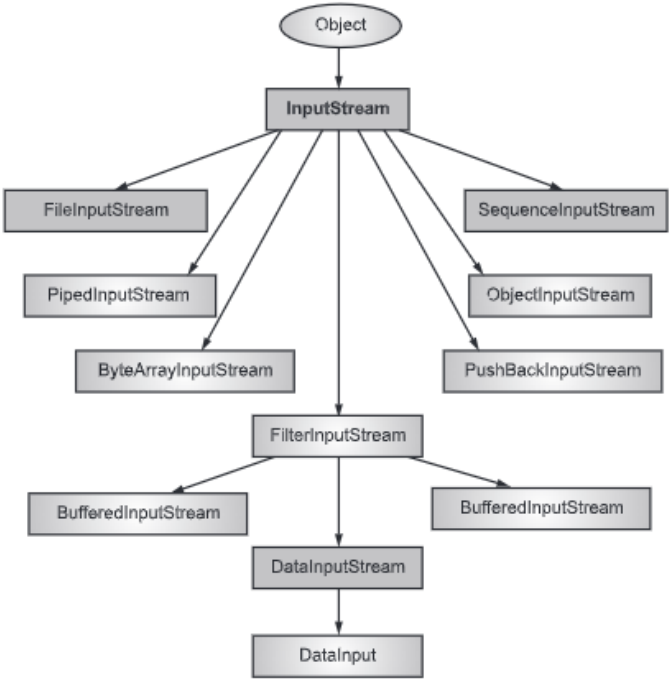


Fig. 4.2: Hierarchy of InputStream Classes

- Table shows methods in InputStream class.

Table 4.1: Methods in InputStream Class [W-18]

Sr. No.	Method	Description
1.	`read()`	It reads a byte from the input stream.
2.	`read (byte b[ ])`	It reads an array of bytes into `b`.
3.	`read (byte b[ ], int n, int m)`	It reads `m` bytes into `b` starting from n^{th} byte.
4.	`available( )`	It gives number of bytes available in the input.
5.	`skip(n)`	It skips over `n` bytes from the inputs stream.
6.	`reset( )`	It goes back to the beginning of the stream.
7.	`close( )`	It closes the input stream.

- Here, are some of the popularly used concrete classes which are sub classes of InputStream abstract class.

(A) ByteArrayInputStream:

- This stream is similar to the FileInputStream but ByteArrayInputStream reads data from an array or from a portion of an array.
- The following are the constructors used to create byte array input streams:

```
ByteArrayInputStream(byte byteArray[])
```

- The above form constructs a stream to read from the specified byte array.

```
ByteArrayInputStream(byte byteArray[], int index, int size)
```

- The above form constructs a stream to read from a portion of the specified byte array that is from given index and of the given size.

Program 4.8: A program that demonstrates ByteArrayInputStream class. We create two byte array streams- one to store sample string in bytes and the other to store a part of the sample byte stream.

```
import java.io.*;
class ByteArrayDemo
{
   public static void main(String args[])
   {
      String myString = "One Two Buckle my shoes";
      byte[] byteArray = myString.getBytes();
      byte b1[];
      int c;
      try
      {
         ByteArrayInputStream arrayInp1 = new ByteArrayInputStream
                                                          (byteArray);
```

```
            ByteArrayInputStream arrayInp2 = new
            ByteArrayInputStream(byteArray,0, 7);
            bl = new byte[byteArray.length];
            for(int i=0; i<2; i++)
            {
                arrayInp1.read(b1);
                System.out.println(new String(b1));
                while((c= arrayInp2.read())!= -1)
                System.out.print( (char) c );
                System.out.println();
                arrayInp1.reset();
                arrayInp2.reset();
            }
        }
        catch(IOException e)
        {
            System.out.println("Error in opening the stream");
        }
    }
}
```

Output:

```
One Two Buckle my shoes
One Two
```

(B) BufferedInputStream:

- This class is extended from FilterInputStream. It is a high level stream. It reads data from a stream and associates a memory buffer to the stream. The reading will be done from this buffer and hence will be much faster as compared to reading directly from input streams. This enhances the performance and also supports operations like skipping, marking and resetting.
- The constructors are as follows:
  1. `BufferedInputStream(InputStream ins)`
  2. `BufferedInputStream(InputStream ins, int bufferSize)`
- The first form wraps the input stream into a buffered stream. The default size of the buffer is used here. The next form allows you to specify a buffer size. The optimum size of the buffer will depend on the host operating system. The smallest size enhances the performance of I/O stream.

- The following statement will wrap the FileInputStream to the buffered stream.
  - `FileInputStream fis = new FileInputStream("test.txt");`
  - `BufferedInputStream bis = new BufferedInputStream(fis);`
  - `bis.read()` method will read data from the buffer and not from the file.

(C) DataInputStream:

- This is a high level 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 4.2 shows commonly used methods. All these methods throw IOException.

Table 4.2: Methods in DataInputStream 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 a 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.

2. OutputStream:

- OutputStream is an abstract class. Most of the methods of this class return void and throw IOException in case of errors.

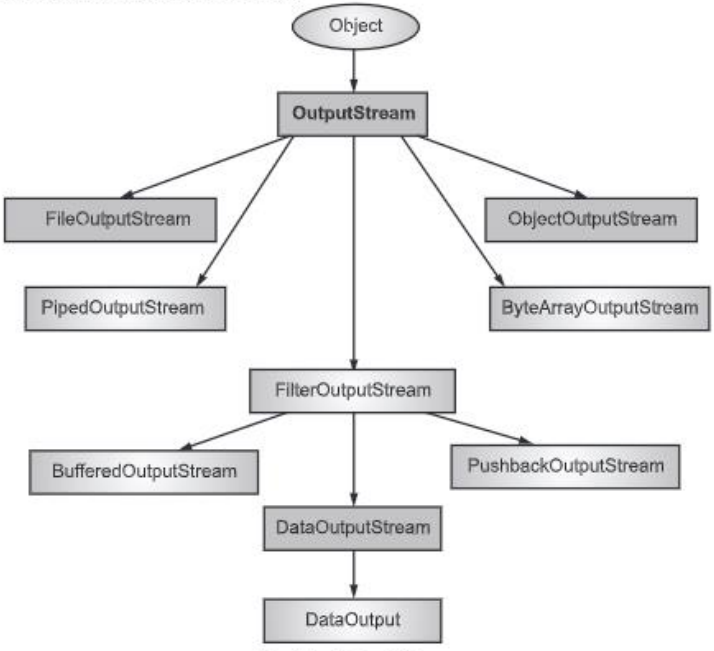


Fig. 4.3: OutputStream

- Table 4.3 shows methods in OutputStream class.

Table 4.3: Methods of OutputStream class

Sr. No.	Method	Description
1.	`write( )`	It writes a byte to the output stream.
2.	`write (byte b[ ])`	It writes all bytes in the array b to the output stream.
3.	`write (byte b[ ], int n, int m)`	It writes m bytes from array b starting from n^{th} byte.
4.	`close( )`	It closes the output stream.
5.	`flush( )`	It flushes the output stream.

(A) ByteArrayOutputStream:

- This class is similar to the ByteArrayInputStream but only it writes bytes to the array. The default array length is 32 bytes. If required, you can also specify the byte array length.

- The constructors are as follows:
  - `ByteArrayOutputStream()`
  - `ByteArrayOutputStream(int byteArrLength)`

Program 4.9: Program using ByteArrayOutputStream.

```
import java.io. *;
class ByteArrayOutputStreamDemo
{
   public static void main(String [] arg) throws Exception
   {
      ByteArrayOutputStream b = new ByteArrayOutputStream();
      String myString = "One Two Buckle my shoes";
      byte[ ] byteArray = myString.getBytes();
      b. write(byteArray );
      byte[] ary = b.toByteArray();
      for (int i=0; i<ary.length; i++)
      System.out.print ((char) ary[i]);
      System.out.println("\n" + b.toString());
      b.reset();
      for(int i=0;i<ary.length;i++)
      b.write('#');
      System.out.println(b.toString());
   }
}
```

Output:

```
One Two Buckle my shoes
One Two Buckle my shoes
```

(B) BufferedOutputStream:

- This stream works with a buffer. It keeps data in buffer and then writes. `flush()` method writes data from the buffer to the output device.
- BufferedOutputStream helps to improve performance.
- The default size of the buffer is 512 bytes. If required, you can change the size of the buffer using an appropriate form of constructor.
- The constructors are as follows:
  - `BufferedOutputStream(OutputStream os)`
  - `BufferedOutputStream(OutputStream os, int size)`

(C) PrintStream:

- PrintStream extends from FilterOutputStream. PrintStream is one of the most used Java class. This class provides all of the output capabilities.
- The following are the constructors:
  - `PrintStream(OutputStream os)`
  - `PrintStream(OutputStream os, boolean flushOnNewLine)`
  - `PrintStream(OutputStream os, boolean flushOnNewLine, String charSet) throws`
- Here,
  - `os` defines the OutputStream that will receive output.
  - Unsupported encoding exception `flushOnNewLine` parameter is true, the output is automatically flushed every time a newline (`\n`) character is recognized or when `println` is called. If false, it is not flushed automatically.
  - `charSet` specifies character encoding scheme.

(D) DataOutputStream:

- This stream also extends from FilterOutputStream. This writes primitive data to a byte oriented 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.

4.6.2 Character Streams

- Character streams can be used to read and write 16 bit unicode characters. Like byte stream classes, character stream classes are reader stream and writer stream classes.

Reader:

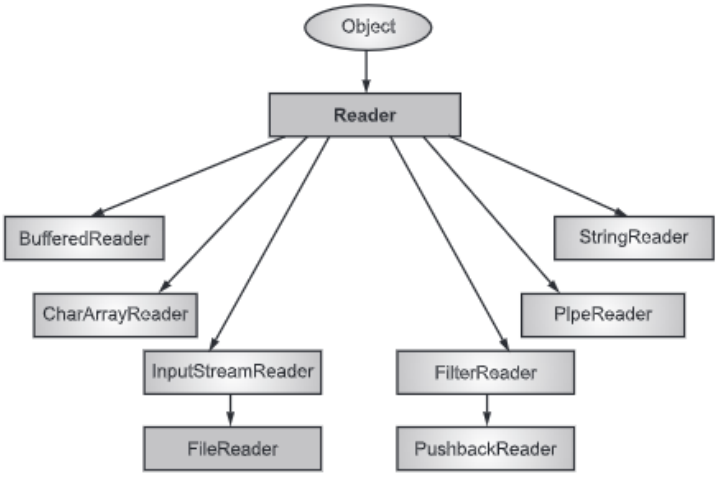


Fig. 4.4: Hierarchy of Reader Stream Classes

- Reader class is the base class for all the classes in this group. It is an abstract class. This class is quite similar to InputStream class in functionality.
- Methods of this class are also identical to those of InputStream class. The only difference is that the fundamental unit of data is character instead of byte.

Writer:

- Writer class is the base class for all other writer classes in this group. It is an abstract class. This class is very similar to OutputStream class in functionality.
- Methods of this class are also identical to those of the OutputStream class. The only difference is that the fundamental unit of data is character instead of byte.

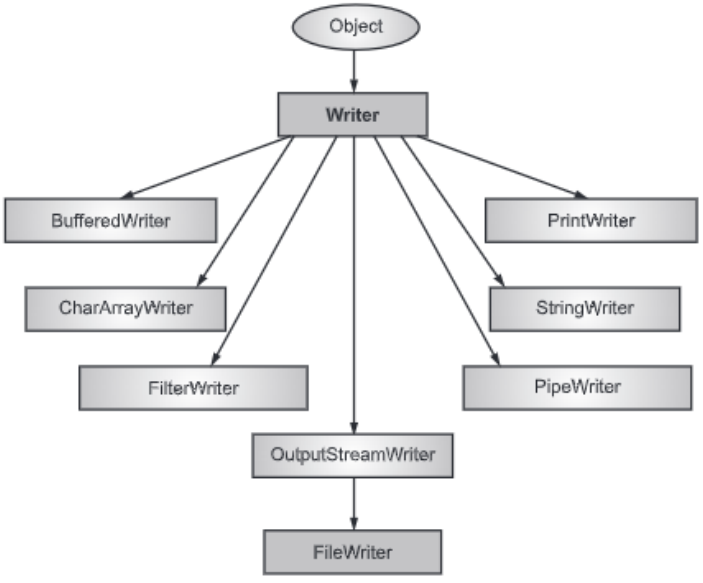


Fig. 4.5: Hierarchy of Writer Stream Classes

4.7 READERS AND WRITERS CLASS

- Input stream is similar to reader classes and output stream is similar to writer classes which are designed for unicode characters.

Writer Class functions:

- **`abstract void flush()`:** This method flushes the output steam by forcing out buffered bytes to be written out.
- **`void write(int c)`:** This method writes a characters to the output stream.
- **`void write(char[] arr)`:** This method writes a whole char array to the output stream.
- **`abstract void close()`:** This method closes this output stream and also frees any resources connected with this output stream.

Reader Class functions:

- **`int read()`:** This method reads a characters from the input stream.
- **`int read(char[] ch)`:** This method reads a chunk of characters from the input stream and store them in its char array, `ch`.
- **`close()`:** This method closes this output stream and also frees any system resources connected with it.

4.8 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 4.4.

Table 4.4: Methods of File Class [S-19]

Method	Description
`String getName()`	Returns name of the file.
`String getParent()`	Returns name of parent directory.
`long length()`	Returns length of the file.

contd. ...

`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.
`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 getAbsolutePath()`	Returns absolute path of the file.
`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 deleteOnExit()`	Deletes the file when the invoking object goes out of scope. You will find this method useful for working with temporary file.
`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.
`boolean mkdirs()`	Creates a directory and its parent directories in the path. Returns true if the creation is successful.

Program 4.10: 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
```

4.9 FILE INPUT STREAM, FILE OUTPUT STREAM

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.
  - fileObject refers to the object that describes the file.

Program 4.11: 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 4.12: 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
```

4.10 INPUT STREAM READER AND OUTPUT STREAM WRITER CLASS

- An InputStreamReader is a bridge from byte streams to character streams: It bytes are read and decoded into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Program 4.13: Program using InputStreamReader.

```
import java.io.InputStreamReader;
import java.io.FileInputStream;
public class Main
{
    public static void main(String[] args)
```

```
    {
        char[] arr = new char[50];
        try
        {
            FileInputStream f1 = new FileInputStream("temp.txt");
            InputStreamReader i1 = new InputStreamReader(f1);

            i1.read(arr);
            System.out.println("\n Temp text file..... ");
            System.out.println(arr);
            i1.close();
        }
        catch(Exception e)
        {
            System.out.println("\n Inside Catch Block.... ");
            e.getStackTrace();
        }
    }
}
```

Output:

```
Inside Catch Block....
```

OutputStreamWriter:

- It works with other output streams works as bridge between byte streams and character streams. It converts its characters into bytes.

Program 4.14: Program using OutputStreamWriter.

```
import java.io.OutputStreamWriter;
import java.io.FileOutputStream;
public class Main
{
    public static void main(String args[])
    {
        String s1 = "aaa bbb ccc ddd eee";
        try
        {
            FileOutputStream f1 = new FileOutputStream("temp.txt");
            OutputStreamWriter o1 = new OutputStreamWriter(f1);
            o1.write(s1);
            o1.close();
        }
```

```
        catch (Exception e)
        {
            System.out.println("\n Catch Block called .... ");
            e.getStackTrace();
        }
    }
}
```

Output:

```
aaa bbb ccc ddd eee
```

4.11 FileReader AND FileWriter CLASS

- 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.
  - o `FileReader(String fPath);`
  - o `FileReader(File fileObject);`
- Both these forms of constructors throw FileNotFound Exception.

Program 4.15: 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.

Program 4.16: Program using two file stream classes to copy the contents of one file into the other file..

```
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 4.17: Program 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 .....
```

4.12 BufferedReader CLASS

- 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 4.18: 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
Shruti
Welcome Shruti
Enter your Roll Number
101
Roll number is 101
Enter your Percentage
78
Percentage is 78.0
```

Program 4.19: Write a Java program to read the content of Imp.text. Display the content of file in encoded format. (Use Command Line Argument). **[W-18]**

```
import java.io.UnsupportedEncodingException;
import java.io.Writer;
import java.nio.charset.StandardCharsets;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
public class Encod
{
   public static void main(String[] args)
   {
      try
      {
         File f1 = new File("temp.txt");
         Writer w1 = new BufferedWriter(new OutputStreamWriter
                     (new FileOutputStream(f1), StandardCharsets.UTF_8));
         w1.append("\n Make a Great Day Ahead ......").append("\r\n");
         w1.append("UTF-8 Demo").append("\r\n");
         w1.append("\n Happy Cycling .......").append("\r\n");
         w1.flush();
         w1.close();
      }
      catch (UnsupportedEncodingException e)
      {
         System.out.println(e.getMessage());
      }
      catch (IOException e)
      {
         System.out.println(e.getMessage());
      }
      catch (Exception e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

Output:

```
temp.txt ->
Make a Great Day Ahead ......
UTF-8 Demo
Happy Cycling .......
```

Program 4.20: Write a Java program to accept string from user, if its length is less than 7, then throw user defined exception "Invalid String" otherwise display the string in upper case. **[W-18]**

```
import java .io.*;
class InvalidString extends Exception
{
   String str;
   InvalidString(String s)
   {
      this.str=s;
   }
   public String toString()
   {
      return "Invalid String !";
   }
}
class MyString
{
   String str1;
   void accept(String s)
   {
      str1=s;
   }
   void display()
   {
      System.out.println("");
   }
}
class Temp
{
   public static void main(String arg[]) throws Exception
   {
      System.out.println("Enter String  :");
      BufferedReader br = new BufferedReader
                                        (newInputStreamReader(System.in));
      String str=br.readLine();
```

```
        try
        {
           if(str.length()<7)
           throw new InvalidString(str);
           else
               str=str.toUpperCase();
               System.out.println("String in uppercase :"+str);
        }
        catch (InvalidString s)
        {
           System.out.println("Invalid String");
        }
    }
    }
```

Output:

```
Enter String :
fabulous
String in uppercase :FABULOUS
Enter String :
Rutu
Invalid String
```

Program 4.21: Write a java program to accept email address of a user and throw a user defined exception "Invalid Email Exception" if it does not contain '@' symbol. **[S-19]**

```
import java .io.*;
class InvalidEmail extends Exception
{
   String str;
   InvalidEmail(String s)
   {
      this.str=s;
   }
   public String toString()
   {
      return "Invalid Email address as it does not contain @ !";
   }
}
class MyEmail
{
   String str1;
```

```
    void accept(String s)
    {
        str1=s;
    }
    void display()
    {
        System.out.println("");
    }
}
class EMain
{
    public static void main(String arg[]) throws Exception
    {
        System.out.println("Enter Emailaddress  :");
        BufferedReader br = new BufferedReader
                                        (new InputStreamReader(System.in));
        String str=br.readLine();
        try
        {
            if(str.contains("@"))
            System.out.println("Email address is Valid");
            else
            throw new InvalidEmail(str);
        }
        catch (InvalidEmail s)
        {
            System.out.println("Invalid Email");
        }
    }
}
```

Output:

```
Enter Emailaddress :
vedant123@gmail.com
Email address is Valid
java EMain
Enter Emailaddress :
adi.g.com
Invalid Email
```

Program 4.22: Write a java program to display the content of file in reverse order. [S-19]

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
public class ReverseFile
{
   public static void main(String[] args) throws IOException
   {
      try
      {
         String in = "input.txt";
         String out = "output.txt";
         File sourceFile=new File(in);
         Scanner content=new Scanner(sourceFile);
         PrintWriter pwriter =new PrintWriter(out);
         while(content.hasNextLine())
         {
            String s=content.nextLine();
            StringBuffer buffer = new StringBuffer(s);
            buffer=buffer.reverse();
            String ans=buffer.toString();
            pwriter.println(ans);
         }
         content.close();
         pwriter.close();
         System.out.println("File is copied successfully!");
      }
       catch(Exception e)
      {
         System.out.println("Something went wrong");
      }
   }
}
```

Output:

```
input.txt :
Hello We are learning Reversing the file contents
output.txt(reversed)
stnetnoc elif eht gnisreveR gninrael era eW olleH
```

Summary

- 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.
- All exception types are subclasses of the built-in class Throwable. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by Exception. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- It is used during program development to create an assertion, which is a condition that should be true during the execution of the program.
- We can make use of variables and arrays for storing data in the program. But this data will be lost if variable goes out of scope or if the program itself is terminated.
- Java uses a concept of streams to represent ordered sequence of data. A stream in java is a path along which data flows. It is like a pipe through which water flows.
- This class is extended from FilterInputStream. This enhances the performance and also supports operations like skipping, marking and resetting.
- 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.
- Only an object that implements the serializable interface can be saved and restored by the serialization facilities. The Serializable interface defines no members.

Check Your Understanding

1. Which of these packages contain all the Java's built in exceptions?
   (a) java.io (b) java.util
   (c) java.lang (d) java.net
2. Which of these class is used to create user defined exception?
   (a) java.lang (b) Exception
   (c) RunTime (d) System
3. What will be the output of the program?

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

```
        try
        {
            badMethod();
            System.out.print("A");
        }
        catch (Exception ex)
        {
            System.out.print("B");
        }
        finally
        {
            System.out.print("C");
        }
        System.out.print("D");
    }
    public static void badMethod() {}
}
```

(a) AC (b) BC
(c) ACD (d) ABCD

4. Which statement is true?
   (a) catch(X x) can catch subclasses of X where X is a subclass of Exception
   (b) The Error class is a RuntimeException
   (c) Any statement that can throw an Error must be enclosed in a try block
   (d) Any statement that can throw an Exception must be enclosed in try block
5. When does Exceptions in Java arises in code sequence?
   (a) Run Time (b) Compilation Time
   (c) Can Occur Any Time (d) None of the mentioned
6. Which of these keywords must be used to monitor for exceptions?
   (a) try (b) finally
   (c) throw (d) catch
7. Which Java package must be imported when you use BufferedReader?
   (a) java.util package (b) javax.awt package
   (c) java.io package (d) java.swing package
8. For reading strings and characters in Java from console which class is used from following:
   (a) OutputStreamReader (b) FileReader
   (c) BufferedReader (d) BufferedInputReader

9. FileReader class is a subclass of _____ class.
   (a) InputStreamReader (b) FileInputStream
   (c) OutputStreamReader (d) BufferedReader
10. FileWriter class is _____ of OutputStreamWriter class.
   (a) Superclass (b) Subclass
   (c) Inputclass (d) Readerclass
11. FileReader class uses which method to read characters from a file
   (a) scanf() (b) getch()
   (c) read() (d) print()
12. Java Stream classes are categorized into two groups:
   (a) Stream and Integer Classes
   (b) String and Character Stream Classes
   (c) FileReader and FileWriter Classes
   (d) Byte and Character Stream Classes
13. Which of these classes are used by Byte streams for input and output operation?
   (a) InputStream (b) InputOutputStream
   (c) Reader (d) All of the mentioned

Answers

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

Practice Questions

Q.I Answer the following Questions in short:

1. What is a stream?
2. Which are the predefined streams?
3. Describe the term file I/O basic
4. What is Error?
5. When is Exception raised?
6. What is FileReader and FileWriter?
7. When is BufferedReader used?
8. What are different types of Erros?
9. When is FileInputStream and FileOutputStream used?

Q.II Answer the following Questions:

1. What is an exception? In which package exceptions are defined?
2. Can user program throw own exceptions? How?
3. Explain the purpose of throws clause.
4. What is the purpose of finally block in the program?
5. Explain how nested try blocks are handled?

6. Describe file handling in brief.
7. Explain various file operation in brief.
8. How to read/write characters in a file? Explain with example.
9. How to read/write bytes in a file?
10. Write a note on Bytestream.
11. Explain Character stream in detail.
12. Write short note on:
    (i) File I/O basics
    (ii) Object serialization

Q.III Define the terms:

1. throws	2. try	3. catch
4. finally	5. ByteStream	6. File
7. Stream	8. Runtime Error	9. Syntax error
10. Logic Error	11. Exception	12. Error
13. FileInputStream	14. FileOutputStream	

Previous Exam Questions

Winter 2018

1. Give syntax of two methods used with InputStream class. **[2 M]**

Ans. Refer to Section 4.6.1.

2. Write a Java program to read the content of Imp.text. Display the content of file in encoded format. (Use Command Line Argument). **[4 M]**

Ans. Refer to Program 4.19.

3. Write a Java program to accept string from user, if its length is less than 7, then throw user defined exception "Invalid String" otherwise display the string in upper case. **[4 M]**

Ans. Refer to Program 4.20.

Summer 2019

1. State the purpose of throw keyword. **[2 M]**

Ans. Refer to Section 4.2.

2. What is Exception? Explain its keyword with example. **[4 M]**

Ans. Refer to Sections 4.1 and 4.2.

3. Write a java program to accept email address of a user and throw a user defined exception "Invalid Email Exception" if it does not contain '@' symbol. **[4 M]**

Ans. Refer to Program 4.21.

4. Write a java program to display the content of file in reverse order. **[4 M]**

Ans. Refer to Program 4.22.

5. Explain any four methods of file class. **[4 M]**

Ans. Refer to Section 4.8.

■■■

5...

Applet, AWT, Event and Swing Programming

Learning Objectives...

- To learn Basic Concepts of Swing
- To write Program using Swing.
- To learn Applet Programming.
- To study AWT in Java.

5.1 APPLET INTRODUCTION [S-19]

- Applets are small Java programs that are mainly used in Internet Applications. They can be transported over the Internet and can be executed using any web browser or an appletviewer.

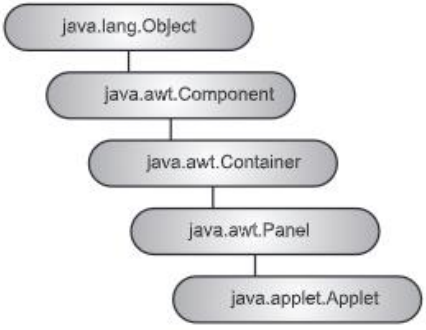


Fig. 5.1: Class JApplet

- All Applets are subclasses of Applet class either directly or indirectly. An Applet is a subclass of Panel and Panel is itself a subclass of Container. The Fig. 5.1 shows hierarchy for Applet class.
- Applets which use Swing classes for GUI are inherited from class JApplet. But JApplet inherits Applet so all the features of Applet are also available to JApplet.

- An Applet does not need to begin its execution with `main()`. Applets are not stand alone programs. They can be executed either in any web browser or in an `appletviewer` provided by JDK.

How Applet differs from Application?

- As we know that applets and stand-alone applications are Java programs, there are significant differences between them. Applets are not full-featured application programs.
- They are usually written to accomplish a small task on the Internet.
  1. Applets do not use the `main()` method for initiating the execution of the code. Applets, when loaded, automatically call certain methods of Applet class to start and execute the applet code.
  2. Unlike stand-alone applications, applets cannot be run independently. They are run from inside a Web page using a special feature known as HTML tag.
  3. Applets cannot read from or write to the files in the local computer.
  4. Applets cannot communicate with other servers on the network.
  5. Applets cannot run any program from the local computer.
  6. Applets are restricted from using libraries from other languages such as C or C++.
- Comparison between Java Applet and Java Application:

Table 5.1: Comparison between Java Applet and Java Application

Sr. No.	Java Applet	Java Application
1.	As applet is downloaded from Internet, it provides security features.	Application of java has no security provided.
2.	Applet runs in web pages.	It runs on stand alone systems.
3.	Parameters are passed by using `param` tag.	String array is used to pass parameter to main function.
4.	Execution starts from `init()` method from main method.	Execution starts from `main()` method.

5.1.1 Advantages and Disadvantages of Applets

Advantages of Applet:

1. Applets are cross platform and can run on Windows, Mac OS (Operating System) and Linux platform.
2. Applets can work all the versions of Java Plug-in.
3. Applets runs in a sandbox, so the user does not need to trust the code, so it can work without security approval.
4. Applets are supported by most web browsers.

5. Applets are cached in most web browsers, so will be quick to load when returning to a web page.
6. User can also have full access to the machine if user allows.
7. It works at client side so less response time.
8. It can be executed by browsers running under many platforms, including Linux, Windows, Mac OSs etc.
9. It is secured.

Disadvantages of Applet:

1. Java plug-in is required to run applet.
2. Java applet requires JVM so first time it takes significant startup time.
3. If applet is not already cached in the machine, it will be downloaded from internet and will take time.
4. It is difficult to design and build good user interface in applets compared to HTML technology.

5.2 TYPESAPPLET

- There are two types of applets that a web page can contain.

1. Local Applet

- Local Applet is written on our own, and then we will embed it into web pages.
- Local Applet is developed locally and stored in the local system. A web page doesn't need the get the information from the internet when it finds the local Applet in the system. It is specified or defined by the file name or pathname.
- There are two attributes used in defining an applet, i.e., the codebase that specifies the path name and code that defined the name of the file that contains Applet's code.
- **Example:** `<applet code = "FirstApplet.class" height = "500" width="500"> </applet>`

2. Remote Applet:

- When an applet is created by someone else and we can access it only when we are connecting with the internet. This type of applet is stored in remote computer and it is known as remote applet.
- We can download this remote applet from webserver to clients system using internet connection and can execute it for searching a particular applet we require to add its URL address into the applet HTML document as the value of codebase attribute.
- **Example:** `<applet codebase=http://www.abedainamdarcollege.org.in code = "MyApp.class" width=200 height=200> </applet>`

5.3 APPLET LIFE CYCLE [S-19]

- Applet can override a set of methods from the base class Applet. These methods provides the mechanism to control the execution of an applet. Applet class resides in java.applet package.
- Applet runs in the browser and its lifecycle method are called by JVM, when it is loaded and destroyed. Here, are the lifecycle methods of an Applet:
  1. `init()` : This method is called to initialized an applet.
  2. `start()` : This method is called after the initialization of the applet.
  3. `stop()` : This method can be called multiple times in the life cycle of an Applet.
  4. `destroy()` : This method is called only once in the life cycle of the applet when applet is destroyed.

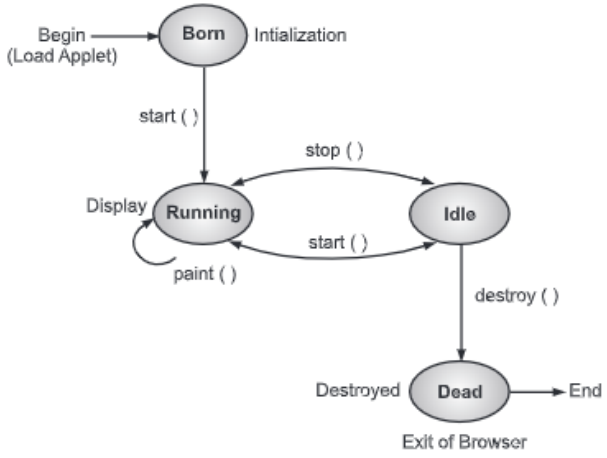


Fig. 5.2: Life Cycle of Applet

1. init () Method:

- The life cycle of an applet is begin on that time when the applet is first loaded into the browser and called the init() method.
- The init() method is called only one time in the Life cycle of an Applet.
- The init() method is basically called to read the PARAM tag in the HTML file.
- The init() method retrieve the passed parameter through the PARAM tag of HTML file using get Parameter() method.
- All the initialization such as initialization of variables and the objects like image, sound file are loaded in the init () method.
- After the initialization of the init() method user can interact with the Applet.

- **Syntax:**

```
public void init()
{
   Statements ...
}
```

2. start () Method:

- The start method of an applet is called after the initialization method init().
- This method may be called multiple times when the Applet needs to be started or restarted.
- For example, if the user wants to return to the Applet, in this situation the start Method() of an Applet will be called by the web browser and the user will be back on the applet. In the start method user can interact within the applet.
- In the start method, user can interact within the applet.
- **Syntax:**

```
public void start()
{
   Statements ...
}
```

3. stop () Method:

- The stop() method can be called multiple times in the life cycle of applet like the start() method. Or should be called at least one time.
- There is only miner difference between the start() method and stop() method. For example, the stop() method is called by the web browser on that time. When the user leaves one applet to go another applet and the start() method is called on that time when the user wants to go back into the first program or Applet.
- In the start method, user can interact within the applet.
- **Syntax:**

```
public void start()
{
   Statements ...
}
```

4. destroy() Method:

- The destroy() method is called only one time in the life cycle of Applet like init() method.
- This method is called only on that time when the browser needs to Shutdown.
- When the applet programs terminate, the destroy function is invoked which makes an applet to be in dead state.
- The destroy() method is called only one time in the life cycle of Applet like init() method.

- **Syntax:**

```
public void destroy()
{
    Statements ...
}
```

5. Display State (paint() Method):

- The applet is said to be in display state when the paint method is called. This method can be used when we want to display output in the screen.
- This method can be called any number of times. Paint() method is must in all applets when we want to draw something on the applet window. paint() method takes Graphics object as argument.
- paint() method is called each time you want to redraw the output of applet. This situation may occur for various reasons. For example, the window in which the applet is running may be overwritten by some other window and then uncovered again or the applet window may be minimized and restored again. This method takes an object of Graphics class as a parameter. This parameter will contain the graphics context. This context is used whenever the output to the applet is required.
- **Syntax:**

```
public void paint(Graphics g)
{
    Statements ...
}
```

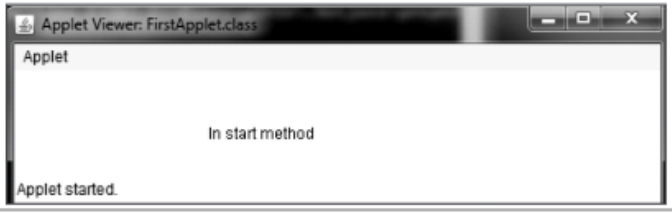
- The following is the source code for the Simple applet. This applet displays a descriptive string whenever, it encounters a major milestone in its life, such as when

Program 5.1: Program for Applet Life Cycle.

```
import java.applet.*;
import java.awt.*;
/* <applet code = "FirstApplet.class" height = "500" width="500">
</applet> */
public class  FirstApplet extends Applet
{
    String message = "";
    public void init()
    {
        message = "In init method";
        System.out.println(message);
    }
```

```
    public void start()
    {
        message = "In start method";
        System.out.println(message);
    }
    public void stop()
    {
        message = "In stop method";
        System.out.println(message);
    }
    public void destroy()
    {
        message = "In destroy method";
        System.out.println(message);
    }
    public void paint(Graphics g)
    {
        g.drawString(message, 200,200);
    }
}
```

Output:



5.3.1 Creating Applet

- An Applet is nothing but the `.class` file which is obtained by compiling the source code of an Applet. Compilation process is exactly similar as that of compiling applications.
- Once `.class` file is ready, now you need to create an HTML file to include your applet that is `.class` file. Let us assume that the name of the class file is `Hello.class`. Let us write the HTML file to include this .class file. The name of the HTML file is Hello.html.

```
<html> <head> <title> Simple Program </title> </head> <body>
<applet code = "Hello.class" width = 100 height = 25>
</applet> </body></html>
```

- Now, we have the following files in the current directory: `Hello.java`, `Hello.class` and `Hello.html`.
- To run the applet, we should have Java enabled browser or an appletviewer. We will be able to see the entire webpage if we run the applet in browser. You can use the appletviewer tool to run the applet as follows:

```
appletviewer Hello.html
```

- Please note that the file name used as argument with appletviewer is not the class file but it is the HTML file.

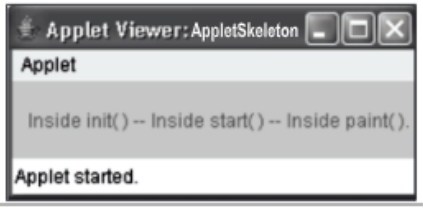
An Applet Skeleton:

- Applet extends AWT class Panel. Panel extends Container, which in turn extends Component class. All these classes support Java's window based graphical interface.
- `init()`, `start()`, `stop()` and `destroy()` apply to all applets. Default implementation for all these methods is provided. Applets do not need to override method if it is not used.

Program 5.2: Program shows the skeleton of the Applet.

```
import java.awt.*;
import java.applet.*;
/* <applet code="AppletSkeleton.class" width=300 height=100> </applet> */
public class AppletSkeleton extends Applet
{
   public void init()
   {
   }
   public void start()
   {
   }
}
}
public void paint(Graphics g)
{
}
}
```

Output:



- Although this skeleton does not do anything, it can be compiled and run. When run, it generates the above window which will be viewed in appletviewer.
- It is important to understand the order in which the methods are called. When an applet begins, methods are called in sequence like init(), start(), paint(), stop() and destroy().

5.3.2 Applet Tags

- We have used Applet tag in its simplest form. The syntax of applet tag is a bit complex. It includes several attributes. The complete tag with all its attributes is shown below:

```
<APPLET [CODEBASE = codebase_url] CODE = appletfilename.class
[ALT = alternate_text]
[NAME = appletinstance_name]
WIDTH = pixels HEIGHT = pixels [ALIGN = alignment] [VSPACE = pixels]
[HSPACE = pixels] >
[ <PARAM NAME = name1 VALUE = value1> ]
[ <PARAM NAME = name2 VALUE = value2> ]
.......
</APPLET>
```

- The following table lists all the attributes along with their meaning.

Table 5.2: Applet tag Attributes

Attribute	Meaning
CODE=AppletFileName.class	Specifies the name of the applet class to be loaded. That is, the name of the already compiled, class file in which the executable Java bytecode for the applet is stored. This attribute must be specified.
CODEBASE=codebase_URL (optional)	Specifies the URL of the directory in which the applet resides. If the applet resides is the same directory as the HTML file, then the CODEBASE attribute may be omitted entirely.
WIDTH=pixels HEIGHT=pixels	These attributes specify the width and height of the space on the HTML page that will be reserved for the applet.
NAME=applet_instance_name (Optional)	A name for the applet may optionally be specified so that other applets on the page may refer to this applet. This facilitates inter-applet communication.

contd. ...

ALIGN=alignment (Optional)	This optional attribute specifies where on the page the applet will appear. Possible values for alignment are: TOP, BOTTOM, LEFT, RIGHT, MIDDLE, ABSMIDDLE, ABSBOTTOM, TEXTTOP and BASELINE.
HSPACE=pixels (Optional)	Used only when ALIGN is to set to LEFT to RIGHT, this attribute specifies the amount of horizontal blank space the browser should leave surrounding the applet.
VSPACE=pixels (Optional)	Used only when some vertical alignment is specified with the ALIGN attribute (TOP, BOTTOM etc.) VSPACE specifies the amount of vertical blank space the browser should leave surrounding the applet.
ALT=alternate_text (Optional)	Non-Java browsers will display this text where the applet would normally go. This attribute is optional.

Program 5.3: create a small applet to accept two values from the user and display the product of it in an applet.

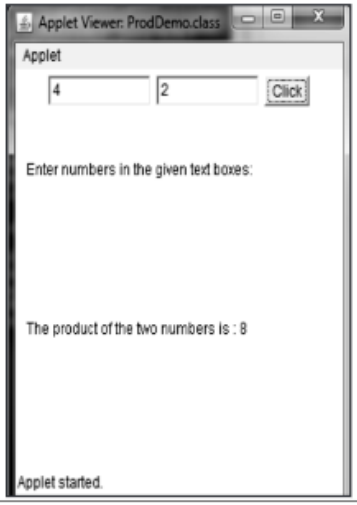
```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class ProdDemo extends Applet implements ActionListener
{
    TextField tf1, tf2;
    int x =0, y = 0, prod = 0;
    String s1, s2 , s3;
    Button b1;
    public void init()
    {
        tf1 = new TextField(10);
        tf2 = new TextField(10);
        b1 = new Button("Click");
        add(tf1);
        add(tf2);
        add(b1);
        tf1.setText("0");
        tf2.setText("0");
        b1.addActionListener(this);
    }
```

```
    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == b1)
        {
            s1 = tf1.getText();
            x = Integer.parseInt(s1);
            s2 = tf2.getText();
            y = Integer.parseInt(s2);
            prod = x * y;
            repaint();
        }
    }
    public void paint(Graphics g)
    {
        g.drawString("Enter numbers in the given text boxes: ", 10,80);
        g.drawString("The product of the two numbers is: "+ prod, 10,200);
    }
}
```

ProdDemo.html

```
<html>
<body>
<applet code = "ProdDemo.class" width = 300 height = 300>
</applet>
</body>
</html>
```

Output:



Program Analysis:

- An Applet contains two text fields and a button. User enters two values into the text fields and when the user clicks on a button, the product of the two values is calculated and displayed in an applet window.
- Class `ProdDemo` extends Applet class and it implements ActionListener interaface to handle the event carried out by a button. It declares all the required variables. Applet's `init()` method declares all the GUI controls and it designs GUI for the applet. It also adds ActionListener interface to the button.
- When the button is clicked, it calls `actionPerformed()` method to handle the action event. `actionPerformed()` method is to be overridden by the code that should be executed to handle the event. `repaint()` method calls `paint()` automatically to display the updated information in the applet.
- `paint()` method is called by an Applet which displays initial informatation. It also displays product of the two numbers entered by the user when called again by `repaint()` method.

Passing Parameters to Applet:

- Many times, when we are writing Java applets, we may need to pass parameters. We can pass these parameters from an HTML page with the help of `<param>` tag. For example, we may want to tell the applet, which color should be set as a background to the applet.
- The benefit of passing parameter from HTML lies in its portability. If you pass parameters from HTML page, they can be passed from one webpage to the other and from one website to the other. We can make our applet more flexible and portable using this technique.
- Parameters are passed by placing them in a `<PARAM>` tag and placing this tag inside opening and closing of `<APPLET>` tag.
- For Example,

```
<APPLET> <PARAM name = 'name1' value = 'value1'> </APPLET>
```

Program 5.4: Program for passing parameters to an Applet using <param> tag.

DrawApplet.java

```
import java.applet.*;
import java.awt.*;
public class ParaApplet extends Applet
{
   String name;
   int age;
   public void init()
   {
      name = getParameter("nm");
```

```
        try
        {
            age = Integer.parseInt( getParameter("input_age"));
        }
        catch(NumberFormatException e){}
     }
    public void paint(Graphics g)
    {
        Font f = new Font("Helvetica", Font.BOLD, 20);
        g.setFont(f);
        g.setColor(Color.red);
        g.drawString("Name is "+ name + " & Age is " + age, 50,25);
    }
}
```

ParaApplet.html

```
<html>
<body>
This is an Applet...
<applet code = "ParaApplet.class" width = 200 height = 200>
<param name = "nm" value = "Manisha">
<param name = "input_age" value = "50">
</applet>
</body>
</html>
```

Output:

5.4 APPLETCLASSES

- Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

5.4.1 Color

- The Color class is a part of Java Abstract Window Toolkit (AWT) package. The Color class creates color by using the given RGBA values where RGBA stands for RED, GREEN, BLUE, ALPHA or using HSB value where HSB stands for HUE, SATURATION, BRI components.

- The value for individual components RGBA ranges from 0 to 255 or 0.0 to 0.1.
- The value of alpha determines the opacity of the color, where 0 or 0.0 stands fully transparent and 255 or 1.0 stands opaque.

Constructors:

1. `Color(ColorSpace c, float[] co, float a)`: Creates a color in the specified ColorSpace with the color components specified in the float array and the specified alpha.
2. `Color(float r, float g, float b)`: Creates a opaque color with specified RGB components(values are in range 0.0 – 0.1)
3. `Color(float r, float g, float b, float a)`: Creates a color with specified RGBA components(values are in range 0.0 – 0.1)
4. `Color(int rgb)`: Creates an opaque RGB color with the specified combined RGB value consisting of the red component in bits 16-23, the green component in bits 8 – 15, and the blue component in bits 0-7.
5. `Color(int rgba, boolean b)`: Creates an color with the specified combined RGBA value consisting of the alpha component in bits 24-31, the red component in bits 16 – 23, the green component in bits 8 – 15, and the blue component in bits 0 – 7.
6. `Color(int r, int g, int b)`: Creates a opaque color with specified RGB components(values are in range 0 – 255).
7. `Color(int r, int g, int b, int a)`: Creates a color with specified RGBA components(values are in range 0 – 255).

5.4.2 Graphics

- The Graphics class is the abstract super class for all graphics contexts which allow an application to draw onto components that can be realized on various devices, or onto off-screen images as well.
- A Graphics object encapsulates all state information required for the basic rendering operations that Java supports. State information includes the following properties.
  1. The Component object on which to draw.
  2. A translation origin for rendering and clipping coordinates.
  3. The current clip.
  4. The current color.
  5. The current font.
  6. The current logical pixel operation function.
  7. The current XOR alternation color

Commonly used methods of Graphics class:

1. `public abstract void drawString(String str, int x, int y)`: Is used to draw the specified string.
2. `public void drawRect(int x, int y, int width, int height)`: Draws a rectangle with the specified width and height.
3. `public abstract void fillRect(int x, int y, int width, int height)`: It is used to fill rectangle with the default color and specified width and height.

4. `public abstract void drawOval(int x, int y, int width, int height)`: It is used to draw oval with the specified width and height.
5. `public abstract void fillOval(int x, int y, int width, int height)`: It is used to fill oval with the default color and specified width and height.
6. `public abstract void drawLine(int x1, int y1, int x2, int y2)`: It is used to draw line between the points(x1, y1) and (x2, y2).
7. `public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)`: It is used draw the specified image.
8. `public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`: It is used draw a circular or elliptical arc.
9. `public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`: It is used to fill a circular or elliptical arc.
10. `public abstract void setColor(Color c)`: It is used to set the graphics current color to the specified color.
11. `public abstract void setFont(Font font)`: It is used to set the graphics current font to the specified font.

5.4.3 Font

- Text fonts in Java are represented by instances of the `java.awt.Font` class. A Font object is constructed from a name, style identifier, and a point size. We can create a Font at any time, but it's meaningful only when applied to a particular component on a given display device.
- Font names come in three varieties: family names, face names (also called font names), and logical names. Family and font names are closely related. For example, Garamond Italic is a font name for a font whose family name is Garamond.
- A logical name is a generic name for the font family. The following logical font names should be available on all platforms:
  1. Serif (generic name for TimesRoman)
  2. SansSerif (generic name for Helvetica)
  3. Monospaced (generic name for Courier)
  4. Dialog
  5. DialogInput
- The logical font name is mapped to an actual font on the local platform. Java's fonts.properties files map the font names to the available fonts, covering as much of the Unicode character set as possible. If you request a font that doesn't exist, you get the default font.

Program 5.5: Program for Font.

```
import java.awt.*;
public class ShowFonts
{
    public static void main(String[] args)
    {
        Font[] fonts;
        fonts =
        GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts( );
        for (int i = 0; i < fonts.length; i++) {
        System.out.print(fonts[i].getFontName( ) + ": ");
        System.out.print(fonts[i].getFamily( ) + ": ");
        System.out.print(fonts[i].getName( ));
        System.out.println( );
    }
}
```

Output:

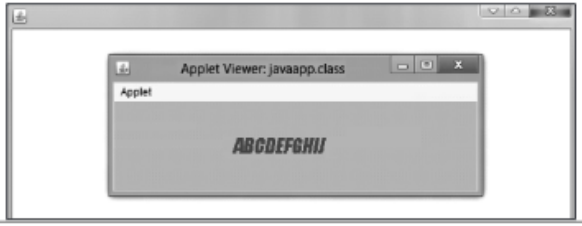
```
Bitstream Charter: Bitstream Charter: Bitstream Charter
Bitstream Charter Bold: Bitstream Charter: Bitstream Charter Bold
Bitstream Charter Bold Italic: Bitstream Charter: Bitstream Charter Bold
Italic
Bitstream Charter Italic: Bitstream Charter: Bitstream Charter Italic
Century Schoolbook L Bold: Century Schoolbook L: Century Schoolbook L Bold
(So on……)
```

Program 5.6: Write java program shows use of Font, Color & Graphics Class.

```
import java.applet.Applet;
import java.awt.Grpahics;
import java.awt.Color;
import java.awt.Font;
/*
 <applet code = "Javaapp.class" width = 300 height = 300>
 </applet>
*/
public class Javaapp extends Applet
{
 public void paint(Graphics g)
 {
```

```
    setBackground(Color.orange);
     Font f=new Font("Impact",Font.ITALIC,22);
     g.setFont(f);
     g.setColor(Color.red);
   g.drawString("ABCDEFGHIJ",100,100);
  }
 }
```

Output:



5.5 COMPONENTS AND CONTAINER USED IN AWT [W-18]

- Java AWT calls native platform (Operating systems) subroutine for creating components such as textbox, checkbox, button etc. For example an AWT GUI having a button would have a different look and feel across platforms like windows, Mac OS & Unix, this is because these platforms have different look and feel for their native buttons and AWT directly calls their native subroutine that creates the button.
- AWT Hierarchy:

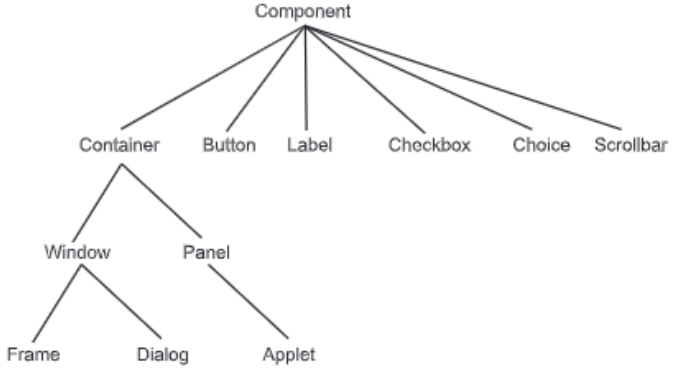


Fig. 5.3: AWT Hierarchy

Components and containers:

1. All the elements like buttons, text fields, scrollbars etc are known as components. In AWT we have classes for each component as shown in the above diagram. To have everything placed on a screen to a particular position, we have to add them to a container.

2. A container is like a screen wherein we are placing components like buttons, text fields, checkbox etc. In short, a container contains and controls the layout of components.
3. A container itself is a component (shown in the above hierarchy diagram) thus we can add a container inside container.

Types of containers:

- There are four types of containers available in AWT: Window, Frame, Dialog and Panel. As shown in the hierarchy diagram above, Frame and Dialog are subclasses of Window class.
  1. **Window:** An instance of the Window class has no border and no title
  2. **Dialog:** Dialog class has border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.
  3. **Panel:** Panel does not contain title bar, menu bar or border. It is a generic container for holding components. An instance of the Panel class provides a container to which to add components.
  4. **Frame:** A frame has title, border and menu bars. It can contain several components like buttons, text fields, scrollbars etc. This is most widely used container while developing an application in AWT.

5.6 LAYOUT MANAGERS [W-18]

- 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.
- The task of layouting the controls are done automatically by the Layout Manager. Layout manager is an object which determines the way that components are arranged in a frame window. All of the components that we have shown so far have been positioned by the default layout manager.
- A layout manager automatically arranges our controls within a window by using some type of algorithm. 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 form:
- **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 Manager:

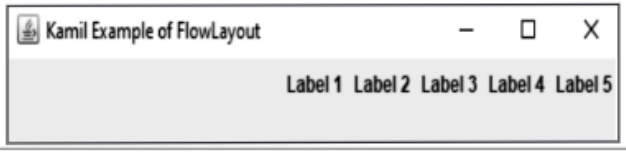
- The FlowLayout is one the most popular and simplest layout manager. FlowLayout Manager places components in a container from left to right in the order in which they were added to the container.
- When one row is filled, layout advances to the next row. It is analogous to lines of text in a paragraph. 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:
  1. **`FlowLayout()`:** This constructor centers all components and leaves five pixel spaces between each component.
  2. **`FlowLayout(int align)`:** This constructor allows to specify how each line of components is aligned i.e., `Flowlayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`.
  3. **`FlowLayout(int align, int hspace, int vspace)`:** This constructor allow to specify the alignment as well as the horizontal and vertical space between components.

Program 5.7: Program for flow layout 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: [S-19]

- 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. 5.4.

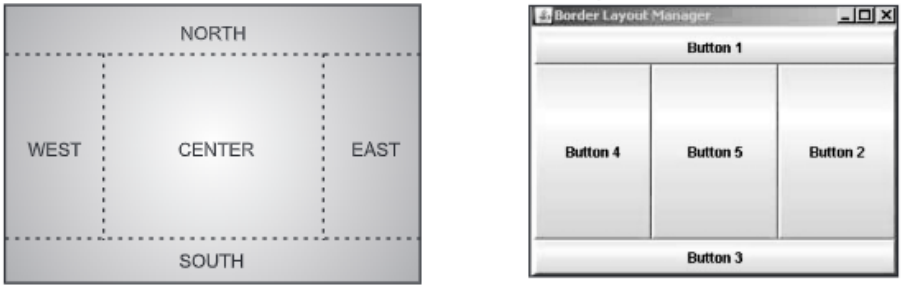


Fig. 5.4: Border Layout Manager

- The constructors defined by BorderLayout as follows:
  1. **`BorderLayout():`** This constructor creates a default border layout.
  2. **`BorderLayout(int hspace, int vspace):`** This constructor allows us to specify the horizontal and vertical space left between components in horz and vert, 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:
  1. **`BorderLayout.CENTER:`** Places component at center.
  2. **`BorderLayout.EAST:`** Places component at right border.
  3. **`BorderLayout.WEST:`** Places component at left border.
  4. **`BorderLayout.SOUTH:`** Places component at lower border.
  5. **`BorderLayout.NORTH:`** Places component at upper border.

Program 5.8: 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 places items in rows (left to right) and columns (top to bottom). The number of rows and columns is defined when the GridLayout manager is created.
- 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. 5.5: Grid Layout Manager

- The constructors supported by GridLayout are shown here:
  1. **`GridLayout()`:** Creates a single-column grid layout.
  2. **`GridLayout(int numRows, int numColumns)`:** Creates a grid layout with the specified number of rows and columns.
  3. **`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 5.9: 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:

5.7 LISTENERS AND ADAPTERCLASSES

5.7.1 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.
- 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.

Event Handling:

- An event is a signal to the program that something has happened.
- The code that performs a task in response to an event is called event handler. 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.
- Each event must return a Boolean value (true or false), indicating whether the event should be made available to other event handlers.
- An Event is an object that describes a state change in a source. It can be generated as a consequence of a user interacting with the elements in a GUI.
- Any object capable of rising an event is an event source. Event handlers are procedures that are called when a corresponding event occurs.

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.

5.7.2 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 represent 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.
- When the action event occurs, that object's actionPerformed method is invoked.
- **Method:**

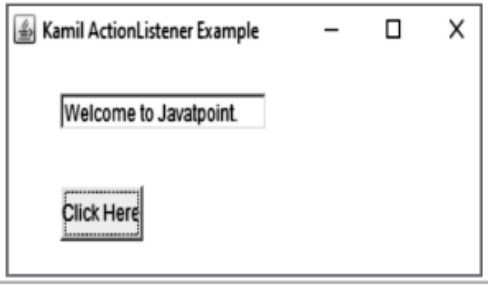
`void actionPerformed(ActionEvent e):` This method invoked when an action occurs.

Program 5.10: 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.
- When the action event occurs, that object's itemStateChanged method is invoked.
- **Method:**
  `void itemStateChanged(ItemEvent e)`: This method invoked when an item has been selected or deselected by the user.

Program 5.11: 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:

3. WindowListener:

- The class which processes the WindowEvent should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the addWindowListener() method.

- **Methods:**
  1. `void windowActivated(WindowEvent e)`: This method invoked when the Window is set to be the active Window.
  2. `void windowClosed(WindowEvent e)`: This method invoked when a window has been closed as the result of calling dispose on the window.
  3. `void windowClosing(WindowEvent e)`: This method invoked when the user attempts to close the window from the window's system menu.
  4. `void windowDeactivated(WindowEvent e)`: This method invoked when a Window is no longer the active Window.
  5. `void windowDeiconified(WindowEvent e)`: This method invoked when a window is changed from a minimized to a normal state.
  6. `void windowIconified(WindowEvent e)`: This method invoked when a window is changed from a normal to a minimized state.
  7. `void windowOpened(WindowEvent e)`: This method invoked the first time a window is made visible.

4. **KeyListener Interface:**

- The class which processes the KeyEvent should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the addKeyListener() method.
  1. `void keyPressed(KeyEvent e)`: This method invoked when a key has been pressed.
  2. `void keyReleased(KeyEvent e)`: This method invoked when a key has been released.
  3. `void keyTyped(KeyEvent e)`: This method invoked when a key has been typed.

5. **MouseListener Interface:**

- The class which processes the MouseEvent should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the addMouseListener() method.
  1. `void mouseClicked(MouseEvent e)`: This method invoked when the mouse button has been clicked (pressed and released) on a component.
  2. `void mouseEntered(MouseEvent e)`: This method invoked when the mouse enters a component.
  3. `void mouseExited(MouseEvent e)`: This method invoked when the mouse exits a component.
  4. `void mousePressed(MouseEvent e)`: This method invoked when a mouse button has been pressed on a component.
  5. `void mouseReleased(MouseEvent e)`: This method invoked when a mouse button has been released on a component.

6. MouseMotionListener Interface:

- The interface MouseMotionListener is used for receiving mouse motion events on a component.
- The class that process mouse motion events needs to implements this interface.
  1. **`void mouseDragged(MouseEvent e):`** This method invoked when a mouse button is pressed on a component and then dragged.
  2. **`void mouseMoved(MouseEvent e):`** This method invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

5.7.3 Adapterclasses [W-18]

- 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.

1. Mouse Adapter:

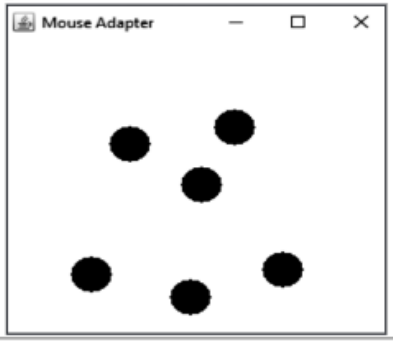
- An abstract adapter class for receiving mouse events. The methods in this class are empty. This class exists as convenience for creating listener objects.
- Mouse events let you track when a mouse is pressed, released, clicked, moved, dragged, when it enters a component, when it exits and when a mouse 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 5.12: 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:



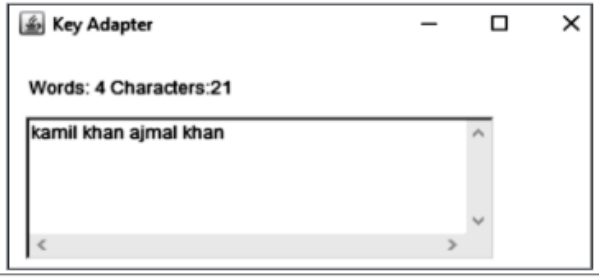
2. Key Adapter

- 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 5.13: Program 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:



5.8 EVENT DELEGATION MODEL

- The Delegation Event model is defined to handle events in GUI programming languages. The GUI stands for Graphical User Interface, where a user graphically/ visually interacts with the system.
- The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.
- Java support event processing since Java 1.0. It provides support for AWT (Abstract Window Toolkit), which is an API used to develop the Desktop application. In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override `action()` or `handleEvent()` methods. The below figure demonstrates the event processing.

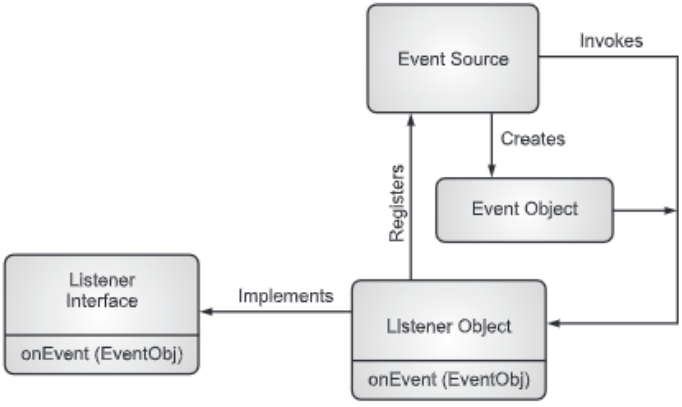


Fig. 5.6: Delegation Event model

- In the older model, an event was propagated up the containment until a component was handled. This needed components to receive events that were not processed, and it took lots of time. The Delegation Event model overcame this issue.
- Basically, an Event Model is based on the following three components:
  1. **Events:** The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.
  2. **Events Sources:** A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

3. **Events Listeners:** An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

5.9 SWING

- 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.
- 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.
- Swing is a set of classes, this provides more powerful and flexible components than are possible with the AWT.
- 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, are platform-independent. The term lightweight is used to describe such elements.
- The number of classes and interfaces in the Swing packages is substantial. Swing is the set of packages built on top of the AWT that provide us with a great number of pre-built classes that is, over 250 classes and 40 UI components.

5.9.1 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 applets 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.

5.9.2 Advantages and Disadvantages of Swing

Advantages:

1. Swing provides paint debugging support when you build your own component.
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.

5.9.3 Introduction to Swing Component and Container Classes

Swing Component:

- 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 their own view supported by java's look and feel 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.

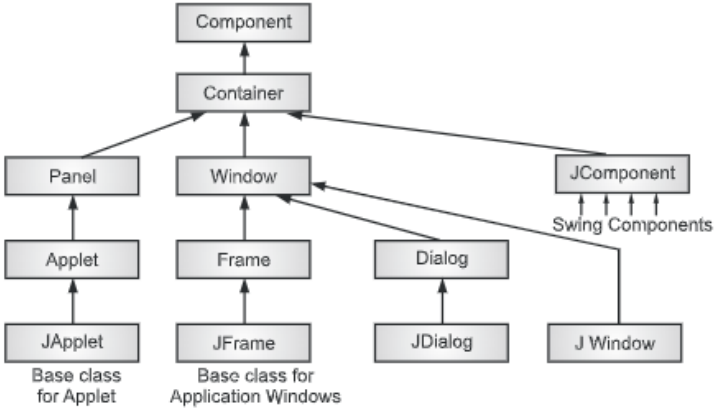


Fig. 5.7: Components of swing

Container Class:

- A Container class can be described as a special component that can hold the gathering of the components.
- There are two types of Swing Containers, they are top-level containers and low-level containers.
- Top-Level containers are heavyweight containers such as JFrame, JApplet, JWindow, and JDialog.
- Low-Level containers are lightweight containers such as JPanel.
- The most commonly used containers are JFrame, JPanel and JWindow.
- The important methods of the Container class are add(), invalidate() and validate().

5.10 EXPLORING SWING CONTROLS - JLABEL AND IMAGE ICON, JTEXT FIELD, THE SWING BUTTONS JBUTTON, JTOGGLE BUTTON, JCHECK BOX, JRADIO BUTTON, JTABBED PANE, JSCROLL PANE, JLIST, JTABLE, JCOMBOBOX, SWING MENUS, DIALOGS, JCOLORCHOOSER

5.10.1 JLabel and Image Icon

- 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 5.14: 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 5.15: Following program of demonstrate icons.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet
{
   public void init()
   {
      // Get content pane
      Container contentPane = getContentPane();
      // Create an icon
      ImageIcon ii = new ImageIcon("Lotus.gif");
      // Create a label
      JLabel jl = new JLabel("Lotus", ii, JLabel.CENTER);
      // Add label to the content pane
      contentPane.add(jl);
   }
}
```

5.10.2 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 5.16: 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:

- The above program, create a frame window in Swing. We then set its existing layout to FlowLayout. We then create a Swing label along with a text field component and add them to the content pane.

5.10.3 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 5.17: 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:

5.10.4 JToggleButton

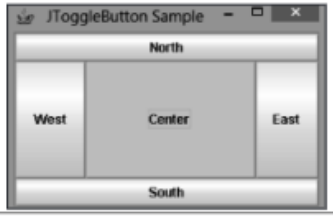
- A toggle button is two-state button that allows user to switch on and off. To create a toggle button in Swing, use JToggleButton class.
- JToggle button, when goes to the inward push state as long as the user has pressed the left mouse key. When he frees the left mouse key, the button comes to its normal state.
- Here, are the most common used constructors of the JToggleButton class:
  1. **`public JToggleButton()`:** Creates a toggle button without text and icon. The state of toggle button is not selected.
  2. **`public JToggleButton(Icon icon)`:** Creates a toggle button with icon.
  3. **`public JToggleButton(Icon icon, boolean selected)`:** Creates a toggle button with icon and initialize the state of toggle button by the boolean parameter selected.
  4. **`public JToggleButton(String text)`:** Creates a toggle button with text.
  5. **`public JToggleButton(String text, boolean selected)`:** Creates a toggle button with text and initialize the state of the toggle button.
  6. **`public JToggleButton(String text, Icon icon)`:** Creates a toggle button which displays both text and icon.
  7. **`public JToggleButton(String text, Icon icon, boolean selected)`:** Creates a toggle button which displays both text and icon. The state of toggle button can be initialized.

Program 5.18: Program to demonstrate JToggleButton.

```
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JToggleButton;
public class ToggleButtonSample
{
   public static void main(String args[])
   {
      JFrame f = new JFrame("JToggleButton Sample");
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      Container content = f.getContentPane();
      content.add(new JToggleButton("North"), BorderLayout.NORTH);
      content.add(new JToggleButton("East"), BorderLayout.EAST);
      content.add(new JToggleButton("West"), BorderLayout.WEST);
      content.add(new JToggleButton("Center"), BorderLayout.CENTER);
```

```
        content.add(new JToggleButton("South"), BorderLayout.SOUTH);
        f.setSize(300, 200);
        f.setVisible(true);
    }
}
```

Output:



5.10.5 JCheckBox

- 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 check box - 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.
- CheckBox has the following constructors:
  1. **`JCheckBox()`:** Creates an initially unselected check box button with no text, no icon.
  2. **`JCheckBox(Action a)`:** Creates a check box where properties are taken from the Action supplied.
  3. **`JCheckBox(Icon icon)`:** Creates an initially unselected check box with an icon.
  4. **`JCheckBox(Icon icon, boolean selected)`:** Creates a check box with an icon and specifies whether or not it is initially selected.
  5. **`JCheckBox(String text)`:** Creates an initially unselected check box with text.
  6. **`JCheckBox(String text, boolean selected)`:** Creates a check box with text and specifies whether or not it is initially selected.
  7. **`JCheckBox(String text, Icon icon)`:** Creates an initially unselected check box with the specified text and icon.
  8. **`JCheckBox(String text, Icon icon, boolean selected)`:** Creates a check box with text and icon, and specifies whether or not it is initially selected.

Program 5.19: Program to demonstrate JCheckBox.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

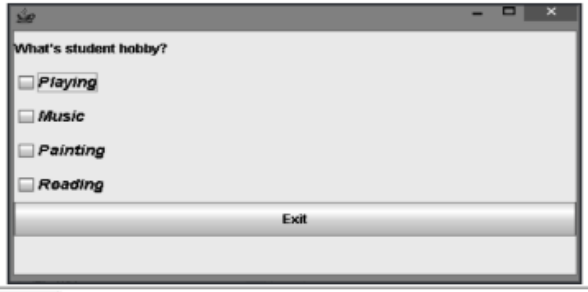
```
class HobbyOfStudent extends JPanel implements ActionListener,ItemListener
{
JCheckBox ch1 = new JCheckBox("Playing", false);
JCheckBox ch2 = new JCheckBox("Music", false);
JCheckBox ch3 = new JCheckBox("Painting", false);
JCheckBox ch4 = new JCheckBox("Reading", false);
JLabel j1 = new JLabel("What's 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);
   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);

   }
}
public void itemStateChanged(ItemEvent e)
{
   String selected= ((JCheckBox)e.getSource()).getText();
   System.out.println(selected);
}
}
```

//this is event handling, so try first without these statements

//Methods of event handling

```
class HobbyTest extends JFrame
{
    HobbyTest()
    {
        super();
        getContentPane().add(new HobbyOfStudent());
        setSize(600,600);
        setVisible(true);
    }
    public static void main(String args[]) {
    new HobbyTest();
    }
}
```

Output:



5.10.6 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 buttons 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:

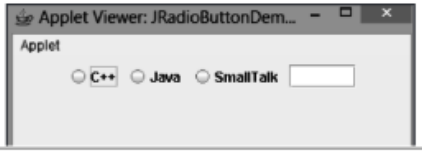
- This class is useful to group the component such as radio buttons.
- If radio button is not grouped, then they work as checkbox. And multiple selection is possible.
- To avoid that we group the Radio button object. This class provides the default constructor ButtonGroup by using this constructor object of button group class is created.
- Then different components are added to the button group object by using `add()` method.

Program 5.20: Program to demonstrate JRadioButton.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet implements
ActionListener
{
   JTextField tf;
   public void init()
   {
      Container contentPane = getContentPane();
      contentPane.setLayout(new FlowLayout());
      JRadioButton b1 = new JRadioButton("C++");
      b1.addActionListener(this);
      contentPane.add(b1);
      JRadioButton b2 = new JRadioButton("Java");
      b2.addActionListener(this);
      contentPane.add(b2);
      JRadioButton b3 = new JRadioButton("SmallTalk");
      b3.addActionListener(this);
```

```
        contentPane.add(b3);
        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);
        tf = new JTextField(5);
        contentPane.add(tf);
    }
    public void actionPerformed(ActionEvent ae)
    {
        tf.setText(ae.getActionCommand());
    }
}
```

Output:



5.10.7 JTabbed Pane

- The JTabbedPane class is used to switch between a group of components by clicking on a tab with a given title or icon. It inherits JComponent class.
- **Constructors:**
  1. **`JTabbedPane()`:** Creates an empty TabbedPane with a default tab placement of JTabbedPane.Top.
  2. **`JTabbedPane(int tabPlacement)`:** Creates an empty TabbedPane with a specified tab placement.
  3. **`JTabbedPane(int tabPlacement, int tabLayoutPolicy)`:** Creates an empty TabbedPane with a specified tab placement and tab layout policy.

Program 5.21: Program for JTabbedpane.

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

```
        JTextArea ta=new JTextArea(200,200);
        JPanel p1=new JPanel();
        p1.add(ta);
        JPanel p2=new JPanel();
        JPanel p3=new JPanel();
        JTabbedPane tp=new JTabbedPane();
        tp.setBounds(50,50,200,200);
        tp.add("main",p1);
        tp.add("visit",p2);
        tp.add("help",p3);
        f.add(tp);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String[] args)
    {
        new TabbedPaneExample();
    }
}
```

Output:

5.10.8 JScrollPane

- A scrolling pane is a container that can be used to hold any component that can be scrolled.
- By default, the list and textarea component do not scroll automatically when number of items in the list or text area component go beyond the displayed area. So to make these components scroll, you must insert them into the scroll pane.

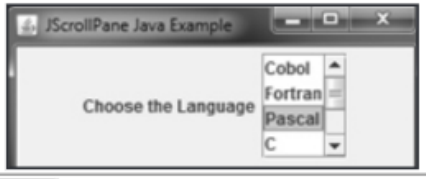
- A scroll pane is a component that presents a rectangular area in which a component may be viewed.
- Horizontal and/or vertical scroll bars may be provided if necessary.
- Scroll panes are implemented in Swing by the JScrollPane class.
- **Constructors:**
  1. **`JScrollPane(Component comp)`:** This creates a scroll panel with the component specified with comp.
  2. **`JScrollPane(int vsb, int hsb)`:** This creates a scroll panel with the vsb and hsb are int constants that define when vertical and horizontal scroll bars for this scroll panel are shown.
  3. **`JScrollPane(Component comp, int vsb, int hsb)`:** This constructor scroll panel creates a combination of above both constructor.
- Constants of scroll pane are:
  1. **HORIZONTAL_SCROLLBAR_ALWAYS:** Always provide horizontal scroll bar.
  2. **HORIZONTAL_SCROLLBAR_AS_NEEDED:** Provide horizontal scroll bar, if needed.
  3. **VERTICAL_SCROLLBAR_ALWAYS:** Always provide vertical scroll bar.
  4. **VERTICAL_SCROLLBAR_AS_NEEDED:** Provide vertical scroll bar, if needed.
- Here, are the steps that you should follow to use a scroll pane in an applet:
  1. Create a JComponent object.
  2. Create a JScrollPane object, (the arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
  3. Add the scroll pane to the content pane of the applet.

Program 5.22: Program to demonstrate JScrollPane.

```
import javax.swing.*;
import java.awt.*;
class JScrollPaneExample extends JFrame
{
   JScrollPaneExample()
   {
   setLayout(new FlowLayout());
   String[] language ={"Cobol","Fortran","Pascal","C","C++","java","C#"};
   JLabel lblLanguage = new JLabel("Choose the Language");
   JList LstMonth = new JList(language);
   LstMonth.setVisibleRowCount (4);
   JScrollPane s = newJScrollPane(LstMonth,JScrollPane.
          VERTICAL_SCROLLBAR_AWAYS,JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
   add(lblLanguage);
```

```
        add(s);
        }
    }
    class JScrollPaneJavaExample
    {
        public static void main(String[] args)
        {
            JScrollPaneExample frame = new JScrollPaneExample();
            frame.setTitle("JScrollPane Java Example");
            frame.setBounds(200,250,180,150);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
         }
     }
```

Output:



5.10.9 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);
```

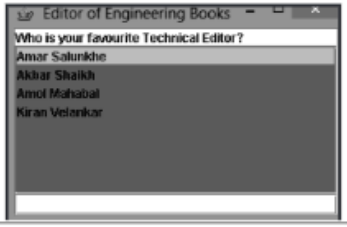
- JList does not provide any special support for handling double or triple mouse clicks. However, using the MouseListener makes it easy to handle these events. The locationToIndex() method is used to determine the cell that was checked.

Program 5.23: 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[] = {"Amar Salunkhe", "Akbar Shaikh",
   "Amol Mahabal", "Kiran Velankar"};
   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.

5.10.10 JTable

- The JTable class is a part of Java Swing Package and is generally used to display or edit two-dimensional data that is having both rows and columns. It is similar to a spreadsheet. This arranges data in a tabular form.

Constructors in JTable:

1. **`JTable():`** A table is created with empty cells.
2. **`JTable(int rows, int cols):`** Creates a table of size rows * cols.
3. **`JTable(Object[][] data, Object []Column):`** A table is created with the specified name where []Column defines the column names.

Functions in JTable:

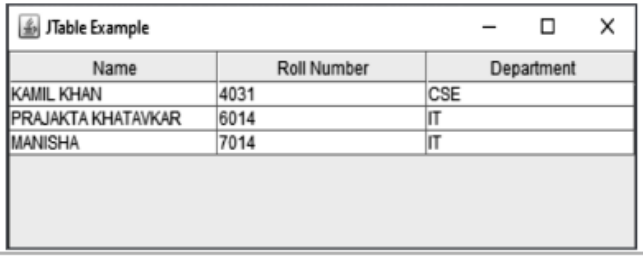
1. **`addColumn(TableColumn []column):`** Adds a column at the end of the JTable.
2. **`clearSelection():`** Selects all the selected rows and columns.
3. **`editCellAt(int row, int col):`** Edits the intersecting cell of the column number `col` and row number `row` programmatically, if the given indices are valid and the corresponding cell is editable.

4. **`setValueAt(Object value, int row, int col)`:** Sets the cell value as 'value' for the position row, col in the JTable.

Program 5.24: Program to display Jtable.

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
public class JTableExamples
{
    JFrame f;
    JTable j;
    JTableExamples()
    {
        f = new JFrame();
        f.setTitle("JTable Example");
        String[][] data =
        {
        { "KAMIL KHAN", "4031", "CSE" },
        { "PRAJAKTA KHATAVKAR", "6014", "IT" },
        { "MANISHA", "7014", "IT" }
    };
    // Column Names
    String[] columnNames = { "Name", "Roll Number", "Department" };

    // Initializing the JTable
    j = new JTable(data, columnNames);
    j.setBounds(30, 40, 200, 300);

    // adding it to JScrollPane
    JScrollPane sp = new JScrollPane(j);
    f.add(sp);
    // Frame Size
    f.setSize(500, 200);
    // Frame Visible = true
    f.setVisible(true);
}
// Driver method
public static void main(String[] args)
{
    new JTableExamples();
}
}
```

Output:



5.10.11 JComboBox

- Swing provides a combo box, (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.
- The JComboBox component, similar to Choice component in AWT, is a combination of textfield and drop down list of items. User can make selection by clicking at an item from the list or by typing into the box.
- A combo box is a combination of a list component and text field component. It can consist of more than one item, however, displays only one item at any point of time. It also allows user to type their selection. Unlike list component, combo box allows user to select only one item at a time. A combo box is an object of JComboBox class.
- Three of JComboBox's constructors are shown here:
  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 combo box 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 combo box 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 program 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 5.25: Program to demonstrate JComboBox.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
/*<APPLET CODE = ComboBoxEventsInJavaAppletSwing.class WIDTH = 320
                                                    HEIGHT = 220 ></APPLET>*/
public class ComboBoxEventsInJavaAppletSwing extends JAppletimplements Item
Listener
{
   JComboBox CmbBx = new JComboBox();
   String OutStrng = " ";
   public void init()
   {
      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);
   }
```

```
    public void itemStateChanged(ItemEvent e1)
    {
        if(e1.getStateChange() == ItemEvent.SELECTED)
        OutStrng += "Selected: " + (String)e1.getItem();
        else
        OutStrng += "DeSelected: " + (String)e1.getItem();
        showStatus(OutStrng);
    }
}
```

Output:

5.10.12 Swing Menus

- Menu components in Swing are subclasses of JComponent so they have all functionality of a normal Swing component.

Menu Hierarchy:

- As we know that every top-level window has a menu bar associated with it. This menu bar consist of various menu choices available to the end user.
- Further each choice contains list of options which is called drop down menu. Menu and MenuItem controls are subclass of MenuComponent class.

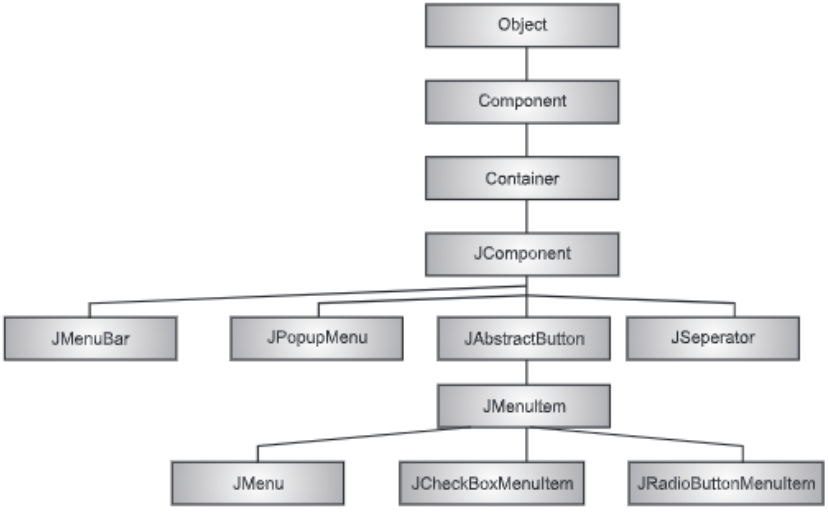


Fig. 5.8: Menu Hierarchy

Menu Controls:

1. **`JMenuBar`:** The JMenuBar object is associated with the top-level window.
2. **`JMenuItem`:** The items in the menu must belong to the JMenuItem or any of its subclass.
3. **`JMenu`:** The JMenu object is a pull-down menu component which is displayed from the menu bar.
4. **`JCheckboxMenuItem`:** JCheckboxMenuItem is subclass of JMenuItem.
5. **`JRadioButtonMenuItem`:** JRadioButtonMenuItem is subclass of JMenuItem.
6. **`JPopupMenu`:** JPopupMenu can be dynamically popped up at a specified position within a component.

5.10.12.1 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 of JMenu class:
  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 5.26: 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:

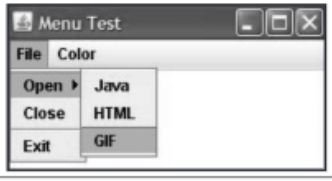
- Each menu can also have a submenu. This way we can put similar commands into groups.

Program 5.27: 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:



5.10.12.2 JMenuBar

- The JMenuBar class provides an implementation of a menu bar. JMenuBar, which is a drop down menu bar at the top application, and JPopupMenu, a menu you get when you press the button right mouse on a particular area.
- The components of a menu can be constructed using the JMenuBar, JMenu, JMenuItem, JCheckBoxMenuItem and JRadioButtonMenuItem classes in a way similar to the MenuBar, Menu and MenuItem classes of AWT.
- When anyone of the menu items in Menu object "BackGround" is selected, the background colour of the frame is changed to the corresponding colour selected in the menu item.
- In the same way, when any of the menu items in the menu for Foreground is selected, the foreground colour of the frame will change to corresponding colour selected in the menu, which is displayed in the text field. Similarly if the Exit menu item is selected from exit menu, the frame is closed.
- Following are the Constructors and methods of the JMenuBar class:

Table 5.3: JMenuBar Methods

Name of methods	Description
`JMenuBar()`	This constuctor constructs a new menu bar.
`JMenu add(Menu menu)`	This method adds the menu specified by a parameter, to the end of the menu bar.
`Component getComponentAtIndex (int index)`	This method returns the component at the specified index passed as a parameter.
`int getComponentIndex(Component c)`	This method returns the index of the component specified by the parameter.
`JMenugetMenu(int index)`	This method returns the menu at the specified index that is passed as a parameter to the function.
`int getMenuCount()`	This method returns a number of menus in the menu bar.

Program 5.28: Program for JMenuBar.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JMenuBarJavaExample extends JFrame implementsActionListener
{
    private JMenuBar MnuBar = new JMenuBar();
    private JMenu MnuOne = new JMenu("File");
```

```
private JMenu MnuTwo = new JMenu("Colors");
private JMenuItem Exit = new JMenuItem("Exit");
private JMenu bright = new JMenu("Bright");
private JMenuItem dark = new JMenuItem("Dark");
private JMenuItem white = new JMenuItem("White");
private JMenuItem pink = new JMenuItem("Pink");
private JMenuItem yellow = new JMenuItem("Yellow");
private JLabel Lbl = new JLabel("Hello");
public JMenuBarJavaExample()
{
    setTitle("Menu Bar In java Swing");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new FlowLayout());
    setJMenuBar(MnuBar);
    MnuBar.add(MnuOne);
    MnuBar.add(MnuTwo);
    MnuOne.add(xit);
    MnuTwo.add(bright);
    MnuTwo.add(dark);
    MnuTwo.add(white);
    bright.add(pink);
    bright.add(yellow);
    xit.addActionListener(this);
    dark.addActionListener(this);
    white.addActionListener(this);
    pink.addActionListener(this);
    yellow.addActionListener(this);
    add(Lbl);
    Lbl.setFont(new Font("Times New Roman", Font.BOLD,22));
}
public void actionPerformed(ActionEvent e)
{
    Object src = e.getSource();
    Container cntnr = getContentPane();
    if(src == xit)
    System.exit(0);
    else if(src == dark)
```

```
            cntnr.setBackground(Color.BLACK);
            else if(src == white)
            cntnr.setBackground(Color.WHITE);
            else if(src == pink)
            cntnr.setBackground(Color.PINK);
            else cntnr.setBackground(Color.YELLOW);
            repaint();
        }
        public static void main(String[] de)
        {
            JMenuBarJavaExample frm = newJMenuBarJavaExample();
            final int WIDTH = 260;
            final int HEIGHT = 220;
            frm.setSize(500,500);
            frm.setVisible(true);
        }
    }
```

Output:

5.10.12.3 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 of JMenuItem:**
  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 5.29: 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:

5.10.12.4 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. You then 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:

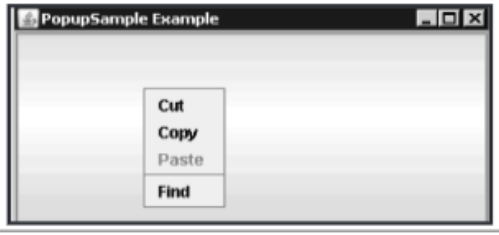
1. **`JPopupMenu()`**: This constructor constructs a JPopupMenu without an "invoker".
2. **`JPopupMenu(String label)`**: This constructor constructs a JPopupMenu with the specified title.

Program 5.30: 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:



5.10.12.5 JCheckBoxMenuItem and JRadioButtonMenuItem

JCheckboxMenuItem:

- The JCheckboxMenuItem class represents a check box which can be included in a menu.
- Selecting the check box in the menu changes control's state from on to off or from off to on.
- **Constructors:**
  1. **`JCheckboxMenuItem()`**: This constructor creates an initially unselected check box menu item with no set text or icon.
  2. **`JCheckboxMenuItem(Action a)`**: This constructor creates a menu item whose properties are taken from the Action supplied.
  3. **`JCheckboxMenuItem(Icon icon)`**: This constructor creates an initially unselected check box menu item with an icon.

4. **`JCheckboxMenuItem(String text)`**: This constructor creates an initially unselected check box menu item with text.
5. **`JCheckboxMenuItem(String text, boolean b)`**: This constructor creates a check box menu item with the specified text and selection state.
6. **`JCheckboxMenuItem(String text, Icon icon)`**: This constructor creates an initially unselected check box menu item with the specified text and icon.
7. **`JCheckboxMenuItem(String text, Icon icon, boolean b)`**: This constructor creates a check box menu item with the specified text, icon, and selection state.

JRadioButtonMenuItem:

- The JRadioButtonMenuItem class represents a check box which can be included in a menu.
- Selecting the check box in the menu changes control's state from on to off or from off to on.
- **Constructors:**
  1. **`JRadioBut tonMenuItem()`**: This constructor creates a JRadioButtonMenuItem with no set text or icon.
  2. **`JRadioButtonMenuItem(Action a)`**: This constructor creates a radio button menu item whose properties are taken from the Action supplied.
  3. **`JRadioButtonMenuItem(Icon icon)`**: This constructor creates a JRadioButtonMenuItem with an icon.
  4. **`JRadioButtonMenuItem(Icon icon, boolean selected)`**: This constructor creates a radio button menu item with the specified image and selection state, but no text.
  5. **`JRadioButtonMenuItem(String text)`**: This constructor creates a JRadioButtonMenuItem with text.
  6. **`JRadioButtonMenuItem(String text, boolean selected)`**: This constructor creates a radio button menu item with the specified text and selection state.
  7. **`JRadioButtonMenuItem(String text, Icon icon)`**: This constructor creates a radio button menu item with the specified text and Icon.
  8. **`JRadioButtonMenuItem(String text, Icon icon, boolean selected)`**: This constructor creates a radio button menu item that has the specified text, image, and selection state.

Program 5.31: Program for JCheckBoxMenuItem and JRadioButtonMenuItem.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JMenuBar2JavaExample extends JFrame
{
   private JMenuBar MnuBar = new JMenuBar();
```

```
    private JMenu MnuOne = new JMenu("File");
    private JCheckBoxMenuItem chkOne = new JCheckBoxMenuItem("Check Box
    First");
    private JCheckBoxMenuItem chkTwo = new JCheckBoxMenuItem("Check Box
    Second");
    private JRadioButtonMenuItem RdoOne = new JRadioButtonMenuItem("Radio
    Button One");
    private JRadioButtonMenuItem RdoTwo = new JRadioButtonMenuItem("Radio
    Button Two");
    private JRadioButtonMenuItem RdoThree = new J RadioButtonMenuItem("Radio
    Button Three");
    private ButtonGroup BtnGrp = new ButtonGroup();
    public JMenuBar2JavaExample()
    {
        setTitle("Menu Bar2 In java Swing");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        setJMenuBar(MnuBar);
        MnuBar.add(MnuOne);
        MnuOne.add(chkOne);
        MnuOne.add(chkTwo);
        MnuOne.addSeparator();
        MnuOne.add(RdoOne);
        MnuOne.add(RdoTwo);
        MnuOne.add(RdoThree);
        BtnGrp.add(RdoOne);
        BtnGrp.add(RdoTwo);
        BtnGrp.add(RdoThree);
    }
    public static void main(String[] ds)
    {
        JMenuBar2JavaExample frm = newJMenuBar2JavaExample();
        final int WIDTH = 160;
        final int HEIGHT = 220;
        frm.setSize(500,500);
        frm.setVisible(true);
    }
}
```

Output:

5.10.13 Dialogs

- Dialog windows or dialogs are an indispensable part of most modern GUI applications.
- A dialog is defined as a conversation between two or more persons. 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.

5.10.13.1 JDialog

- A dialog can be either modal or modeless. Modal dialogs block input to other top-level windows. Modeless dialogs allow input to other windows.
- A modal dialog blocks user input to all other windows in the same application when it is visible. You have to close a modal dialog before other windows in the same application can get focus. A modeless one does not block user input.
- 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.
  1. `JDialog (Frame parent. String title)`: This constructor creates a dialog box which have some title.

2. **`JDialog (Frame parent. String title. Boolean modal)`**: This constructor creales 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 5.32: 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());
        // Must be called before creating JDialog for
        // the desired effect
        JDialog.setDefaultLookAndFeelDecorated(true);
        // A perfect constructor, mostly used.
        // A dialog with current frame as parent
        // a given title, and modal
        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:

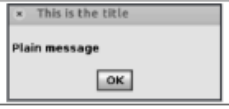
Message Dialogs:

- 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:

Table 5.4: Different kind of messages in Dialog box with symbol

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, JOptionPaneprovides showConfirmDialog() method to ask users to confirm an action and showInputDialog() to get simple input from the users.

Program 5.33: 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.

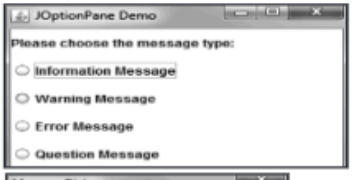
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Main
{
   public static void main(String[] args)
   {
      final JFrame frame = new JFrame("JOptionPane Demo");
      // implement ItemListener interface
      class MyItemListener implements ItemListener
      {
         public void itemStateChanged(ItemEvent ev)
         {
            boolean selected = (ev.getStateChange() == ItemEvent.SELECTED);
            AbstractButton button
                                 = (AbstractButton)ev.getItemSelectable();
            String command = button.getActionCommand();
```

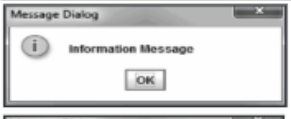
```
            if (selected)
            {
               int messageType = -1;
               String message = "";
               if (command.equals("INFORMATION"))
               {
                  messageType = JOptionPane.INFORMATION_MESSAGE;
                  message = "Information Message";
               }
               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");
```

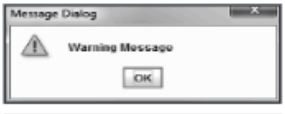
```
        // add event handler
        MyItemListener myItemListener = new MyItemListener();
        r1.addItemListener(myItemListener);
        r2.addItemListener(myItemListener);
        r3.addItemListener(myItemListener);
        r4.addItemListener(myItemListener);
        // add radio buttons to 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);
    }
}
```

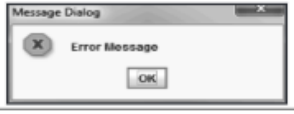
Output:

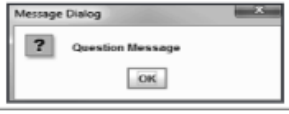
When you click on each radio button, it will display corresponding message type.











5.10.14 JColorChooser

- The class JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.

Constructors:

1. `JColorChooser()`: This constructor creates a color chooser pane with an initial color of white.
2. `JColorChooser(Color initialColor)`: This constructor creates a color chooser pane with the specified initial color.
3. `JColorChooser(ColorSelectionModel model)`: This constructor creates a color chooser pane with the specified ColorSelectionModel.

Methods:

1. `void addChooserPanel(AbstractColorChooserPanel panel)`: This method adds a color chooser panel to the color chooser.
2. `static JDialog createDialog(Component c, String title, boolean modal, JColorChooser chooserPane, ActionListener okListener, ActionListener cancelListener)`: This method creates and returns a new dialog containing the specified ColorChooser pane along with "OK", "Cancel", and "Reset" buttons.
3. `AccessibleContext getAccessibleContext()`: This method gets the AccessibleContext associated with this JColorChooser.
4. `AbstractColorChooserPanel[] getChooserPanels()`: This method returns the specified color panels.
5. `Color getColor()`: This method gets the current color value from the color chooser.
6. `boolean getDragEnabled()`: This method gets the value of the dragEnabled property.
7. `JComponent getPreviewPanel()`: This method returns the preview panel that shows a chosen color.
8. `ColorSelectionModel getSelectionModel()`: This method returns the data model that handles color selections.
9. `ColorChooserUI getUI()`: This method returns the L&F object that renders this component.
10. `String getUIClassID()`: This method returns the name of the L&F class that renders this component.
11. `protected String paramString()`: This method returns a string representation of this JColorChooser.
12. `AbstractColorChooserPanel removeChooserPanel(AbstractColorChooserPanel panel)`: This method removes the Color Panel specified.
13. `void setChooserPanels(AbstractColorChooserPanel[] panels)`: This method specifies the Color Panels used to choose a color value.

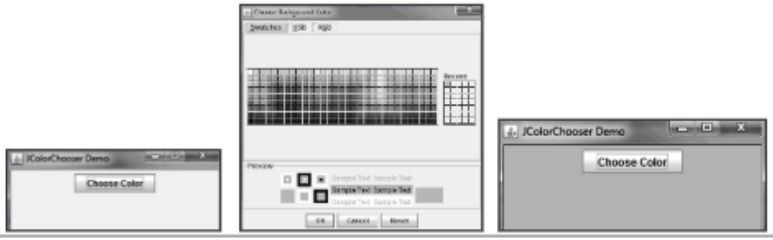
14. **`void setColor(Color color)`:** This method sets the current color of the color chooser to the specified color.
15. **`void setColor(int c)`:** This method sets the current color of the color chooser to the specified color.
16. **`void setColor(int r, int g, int b)`:** This method sets the current color of the color chooser to the specified RGB color.
17. **`void setDragEnabled(boolean b)`:** This method sets the dragEnabled property, which must be true to enable automatic drag handling (the first part of drag and drop) on this component.
18. **`void setPreviewPanel(JComponent preview)`:** This method sets the current preview panel.
19. **`void setSelectionModel(ColorSelectionModel newModel)`:** This method sets the model containing the selected color.
20. **`void setUI(ColorChooserUI ui)`:** This method sets the L&F object that renders this component.
21. **`static Color showDialog(Component component, String title, Color initialColor)`:** This method shows a modal color-chooser dialog and blocks until the dialog is hidden.
22. **`void updateUI()`:** This method shows notification from the UIManager that the L&F has changed.

Program 5.34: Program for JColorChooser.

```
package jcolorchooserdemo;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Main
{
    public static void main(String[] args)
    {
        final JFrame frame = new JFrame("JColorChooser Demo");
        JButton btn1 = new JButton("Choose Color");
        btn1.addActionListener(new ActionListener())
        {
            public void actionPerformed(ActionEvent e)
            {
                Color newColor = JColorChooser.showDialog(
                frame, "Choose Background Color", frame.getBackground());
```

```
                if(newColor != null)
                {
                    frame.getContentPane().setBackground(newColor);
                }
            }
        });
        Container pane = frame.getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(btn1);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```

Output:

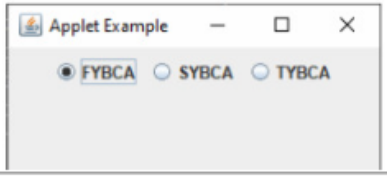


Program 5.35: Program shows the use of Radiobuttons with action listener.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
/*< applet code="TestRadioButton" width=400 height=200></applet>*/
public class TestRadioButton extends JApplet implements ActionListener
{
    public void init()
    {
        Container c=getContentPane();
        c.setLayout(new FlowLayout());
        JRadioButton fy = new JButton("FYBCA");
        JRadioButton sy = new JButton("SYBCA");
```

```
        JRadioButton ty = new JButton("TYBCA");
        c.add(fy);
        c.add(sy);
        c.add(ty);
        fy.addActionListener(this);
        sy.addActionListener(this);
        ty.addActionListener(this);
        ButtonGroup bg=new ButtonGroup();
        bg.add(fy);
        bg.add(sy);
        bg.add(ty);
    }
    public void actionPerformed(ActionEvent ae)
    {
        showStatus(ae.getActionCommand());     //display FYBCA or SYBCA
                                               // or TYBCA in status window
    }
}
```

Output:



Program 5.36: An applet which accepts username and password from the user. If the user is valid, it displays 'Welcome' message. If the user is invalid, it displays 'Invaild' message.

Login.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
public class  Login extends JApplet implements ActionListener
{
    private JLabel l1, l2, l3;
    private JButton ok, cancel;
    private JTextField t1, t2;
```

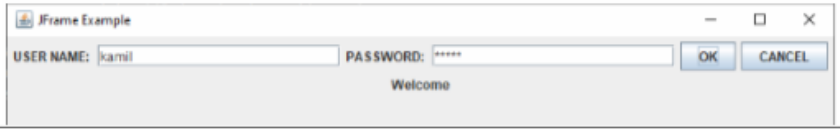
```
public void init()
{
    Container c=getContentPane();
    c.setLayout(new FlowLayout());
    l1 = new JLabel("USER NAME: ");
    l2 = new JLabel("PASSWORD: ");
    l3 = new JLabel("            ");
    t1 = new JTextField(20);
    t2 = new JTextField(20);
    t2.setEchoChar('*');
    b1 = new JButton("OK");
    b2 = new JButton("CANCEL");
    add(l1);  add(t1);  add(l2);  add(t2);
    add(b1);  add(b2);
    add(l3);
    b1.addActionListener(this);
    b2.addActionListener(this);
}
public void actionPerformed(ActionEvent ae)
{
    if(ae.getSource() == ok)
    {
        String user = t1.getText();
        String pass = t2.getText();
        String htmluser = getParameter("htmluser");
        String htmlpass = getParameter("htmlpass");
        if(user.equals(htmluser) && pass.equals(htmlpass))
            l3.setText("Welcome");
        else
        {
            l3.setText("Invalid");
            t1.setText("");
            t2.setText("");
        }
    }
```

```
        else
        {
           t1.setText("");
           t2.setText("");
           l3.setText("");
        }
     }
  }
```

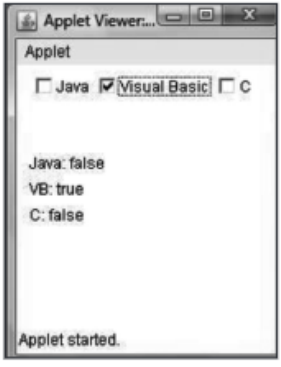
Login.html

```
<html>
<body>
<applet code = "Login.class" width = 500 height = 300>
<param name = "htmluser" value = "tybca">
<param name = "htmlpass" value = "tybca">
</applet>
</body>
</html>
```

Output:



Program 5.37: Program to use checkbox and display message as follows:



```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
/*  <APPLET  CODE  ="CheckboxAppletExample.class"  WIDTH=300  HEIGHT=200>
</APPLET> */
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code=" CheckboxAppletExample.class"
                                              height=200 width=400></applet>*/
public class CheckboxAppletExample extends Applet implements ItemListener
{
   String msg="Current Selection is:";
   Checkbox c1,c2,c3;
   CheckboxGroup cbg;
   public void init()
   {
      cbg=new CheckboxGroup();
      c1=new Checkbox("java",cbg,true);
      c2=new Checkbox("visual basic",cbg,false);
      c3=new Checkbox("C",cbg,false);
      add(c1);
      add(c2);
      add(c3);
      c1.addItemListener(this);
      c2.addItemListener(this);
      c3.addItemListener(this);
   }
   public void itemStateChanged(ItemEvent ie)
   {
      if(ie.getSource()==c1)
      {
         cbg.setSelectedCheckbox(c1);
      }
      else if(ie.getSource()==c2)
      {
         bg.setSelectedCheckbox(c2);
      }
```

```
        else
        {
            cbg.setSelectedCheckbox(c3);
        }
        repaint();
    }
    public void paint(Graphics g)
    {
        msg+=cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg,6,130);
    }
}
```

Example 5.38: To creates an applet to handle all mouse movements and print the position of mouse.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class MouseTest extends Applet implements MouseListener
{
    private int x, y = 10;
    private String s = " ";
    public void init()
    {
        addMouseListener(this);
    }
    public void paint(Graphics g)
    {
        g.drawStrings(s * "at" + "(" + x + ", "+ y + ")"; 50, 50);
    }
    private void setValues(String event, int x, int y)
    {
        s = event;
        this.x = x;
        this.y = y;
        repaint();   // This is called, whenever applet needs to update this
                     information display in its window
    }
```

```
    public void mouseClicked(MouseEvent e)
    {
        setValues("Clicked", e.getX(), e.getY());  //call method setValue
    }
    public void mousePressed(MouseEvent e)
    {
        setValues("Pressed", e.getX(), e.getY());
    }
    public void mouseReleased(MouseEvent e)
    {
        setValues("Released", e.getX(), e.getY());
    }
    public void mouseEntered(MouseEvent e)
    {
        setValues("entered", e.getX(), e.getY());
    }
    public void mouseExited(MouseEvent e)
    {
        setValues("Exited", e.getX(), e.getY());
    }
}
```

Output:

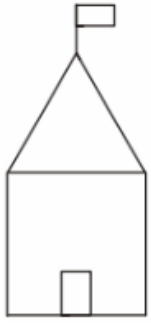
Program 5.39: Write a Java program to display "Hello World" with settings font—Times New Roman, Color–Blue, Background Color–Red on the frame. **[W-18]**

```
import java.awt.*;
class Hello extends Frame
{
    Label l;
    Hello()
    {
        l=new Label("Hello java");
```

```
        l.setFont(new Font("Georgia",Font.BOLD,14));
        l.setForeground(Color.RED);
        add(l);
        setBackground(Color.BLUE);
        setSize(300,300);
        setLayout(new FlowLayout());
        setVisible(true);
    }
    public static void main(String a[])
    {
        new Hello();
    }
}
```

Output:

Program 5.40: Write a applet application in java for designing temple. **[S-19]**



```
import java.applet.Applet;
import java.awt.*;
public class Slip10 extends Applet
{
    public void paint(Graphics g)
    {
```

```
        g.drawRect(100,150,90,120);
        g.drawRect(130,230,20,40);
        g.drawLine(150,100,100,150);
        g.drawLine(150,100,190,150);
        g.drawLine(150,50,150,100);
        g.drawRect(150,50,20,20);
    }
}
```

Summary

- Java AWT stands for Abstract Windowing Toolkit. Java Swing is a part of JFC (Java Foundation Classes) that is used to create window-based applications. It is built on the top of AWT API and entirely written in java.
- MVC stands for Model View and Controller. MVC is a design pattern that separates the business logic, presentation logic and data.
- 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 represents 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.
- Java allows us to listen for keyboard events.

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. _____ are small Java programs that are mainly used in Internet Applications.
   (a) Applets (b) Layouts
   (c) Swings (d) All of the Mentioned

Answers

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

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 do you mean by event listener?
5. How an applet is executed?
6. How events are generated?
7. What is JApplet?
8. What is a listener?
9. What is init?
10. What is component?
11. What is a panel?
12. What is a container?

Q.II Answer the following questions.

1. Explain in brief Delegation Event model for handling events.
2. Enlist the features of swing?
3. Explain in brief the fundamental idea behind MVC architecture?
4. What is difference between Applet and JApplet class?
5. What is Applet? How it is different from application?
6. Write a note on: Applet lifecycle.
7. What is the advantage of using update() method in applet?
8. Which are the various advantages of applet over java application?
9. When start() method is called?
10. To remove applet which method is used?
11. How to compile applet program?
12. Which attributes defines the dimensions of the applet?
13. What repaint method does?
14. How to retrieve parameter in applet?
15. Which swing classes are used to create menu?
16. List layout manager programs.
17. What is event? How to handle events in applets? Explain with example.

Q.III Define the terms.

1. Swing
2. AWT
3. Container
4. Applet
5. Componant
6. Event
7. Event handling

Previous Exam Questions

Winter 2018

1. What is Layout Manager? Explain any one in detail. **[4 M]**

Ans. Refer to Section 5.6.

2. What is Adapter class? Explain its purpose. **[4 M]**

Ans. Refer to Section 5.7.3.

3. Write a Java program to display "Hello World" with settings font – Times New Roman, Color–Blue, Background Color–Red on the frame. **[4 M]**

Ans. Refer to Program 5.39.

4. What is AWT? How to add any component on frame using AWT? **[4 M]**

Ans. Refer to Section 5.5.

Summer 2019

1. List any two restriction for applet. **[2 M]**

Ans. Refer to Section 5.1.

2. Which container use border layout as its default layout? **[2 M]**

Ans. Refer to Section 5.6.

3. What is Applet? Explain life cycle of Applet. **[4 M]**

Ans. Refer to Section 5.3.

4. Write a java program to accept the details of employee (Eno,Ename, Sal) from the user and display it on the next frame (Use AWT). **[4 M]**

Ans. Refer to Program 4.40.

■■■

Bibliography

1. https://www.tutorialspoint.com

2. https://www.javatpoint.com

■■■