GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.

Department of Computer Science and Engineering

HANDOUT

on

OBJECT ORIENTED PROGRAMMING THROUGH JAVA

Vision

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.

Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behavior & respect for others.
- To foster industry-academia relationship for mutual benefit and growth.

Program Educational Objectives

PEO1 : Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.

PEO2 : Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations.

PEO3 : Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the public.

HANDOUT ON OBJECT ORITENTED PROGRAMMING THROUGH JAVA

Class & Sem. :II B.Tech – I Semester Year: 2019-20

Branch : CSE Credits: 3

1.Brief History and Scope of the Subject

- The Java platform was developed at Sun in the early 1990s with the objective of allowing programs to function regardless of the device they were used on, sparking the slogan "Write once, run anywhere" (WORA). Java is regarded as being largely hardware- and operating system-independent.
- Java was initially promoted as a platform for client-side *applets* running inside web browsers. Early examples of Java applications were the Hot Java web browser and

  the Hot Java Views suite. However, since then Java has been more successful on the server side of the Internet.
- The platform consists of three major parts: the Java programming language, the Java Virtual Machine (JVM), and several Java Application Programming Interfaces (APIs).
- Java is an object-oriented programming language. Since its introduction in late 1995, it became one of the world's most popular programming languages.
- Java programs are compiled to byte code, which can be executed by any JVM, regardless of the environment.
- The Java APIs provide an extensive set of library routines. These APIs evolved into the *Standard Edition* (Java SE), which provides basic infrastructure and GUI functionality; the *Enterprise Edition* (Java EE), aimed at large software companies implementing enterprise-class application servers; and the *Micro Edition* (Java ME), used to build software for devices with limited resources, such as mobile devices.

- On November 13, 2006, Sun announced it would be licensing its Java implementation under the GNU General Public License; it released its Java compiler and JVM at that time
- Java 8 was released on 18 March 2014 and included some features that were planned for Java 7 but later deferred.

2.Pre-Requisites

Basic knowledge on programming language constructs.

3.Course Objectives:

- To familiarize with the concepts of object oriented programming
- impart the knowledge of AWT components in creation of GUI

4.Course Outcomes:

CO1 : apply object oriented approach to design software .

CO2 : create user defined interfaces and packages for a given problem

CO3 : develop code to handle exceptions.

CO4 : implement multi tasking with multi threading.

CO5 : develop applets for web applications.

CO6 : design and develop GUI programs using AWT components

5.Program Outcomes:

Graduates of the Computer Science and Engineering Program will have ability to

a. apply knowledge of computing, mathematics, science and engineering fundamentals to solve complex engineering problems.

b. formulate and analyze a problem, and define the computing requirements appropriate to its solution using basic principles of mathematics, science and computer engineering.

c. design, implement, and evaluate a computer based system, process, component, or software to meet the desired needs.

d. design and conduct experiments, perform analysis and interpretation of data and provide valid conclusions.

e. use current techniques, skills, and tools necessary for computing practice.

f. understand legal, health, security and social issues in Professional Engineering practice.

g. understand the impact of professional engineering solutions on environmental context and the need for sustainable development.

h. understand the professional and ethical responsibilities of an engineer.

i. function effectively as an individual, and as a team member/ leader in accomplishing a common goal.

j. communicate effectively, make effective presentations and write and comprehend technical reports and publications.

k. learn and adopt new technologies, and use them effectively towards continued professional development throughout the life.

l. understand engineering and management principles and their application to manage projects in the software industry.

6.Mapping of Course Outcomes with Program Outcomes:

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**
CO1	M		H									
CO2			M									
CO3	M			H								
CO4					M							
CO5					H						M	M
CO6					H						H	H

H- High Level Mapping **M-** Medium Level Mapping **L-**Low Level Mapping

7.Prescribed Text Books

a) Herbert Schildt, "Java The Complete Reference", TMH, 7th edition.

b) Sachin Malhotra, Saurabh choudhary, "Programming in JAVA", Oxford, 2nd edition.

8.Reference Text Books

a) Joyce Farrel, Ankit R.Bhavsar, "JAVA for Beginners", Cengage Learning, 4th edition.

b) Y.Daniel Liang, "Introduction to Java Programming", Pearson, 7th edition.

c) P.Radha Krishna, "Object Oriented Programming Through Java", Universities Press

9.URLs and Other E-Learning Resources

CDs :

Subject: object oriented system design

Faculty: Prof. A.K. Mazundar IIT, Kharagpur

Units : 36

Websites:

www.java.sun.com

www.roseindia.net/java

www.javabeginner.com/learn-java/introduction-to-java-programming

www.tutorialspoint.com/java/index.htm

10.Digital Learning Materials:

http://nptel.ac.in/courses/106103115/36

http://www.nptelvideos.com/video.php?id=1472

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-

introduction-to-computer-science-and-programming-fall-2008/video-lectures/lecture-14/

http://192.168.0.49/videos/videosListing/435 (our library IP)

11.Lecture Schedule / Lesson Plan

Topic	No. of Periods
UNIT-I:Fundamentals of OOP and Java	
Need of OOP	1
Principles of OOP Languages	1
Procedural Languages vs OOP	1
Java Virtual Machine	1
Java Features	1
Variables, primitive data types	1
Identifiers, keywords, literals, operators	1
Arrays, type conversion and casting	1
UNIT- II: Class Fundamentals &Inheritance	
Class Fundamentals, Declaring Objects	1
Methods, Constructors	1
this keyword	1
Overloading methods and constructors	1
access control	1
Inheritance Basics, types	1
Using super keyword	1
Method overriding, Dynamic method dispatch	1
Abstract classes, using final with inheritance	1
Object class	1
UNIT –III: Interfaces and Packages	
Interfaces: Defining an interface, Implementing interfaces	2
Nested interfaces	1

Variables in interfaces and extending interfaces	1
Packages: Defining, Creating and Accessing a Package	3
UNIT – IV: Exception Handling & Multithreading	
Exception-Handling	1
Exception handling fundamentals, uncaught exceptions	1
Using try and catch, Multiple catch clauses	1
Nested try statements, throw	1
throws, finally	1
User-defined exceptions	1
Multithreading: Introduction to multi tasking thread life cycle	2
Creating threads	1
Synchronizing threads	2
thread groups	1
UNIT – V: Applets & Event Handling	
Applets: Concepts of Applets	1
Differences between applets and applications, life cycle of an	1
Applet	1
Creating applets	1
Event Handling: Events, Event sources	1
Event classes, Event Listeners, Delegation event model	2
Handling mouse and keyboard events	2
Adapter classes	1
UNIT – VI: AWT	
The AWT class hierarchy	1
User interface components- label, button	2
Checkbox, checkboxgroup	1

Choice, list,textfield	1
Scrollbar	1
Layout managers – Flow, Border	1
Grid, Card, GridBag layout	2
Total No.of Periods:	**56**

12. Seminar Topics

- Forms of Inheritance
- AWT hierarchy
- Applet life cycle
- Menu Creation

UNIT – I

Objective:

- To get acquainted with the concepts of object-oriented programming.

Syllabus:

Need of OOP, Principles of OOP Languages, Procedural Languages vs OOP, Java Virtual Machine, Java Features.

Java Programming constructs: variables, primitive data types, identifiers, keywords, literals, operators, arrays, type conversion and casting

Learning Outcomes:

At the end of the unit, students will be able to

- Understand the principles of object oriented programming.
- Differentiate between Oriented Programming and Procedural Oriented Programming.
- Know the Evolution and Features of java.
- Understand Syntax of basic Java Programming Constructs and apply them in writing simple programs.
- Distinguish between Implicit and Explicit Casting.

LEARNING MATERIAL

➢ NEED OF OOP

Procedural Languages:

- C, PASCAL, FORTRAN languages are all procedural languages.
- Procedure oriented programming basically consists of writing a list of instructions for the computer to follow, and organizes these instructions into groups known as functions.
- Problems with Procedural languages are
    - Functions have unrestricted access to global data.

- Cannot model real world problems very well.
- Complexity increases as the length of a program increases.
- Not extensible.

Object Oriented Programming Language:

- The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach.
- In the real-world situations, we have objects which have some attributes and behavior.
- OOP can represent the real-world objects.
- Objects are defined by their unique identity, state and behavior.
- The state of an object is identified by the value of its ***attributes*** and behavior by ***methods.***
- Attributes defines the **data** for an object, as every object has some attributes. For example, attributes of an Account Holder object are Name, DoB, Acount_Number, Aadhar_number and PAN.
- Behavior is synonym to functions or methods, called to perform some task and may manipulate the attributes of an object. For example, behavior exhibited by a account holder are withDrawMoney(), checkBalance(), transferFunds().
- OOP organizes a program around its data(i.e., objects) and a set of well-defined interfaces to that data**.**
- An object-oriented program can be characterized as data controlling access to the code.
- The organization of data and function(s) in object-oriented programs is shown in the figure 1.1.

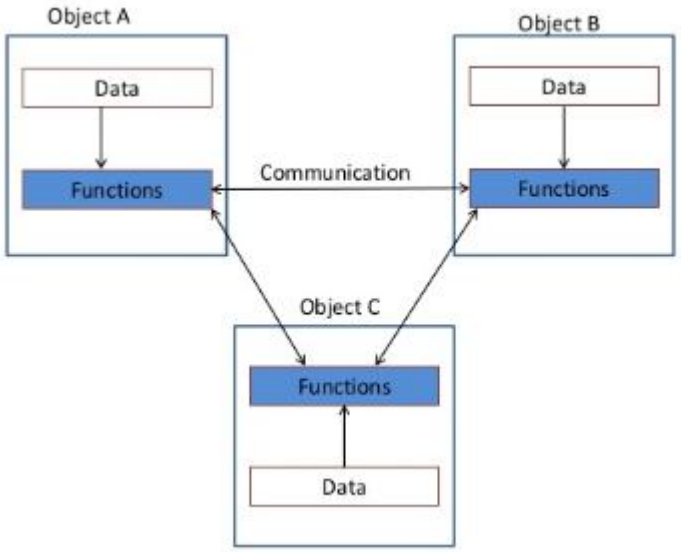


Fig 1.1: Organization of Data and Functions in OOP

- The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

➢ PRINCIPLES OF OOP LANGUAGES

The following are the general concepts of OOP

1. Objects
2. Classes
3. Abstraction
4. Encapsulation
5. Inheritance
6. Polymorphism

1. OBJECTS:

- An object is an entity in the real-world that can be distinguishable with other objects that have some attributes and exhibits some behavior.
- Objects are the ***basic run time entities*** in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.
- Objects take up space in the memory and have an associated address.

- When a program is executed, the objects interact with each other by sending messages to one another.
- For example, if "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance.
- Each object contains data and code to manipulate data.

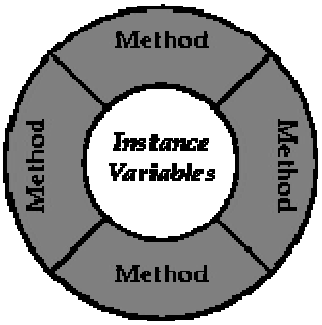


Fig 1.2: Object = Data+Methods

2. CLASSES:

- A class is defined as a blueprint of an object. It serves as a template.
- A class is a combination of common attributes and common behavior, thus we can represent class a ***collection of similar type of objects.***
- Object is an ***instance of a class.***
- The entire set of data and code of an object can be made a **user-defined data type** with the help of class. Objects are variables of the class type.
- Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created.
- For example Mango, Apple and Orange are objects from class fruit.
- Classes are user-defined data types and behave like the built-in types of a programming language.

3. ABSTRACTION:

- Abstraction refers to the "***act of representing essential features without including the background details or explanation"***.
- Classes use the concept of abstraction and are defined as a list of abstract attributes and functions operate on these attributes.
- The attributes are called data members because they hold information.
- The functions that operate on these data are called methods or member functions.

4. ENCAPSULATION:

- The process of binding together code and data it manipulates, to hide them from the outside world is called Encapsulation.
- Encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access the data.
- This insulation of the data from direct access by the program is called **data hiding** or **information hiding.**

5. INHERITANCE:

- Inheritance is the process by which ***one object acquires the properties of another object.***
- It supports the concept of hierarchical classification.
- For example the bird **robin** is a part of class '**flying bird**' which is again a part of the class '**bird**'. The principal behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in fig 1.3.

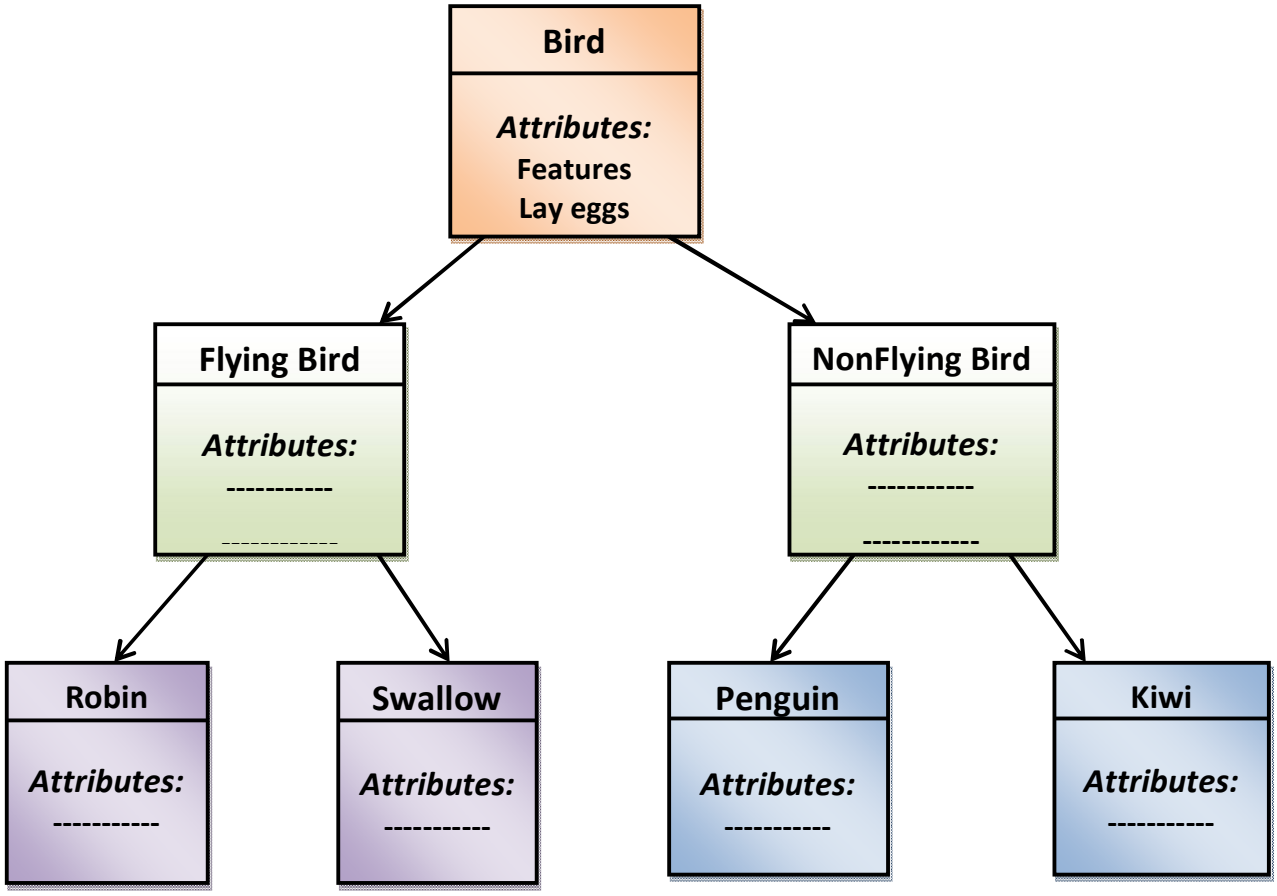


Fig 1.3: Inheritance

- In OOP, the concept of inheritance provides the idea of **reusability**. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class(sub class) from the existing class(super class). The new class will have the combined features of both the classes.

6. POLYMORPHISM:

- Polymorphism, a Greek term, means the ***ability to take more than one form***.
- For example, an operation may exhibit different behaviour at different instances. The behaviour depends upon the types of data used in the operation.

- Consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

- This is similar to polysemy (a word having different meanings depending on the context the word is used).
- The figure 1.4 illustrates that a single function name can be used to handle different number and different types of arguments.

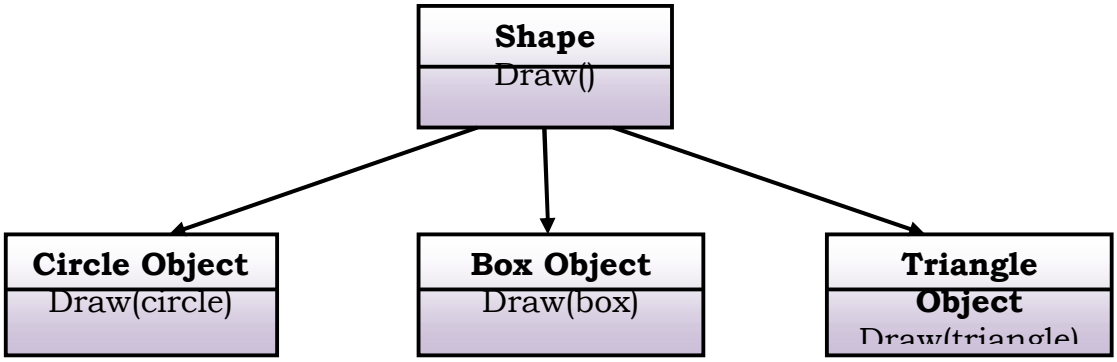


Fig 1.4: Polymorphism

➢ PROCEDURAL LANGUAGES Vs OOP:

Procedural language	Object Oriented language
Separates data from function that operate on them	Encapsulate data and methods in a class
Not suitable for defining abstract types	Suitable for defining abstract types
Debugging is difficult	Debugging is easier
Difficult to implement change	Easier to manage and implement change
Not suitable for larger applications/programs	Suitable for larger programs and applications
Analysis and design not so easy	Analysis and Design Made Easier
Faster	Slower
Less flexible	Highly flexible
Data and procedure based	Object oriented
Less reusable	More reusable

Only data and procedures are there	Inheritance, encapsulation and polymorphism are key features
Uses top down approach	Uses bottom up approach
Only a function calls another function	Object communication is there
C, Basic, FORTRAN	JAVA,C++, VB.NET, C#.NET

➢ JAVA VIRTUAL MACHINE (JVM):

- The key that allows Java to solve both the security and portability problems is that the output of a Java compiler is not executable code rather it is ***byte code.***
- Byte code is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine* (*JVM*).
- That is, in its standard form, the JVM is an *interpreter for byte code.*
- Translating a java program into byte code allows us to run a program in a wide variety of environments because only JVM details will differ from platform to platform although all understands the same java byte code.
- Just In Time (JIT) compiler is part of the JVM, which compiles selected portions of the byte code into executable code in real-time, on the fly.
- Only the sequences of byte code that will get benefit from compiling will be given to JIT, the code that requires run-time check during run-time will be given to the interpreter.

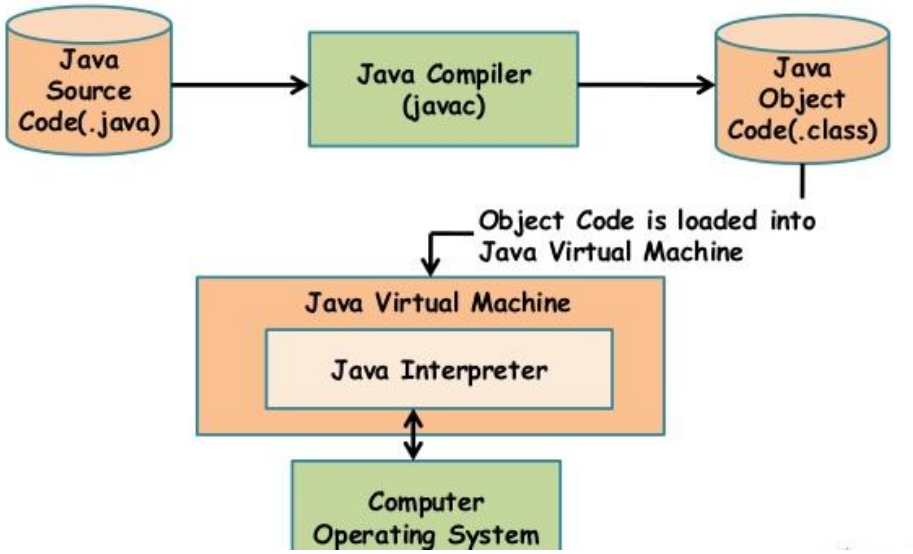


Fig 1.5: JVM

➢ JAVA FEATURES

The key considerations were summed up by the Java team in the following list of buzzwords/features:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted and High performance
- Distributed
- Dynamic

1. Simple:

- Java was designed to be easy for the professional programmer to learn and use effectively.
- Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble in learning Java.
- Many of C & C++ language features that result in unreliable code were not included in Java

2. Secure:

- Java has several language features that protect the integrity of the system and prevent several common attacks.
- Security becomes an important issue for a language that is being used on internet.
- Java provides security through lack of pointer arithmetic.
- Garbage collection makes java program more secure and robust by automatically freeing the memory.
- Has strict compile-time checking which makes java programs more robust and avoids run-time errors. The compiler also ensures that a program does not access any uninitialized variables.
- Java security model focuses on protecting users from hostile programs downloaded fromuntrusted sources across a network. Programs downloaded over the internet are executed in a ***sandbox***, cannot take any action outside of the boundaries specified by the sandbox.
- By using a Java-compatible web browser, applets can be downloaded from internet without fear of viral infection or malicious intent.

3. Portable:

- Portability is the major aspect of the internet because different types of computers and operating systems connected to it.
- The output of a Java compiler is not executable code. Rather, it is byte code.
- Translating a Java program into byte code makes it much easier to run a program in a wide variety of environments.
- Java Programs can be easily moved from one computer system to another anywhere and at anytime.

4. Object-Oriented:

- Java is pure object-oriented language, i,e., the outermost level of data structure in java is the object.

- Everything in java (constants, variables and methods) are defined inside a class and accessed through objects.
- But some constraints violate the purity of java, which was mainly designed for OOP, but with some procedural elements. For examples, java supports primitive data types that are not objects.

- **Robust:** Java is a strictly typed language, it checks the code both at compile time and runtime.
- The two of the main reasons for program failure are:
  (1) Memory Management Mistakes and
  (2) Mishandled Exceptional Conditions (that is, Run-Time errors).
- Java virtually eliminates memory Management Mistakes: Deallocation is completely automatic in Java as it provides garbage collection for unused objects.
- Java also incorporates the concept of handling the exceptional conditions that may arise in situations such as division by zero or file not found, and thus eliminates the abnormal termination of program.

5. Multithreaded:

- Java was designed to meet the real-world requirement of creating interactive, networked programs.
- To accomplish this, Java supports multithreaded programming, allows writing a program that does many independent subtasks simultaneously.
- For example, while typing in a word-processor the spell check will also does it task simultaneously.

6. Architecture-Neutral:

- One of the main problem faced by 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 make a program malfunction.

- The goal of Java designers was "**write once; run anywhere, any time, forever."** To a great extent, this goal was accomplished by bytecode, the output from java compiler.

7. Interpreted and High Performance:

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode, leads to impressive performance.
- This code can be executed on any system that implements the Java Virtual Machine.
- Also due to the incorporation of multithreading, java improved the overall execution speed of java programs.
- Java byte code was carefully designed so that it would be easy to translate it directly into native machine code for very high performance by using a just-in-time compiler.

8. Distributed:

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols
- Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.
- This enables multiple programmers at multiple remote locations can work together on a single project.

9. Dynamic:

- Java programs have run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and convenient manner.
- Java Supports functions written in other languages such as C and C++. These functions are known as Native Methods, which can be linked dynamically at Run-time.

JAVA PROGRAMMING CONSTRUCTS

➢ VARIABLES:

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable:

- In Java, all variables must be declared before they can be used. So java is termed as a *strongly typed language.*
- The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] ...] ;

- The type is one of Java's atomic types, or the name of a class or interface.
- The identifier is the name of the variable.
- To declare more than one variable of the specified type, use a comma separated list.

Examples:

```
int a, b, c;            // declares three int variables, a, b, and c.
int d = 3, e, f = 5;    // declares three int variables, initializing d and f.
byte z = 22;            // initializes z.
double pi = 3.14159;    // declares and initializes pi.
char x = 'x';           // the variable x has the value 'x'.
```

Dynamic Initialization:

- Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
- **Example:**

```
classDynInit
{
    public static void main(String args[])
    {
    double a = 3.0, b = 4.0;
```

```
        double c = Math.sqrt(a * a + b * b);
            // c is dynamically initialized
        System.out.println("Hypotenuse is " + c);
        }
    }
```

- Here, three local variables—a, b, and c—are declared. The first two a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse

The Scope and Lifetime of Variables:

- Java allows variables to be declared within any block.
- "A block begins with an opening curly brace and ends with a closing curly brace". A block defines a scope. Thus, a new scope is created each time a new block starts.
- "A scope determines what objects are visible to other parts of your program". It also determines the lifetime of those objects.
- The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.
- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code outside that scope.
- Thus, when we declare a variable within a scope, we are localizing that variable and protecting it from unauthorized access and/or modification.
- Scopes can be nested. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.
- **Example:**

```
class Scope
{
    public static void main(String args[])
    {
        int x;              // known to all code within main
        x = 10;
        if(x == 10)
```

```
        {                                   // start new scope
            int y = 20;                     // known only to this block
            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        y = 100;                            // Error! y not known here
        System.out.println("x is " + x); // x is still known here.
    }
}
```

- Within a block, variables can be declared at any point, but are valid only after they are declared.
- Thus, if you define a variable at the start of a method, it is available to all of the code within that method.
- Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.
- For example, this fragment is invalid because count cannot be used prior to its declaration:

```
// This fragment is wrong!
    count = 100; // oops! cannot use count before it is declared!
    int count;
```

- Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.
- Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.
- If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.

```
// Demonstrate lifetime of a variable.
class LifeTime
{
    public static void main(String args[])
    {
        int x;
        for(x = 0; x < 3; x++)
        {
```

```
                int y = -1;    // y is initialized each time block is entered
                System.out.println("y is: " + y); // this always prints -1
                y = 100;
                System.out.println("y is now: " + y);
            }
        }
    }
```

Output:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

y is reinitialized to –1 each time the inner for loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

- Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.
- For example, the following program is illegal:

```
// This program will not compile
    class ScopeErr
    {
    public static void main(String args[])
    {
        int bar = 1;
        {                           // creates a new scope
            int bar = 2;  // Compile-time error – bar already defined!
        }
    }
    }
```

➢ PRIMITIVE DATA TYPES

- Java defines eight primitive data types they are: **byte, short, int, long, char, float, double, and boolean.**
- The primitive types are also commonly referred to as simple types
- These can be put in four groups:

- **Integers:** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision
- **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes boolean, which is a special type for representing true/false values.

- The primitive types represent single values—not complex objects.
- Although Java is otherwise completely object-oriented, the *primitive types are not*. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.
- The following chart summarizes the default values for all the above data types.

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

1. Integers:

- Java defines four integer types: *byte, short, int, and long.* All of these are signed values.

- The width and ranges of these integer types vary widely, as shown in this table:

Name	Width(in bits)	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

a. byte:

- The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127.
- Variables of type byte are especially useful while working with
  - a stream of data from a network or file.
  - raw binary data that may not be directly compatible with Java's other built-in types.
- Byte variables are declared by use of the keyword **byte**.
- **Example:** byte b, c;

b. short:

- short is a signed 16-bit type.
- It has a range from –32,768 to 32,767. It is probably the least-used Java type.
- **Examples**: short s, t;

c. int:

- The most commonly used integer type is int.
- It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
- In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.
- **Examples**: int a, b =5;

d. long:

- long is a signed 64-bit type and is useful in cases where an int type is not large enough to hold the desired value.
- The range of a long is quite large. This makes it useful when big, whole numbers are needed.
- **Examples:** long d, s;

Example program for all Integer Types:

```
public class Demo
    {
        public static void main(String[] args)
        {
            byte b =100;
            short s =123;
            int v = 123543;
            int calc = -9876345;
            longamountVal = 1234567891;
            System.out.println("byte Value = "+ b);
            System.out.println("short Value = "+ s);
            System.out.println("int Value = "+ v);
            System.out.println("int second Value = "+ calc);
            System.out.println("long Value = "+ amountVal);
        }
    }
```

Output:

```
byte Value = 100
short Value = 123
int Value = 123543
int Second value = -9876345
long Value = 1234567891
```

2. Floating-Point Types:

- Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.
- For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.

- There are two kinds of floating-point types, ***float and double***, which represent single- and double-precision numbers, respectively.

Name	Width in Bit	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

a. float:

- The type float specifies a single-precision value that uses 32 bits of storage.
- Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small
- **Examples**: float hightemp, lowtemp;

b. double:

- Double precision, as denoted by the double keyword, uses 64 bits to store a value.
- Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. .
- All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values.
- Here is a short program that uses double variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area
{
        public static void main(String args[])
        {
                double pi, r, a;
                r = 10.8;               // radius of circle
```

```
                pi = 3.1416;            // pi, approximately
                a = pi * r * r;         // compute area
                System.out.println("Area of circle is " + a);
        }
}
```

3. Characters:

- In Java, the data type used to store characters is char.
- In C/C++, char is 8 bits wide. This is not the case in Java.
- Instead, Java uses ***Unicode*** to represent characters. "*Unicode defines a fully international character set that can represent all of the characters found in all human languages*". It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in ***Java char is a 16-bit type***.
- The range of a char is 0 to 65,535. There are no negative chars.

Example:

```
// Demonstrate char data type.
class CharDemo
{
        public static void main(String args[])
        {
                char ch1, ch2;
                ch1 = 88;           // code for X
                ch2 = 'Y';
                System.out.print("ch1 and ch2: ");
                System.out.println(ch1 + " " + ch2);
        }
}
```

Output: ch1 and ch2: X Y

- Although char is designed to hold Unicode characters, it can also be thought of as an integer type on which you can perform arithmetic operations.
- For example, you can add two characters together, or increment the value of a character variable.

- **Example:**

```
// char variables behave like integers.
class CharDemo2
{
        public static void main(String args[])
        {
                char ch1;
                ch1 = 'X';
                System.out.println("ch1 contains " + ch1);
                ch1++;                  // increment ch1
                System.out.println("ch1 is now " + ch1);
        }
}
```

Output:

```
                ch1 contains X
                ch1 is now Y
```

In the program, ch1 is first given the value X. Next, ch1 is incremented. This results in ch1 containing Y, the next character in the ASCII (and Unicode) sequence.

4. Boolean:

- Java has a primitive type, called boolean, for logical values.
- It can have only one of two possible values, true or false.
- boolean is also the type required by the conditional expressions that govern the control statements such as if and for.
- **Example:**

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[])
{
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if(b)         System.out.println("This is executed.");
        b = false;
        if(b)    System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
```

```
        }
    }
```

Output:

```
b is false
b is true
This is executed.
10 > 9 is true
```

➢ IDENTIFIERS:

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any combination of uppercase and lowercase letters, digits, or the underscore and dollar-sign characters.
- They must not begin with a digit.
- Java's reserved words keywords cannot used as identifiers.
- Java is case-sensitive, so VALUE is a different identifier than Value.
- **Examples**
  - valid identifiers are:

    AvgTemp count a4 $test this_is_ok
  - Invalid identifier names include these:

    2count high-temp Not/ok

➢ LITERALS:

- A constant value in Java is created by using a literal representation of it.
- Different types of literals those can be assigned to a variable are integer, floating-point, boolean, character and string literals.
- True, false and null are reserved literals in java.
- For example, here are some literals:

  100 98.6 'X' "This is a test"
- Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string.

- A literal can be used anywhere a value of its type is allowed.

➢ OPERATORS

- An operator performs an action on one or more operands.
- An operator that performs operation on one operand is called a *unary operator(+, -, ++, --)* , on two operands called as *binary operator(+,-,/,*,<<,>>,<,> and more)* and on 3 operands called as *ternary operator(?:).*

Java supports all the three types of operators, in addition to special operators like instanceof, .(dot), new, (type) casting operators.

1. Arithmetic Operators:

- Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+ =	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

- The operands of the arithmetic operators must be of a numeric type.
- We cannot use them on boolean types, but can use them on char types, since the char type in Java is, essentially, a subset of int.

- **Example:**

```
public class Test
{
public static void main(String args[])
  {
int a = 10;
int b = 20;
int c = 25;
int d = 25;
System.out.println("a + b = " + (a + b) );
System.out.println("a - b = " + (a - b) );
System.out.println("a * b = " + (a * b) );
System.out.println("b / a = " + (b / a) );
System.out.println("b % a = " + (b % a) );
System.out.println("c % a = " + (c % a) );
System.out.println("a++   = " +  (a++) );
System.out.println("b--   = " +  (a--) );
    // Check the difference in d++ and ++d
System.out.println("d++   = " +  (d++) );
System.out.println("++d   = " +  (++d) );
  }
}
```

Output:

```
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
```

2. The Bitwise Operators:

- In Java these will be operated on int and long values.
- If any of the operand is shorter than an int, it is automatically promoted to int before performing the operations.
- These operators act upon the individual bits of their operands.
- Negative numbers are represented in 2's complement arithmetic and then the operators are applied.

- They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

- **Example:**

```
public class Test
{
        public static void main(String args[])
        {
        int a = 60;          /* 60 = 0011 1100  in binary*/
        int b = 13;          /* 13 = 0000 1101 */
        int c = 0;

            c = a & b;              /* 12 = 0000 1100 */
        System.out.println("a & b = " + c );

            c = a | b;              /* 61 = 0011 1101 */
        System.out.println("a | b = " + c );

            c = a ^ b;              /* 49 = 0011 0001 */
        System.out.println("a ^ b = " + c );

            c = ~a;                 /*-61 = 1100 0011 */
        System.out.println("~a = " + c );

            c = a << 2;             /* 240 = 1111 0000 */
        System.out.println("a << 2 = " + c );
```

```
    c = a >> 2;                /* 15 = 1111 */
System.out.println("a >> 2  = " + c );

    c = a >>> 2;              /* 15 = 0000 1111 */
System.out.println("a >>> 2 = " + c );
  }
}
```

Output:

```
a& b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a>> 15
a>>> 15
```

3. Relational Operators:

- The relational operators determine the relationship that one operand has to the other.
- Specifically, they determine equality and ordering. The relational operators are shown here:

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

- The outcome of these operations is a boolean value.
- The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.
- Any type in Java, including integers, floating-point numbers, characters, and boolean can be compared using the equality test, ==, and the inequality test, !=.

- **Example:**

```
public class Test
{
public static void main(String args[])
  {
        int a = 10;
        int b = 20;
        System.out.println("a == b = " + (a == b) );
        System.out.println("a != b = " + (a != b) );
        System.out.println("a > b = " + (a > b) );
        System.out.println("a < b = " + (a < b) );
        System.out.println("b >= a = " + (b >= a) );
        System.out.println("b <= a = " + (b <= a) );
  }
}
```

Output:

```
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

4. Boolean Logical Operators:

- The Boolean logical operators shown here operate only on boolean operands or expressions. These operators combine two boolean values to form a resultant boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR(exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

```
// Demonstrate the boolean logical operators.
class BoolLogic
{
        public static void main(String args[])
        {
                boolean a = true;
                boolean b = false;
                boolean c = a | b;
                boolean d = a & b;
                boolean e = a ^ b;
                boolean f = (!a & b) | (a & !b);
                boolean g = !a;
                System.out.println(" a = " + a);
                System.out.println(" b = " + b);
                System.out.println(" a|b = " + c);
                System.out.println(" a&b = " + d);
                System.out.println(" a^b = " + e);
                System.out.println("!a&b|a&!b = " + f);
                System.out.println(" !a = " + g);
        }
}
```

Output:

```
a = true
b = false
a|b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false
```

5. Short-Circuit Logical Operators:

- These are secondary versions of the Boolean AND (&&) and OR (||) operators, and are known as short-circuit logical operators, conditionally evaluate the second operand or expression.

Examples:

(i) In case of AND if the first operand is false, no matter what the second operand is, the answer is false. No need to evaluate the second operand.

```
if (0 == 1 && 2 + 2 == 4)
```

```
{
        System.out.println("This line won't be printed.");
}
```

Java does the following:

1. Evaluate 0 == 1, discovering that 0 == 1 is false.
2. Realize that the condition (0 == 1 && whatever) can't possibly be true, no matter what the condition happens to be.
3. Return false (without bothering to check if 2 + 2 == 4).

(ii) In case of OR, if the first operand is true, no matter what the second operand is, the answer is true.

```
if (2 + 2 == 4 || 0 == 1)
  {
    System.out.println("This line will be printed.");
}
```

Java does the following:

1. Evaluate 2 + 2 == 4, discovering that 2 + 2 == 4 is true.
2. Realize that the condition (2 + 2 == 4 || whatever) must be true, no matter what the whatever condition happens to be.
3. Return true (without bothering to check if 0 == 1).

6. <u>The Assignment Operator:</u>

- The assignment operator is the single equal sign i.e. **=**
- It has this general form:

  var = expression;

  Here, the type of var must be compatible with the type of expression.
- The assignment operator allows you to create a chain of assignments.
- For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement.

7. The ?: Operator

- Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?:
- General form:

  expression1 ?expression2 : expression3

- Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ?operation is that of the expression evaluated.
- Both expression2 and expression3 are required to return the same type, which can't be void.
- **Example:**

```
public class Test
{
        public static void main(String args[])
        {
                int a, b;
                a = 10;
                b = (a == 1) ? 20: 30;
                        System.out.println( "Value of b is : " +  b );
                 b = (a == 10) ? 20: 30;
                System.out.println( "Value of b is : " + b );
        }
}
```

Output: Value of b is : 30
Value of b is : 20

➢ ARRAYS:

- **Definition**: An array is a memory space allocated that can store multiple values of same data type in contiguous locations.(ex., array 'marks' to represent set of marks of a group of students).
- This memory space can be accessed with a common name and a specific element is accessed by using a subscript or an index inside the brackets,

along with the name of the array.(ex., marks[5], stores marks of a fifth student),each individual value in array called as elements.

- Arrays offer a convenient means of grouping related information.
- Arrays of any type can be created and may have one or more dimensions.

ONE-DIMENSIONAL ARRAYS:

- A one-dimensional array is, essentially, a list of like-typed variables.
- **Creating Arrays:**
  - declare a variable of the desired array type.
  - allocate the memory that will hold the array, using new
  - initializing/assigning valuesinto an array.
- **Declaring:** To create an array, you first must create an array variable of the desired type.
- The general form of a one-dimensional array declaration is

typearray_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.
- **Creating memory location:** ***new*** is a special operator that allocates memory.
- The general form of **new** as it applies to one-dimensional arrays appears as follows:

Array_name = new type[size];

  - Here, type specifies the type of data being allocated,
  - size specifies the number of elements in the array,
  - andarray_name is the array variable that is linked to the array.
- **Initializing:** The elements in the array allocated by new will automatically be initialized to zero.
- To assign values to an array: **Array_name[index]=value;**

  Or **type Array_name[ ]={list of values};**

- **Example:** Allocating a 12-element array of integers and links them to month_days.

  month_days = new int[12];

  month_days will refer to an array of 12 integers and all elements in the array will be initialized to zero.
- We can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.
- The for loops can be used to assign and access values from an array.
- To obtain the number of elements in an array, use the ***length*** property associated with all the arrays in java. I.e., use array name followed by dot operator and the variable *length*.
- **Example:**

  month_days[1] = 28;

  // assigns the value 28 to the second element of month_days.

  System.out.println(month_days[3]);//displays the value stored at index 3

  Example:

```
// Demonstrate a one-dimensional array.
class Array
{
        public static void main(String args[])
        {
        intmarks[]={9,8,6,5,10};
        int n=marks.length;
        System.out.println("marks of a student in a class test are:");
        for(int i=0;i<n;i++)
                System.out.println(marks[i]);
        }
}
```

Output: marks of a student in a class test are: 9 8 6 5 10

MULTIDIMENSIONAL ARRAYS

- In Java, multidimensional arrays are actually arrays of arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.

- Suppose to store the marks of students in different subjects, need a 2D array conceptualized in the form of table, with rows representing marks of each student and columns represent the subjects.
- For example, the following declares a two dimensional array variable called marks.

int marks[][] = new int[5][6];

This allocates a 5 by 6 array and assigns it to marks, to store marks of 5 students in 6 subjects. Internally this matrix is implemented as an array of arrays of int.

- **Example:**

```
// Demonstrate a two-dimensional 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
```

Initializing Multidimensional Arrays:

- Enclose each dimension's initializer within its own set of curly braces.
- The following program creates a matrix where each element contains the product of the row and column indexes.

```
// Initialize a two-dimensional array.
class Matrix
{
        public static void main(String args[])
        {
                double m[][] = {{ 0*0, 1*0, 2*0, 3*0 },
                                { 0*1, 1*1, 2*1, 3*1 },
                                { 0*2, 1*2, 2*2, 3*2 },
                                { 0*3, 1*3, 2*3, 3*3 }};
                int i, j;
                for(i=0; i<4; i++)
                {
                        for(j=0; j<4; j++)
                                System.out.print(m[i][j] + " ");
                        System.out.println();
                }
        }
}
```

Output:

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

Alternative Array Declaration Syntax:

- There is a second form that may be used to declare an array:

  type[ ] var-name;

  Here, the square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

- This alternative declaration form offers convenience when declaring several arrays at the same time.
- For example,

  int[] num, num2, num3; // creates three array variables of type int.

➢ TYPE CONVERSION AND CASTING

- If the two types are compatible, then Java will perform the conversion automatically.
- For example, it is always possible to assign an int value to a long variable.
- However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
- For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types.
- To do so, you must use a cast, which performs an explicit conversion between incompatible types.

Java's Automatic Conversions

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.
- When these two conditions are met, a **widening conversion** takes place.
- For example, a smaller box can be placed in short, short in an int, int in long, and so on. Any value can be assigned to double. Any value except a double can be assigned to a float. Any whole number can be assigned to longand int, short, byte and char all can fit inside int.
  - byte b=10; //byte variable

- inti=b; // implicit widening byte to int

- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
- However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.
- Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

- For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int.
- For example, a bigger box has to be placed in a small box. Then the small box has to be chopped(casted) so that the bigger box (which has now become smaller) can be placed in the small box.
- This kind of conversion is sometimes called a **narrowing conversion**, and also termed as casting, since you are explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast.
- A cast is simply an explicit type conversion.
- It has this general form:

  (target-type) value

  Here, target-type specifies the desired type to convert the specified value to.
- For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;
// ...
```

```
b = (byte) a;
```

- A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.
- The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion
{
public static void main(String args[])
  {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);
        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);
        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
  }
}
```

Output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Automatic Type Promotion in Expressions

- In addition to assignments, there is another place where certain type conversions may occur: in expressions.

- In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.
- For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term a*b easily exceeds the range of either of its byte operands.

- To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression. This means that the sub expression a*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.
- As useful as the automatic promotions are, they can cause confusing compile-time errors.
- For example, this seemingly correct code causes a problem: byte b = 50;

```
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast.

The Type Promotion Rules

- Java defines several type promotion rules that apply to expressions.
- They are as follows:

1. All byte, short, and char values are promoted to int
2. If one operand is a long, the whole expression is promoted to long.
3. If one operand is a float, the entire expression is promoted to float.
4. If any of the operands is double, the result is double.

- **Example:** Demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote
{
        public static void main(String args[])
        {
                byte b = 42;
                char c = 'a';
                short s = 1024;
                int i = 50000;
                float f = 5.67f;
                double d = .1234;
                double result = (f * b) + (i / c) - (d * s);
                System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
                System.out.println("result = " + result);
        }
}
```

- **Explanation:** In the first subexpression, f * b, b is promoted to a float and the result of the subexpression is float. Next, in the subexpression i/c, c is promoted to int, and the result is of type int. Then, in d*s, the value of s is promoted to double, and the type of the subexpression is double. Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

UNIT-I
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1) Java programs are _____ [ ]

(a) Compiled (b) Interpreted

(c) Both Compiled & Interpreted (d) None of these

2) The outcome of a Java Compiler is _____ file [ ]

(a) .class (b) .obj (c) .exe (d)None of these

3) If an expression contains double, int, float, long, then whole expression will promoted into which of these data types? [ ]

(a) long (b) int (c) double (d) float

4) Which of these can be returned by the operator & . [ ]

(a) int (b)boolean (c)char (d) int or boolean

5) Consider the statement **c=a-(b*(a/b)).** Here c contains ___ [ ]

(a) Difference of a and b (b)Sum of a and b

(c) Quotient of a/b (d) Remainder of a/b

6) With x = 1, which of the following are legal lines of Java code for changing the value of x to 2 [ ]

(1) x++; (2) x=x+1; (3) x+=1; (4)x=+1

(a) 1, 2 & 3 (b) 1 & 4 (c) 1, 2, 3 & 4 (d) 3 & 2

7) What is the output of the following program? [ ]

```
class increment
{
public static void main(String args[])
{
double var1 = 1 + 5;
double var2 = var1 / 4;
int var3 = 1 + 5;
int var4 = var3 / 4;
System.out.print(var2 + " " + var4);
}
}
```

(a) 1 1 (b) 0 1 (c) 1.5 1 (d) 1.5 1.0

8) Consider the following statements

```
byte b;          // statement1
int i=100;       // statement2
b=i;             // statement3
```

Which of the above 3 statements will cause a compilation error:
(a) statement 1 (b) statement 2 (c) statement 3 (d)none

9) What is the output of the following program? [ ]

```
class conversion
{
public static void main(String args[])
{
double a = 295.04;
int  b = 300;
byte c = (byte) a;
byte d = (byte) b;
System.out.println(c + " "  + d);
}
}
```

(a) 38 43 (b) 39 44 (c) 295 300 (d) 295.04 300

10) What will this code print? [ ]

```
int arr[] = new int [5];
System.out.print(arr);
```

(a) 0 (b) value stored in arr[0] (c) 00000 (d) None

11) What is the output of this program? [ ]

```
class bitwise_operator
{
public static void main(String args[])
{
int a = 3;
int b = 6;
int c = a | b;
int d = a & b;
System.out.println(c + " "  + d);
}
}
```

(a) 7 2 (b) 7 7 (c) 7 5 (d) 5 2

12) What is the output of this program? [ ]

```
class Modulus
{
```

```
public static void main(String args[])
{
double a = 25.64;
int  b = 25;
a = a % 10;
b = b % 10;
System.out.println(a + " "  + b);
}
}
```

(a)5.640000000000001 5 (b)5.640000000000001 5.0

(c)5 5 (d)5 5.640000000000001

13) What is the output of this program? [ ]

```
class Output
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
c++;
b++;
++a;
System.out.println(a + " " + b + " " + c);
}
}
```

(a) 3 2 4 (b)3 2 3 (c)2 3 4 (d) 3 4 4

SECTION-B

SUBJECTIVE QUESTIONS

1) Summarize the Need of OOP.
2) List and explain the Principles of OOP paradigm
3) Differentiate Procedure Oriented Programming (POP) with Object Oriented Programming (OOP).
4) List and explain the Features of java.
5) Outline the role of JVM in making Java platform independent.

6) Consider the statements below:

```
byte b;       // statement1
int a;        // statement2
a=b;          // statement3
b=a;          // statement4
```

Comment about statement 3 and statement4.

7) Write a java program to do linear search on a list of integers

8) Write a java program to check whether a given number is prime or not.

9) Write a java to multiply 2 numbers without using * operator.
[**HINT:** use the operator + and loop statement]

10) Write a java program to sort given list of integers in ascending order.

UNIT – II

CLASS FUNDAMENTALS AND INHERITANCE

Objective:

- Develop the code with the concepts of Class and Inheritance.

Syllabus:

Class Fundamentals, Declaring Objects, Methods, Constructors, This Keyword, Overloading Methods and Constructors, Access Control.

Inheritance- Basics, Types, Using Super Keyword, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Using Final With Inheritance, Object Class.

Learning Outcomes:

Students will be able to

- Describe how classes, objects, methods and Constructors are created and applied in java.
- Apply different types of Inheritance and can develop simple programs using Inheritance.
- Differentiate between Method overloading and Method Overriding
- Differentiate between Abstract methods and Concrete methods.
- Demonstrate the importance of this, super and final keywords, and will be able to distinguish between them.

LEARNING MATERIAL

➢ CLASS FUNDAMENTALS:

- A class is a blueprint or prototype that defines the variables and methods common to all objects of same kind. A class can be defined as a user-defined data type and an object as a variable of that data type that can contain data and methods that manipulates the data.
- **Ex:**

Bike
boolean kickstart boolean buttonstart int gears
accelerate() applyBrake() changeGear()

Fig. Bike class

Manufacturers produce many bikes from the same blueprint as every bike share similar characteristics. There are many objects of same kind belonging to same classes that share certain characteristics. Bikes have attributes (speed, engine capacity, number of wheels, number of gears, brakes) behaviors (braking, accelerating, slowing down and changing gears).

- A class is a ***template*** for an object, and an object is an ***instance*** of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.
- A class is declared by use of the **class** keyword.
- **Declaration of class:**

```
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
    }
}
```

- The data, or variables, defined within a class are called *instance variables.*
- The code is contained within *methods.* Collectively, the methods and variables defined within a class are called *members* of the class.
- The instance variables are directly accessible by methods defined in the class.
- Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- The data for one object is separate and unique from the data for another.

Example of class:

- Create a class structure that may represent the structure of a hospital or other medical organization, begin with a class called **Employee**.
- An employee has several characteristics that you can represent as **variables**, such as name, salary, and sickDays.
- Write a method **details()** which consists of three println() statements that print the values of instance variables. The Employee class can be defined as follows:

```
class Employee
{
    String name;     // Instance Variables
    int salary;      // Instance Variables
    void details()   // Instance Method
    {
        System.out.println("Name: " + name);
        System.out.println("Salary: " + salary);
    }
}
```

➢ CREATING OBJECTS:

- Object is an instance of a class.
- An object is created by creating an instance of a class. The type of the object is class itself.
- Creating an object for a class is a two-step process.
  - First, **declare** a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
  - Second, acquire an actual, physical copy of the object and assign it to that variable by using the **new** operator.
- The new operator dynamically allocates memory for an object and returns a reference to it. This reference is nothing but the address in memory of the object allocated by new. This reference is then stored in the variable declared.

Syntax for creating an Object:

```
Classname objectname=new Classname();
```

Example:

```
Employee sam=new Employee();
```

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

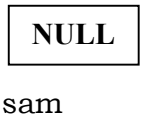
```
Employee sam;              // declare reference to object
sam=new Employee();        // allocate a Employee object
```

- The first line declares *sam* as a reference to an object of type Employee.
- After this line executes, sam contains the value null, which indicates that it does not yet point to an actual object.
- Any attempt to use sam at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to sam.

- After the second line executes, you can use sam as if it were an Employee object. But in reality, sam simply holds the memory address of the actual Employee object.
- The effect of these two lines of code is depicted in Figure :

Statement **Effect**

Employee sam;



sam= new Employees();

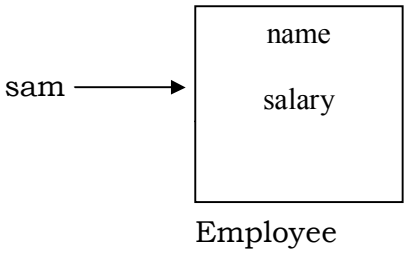


Fig 2.1: Declaring a Object type sam

Example creating an object and accessing class members via an object:

```
class Employee
{
        String name; // person's name          // Instance Variables
        double salary; // salary in dollars
        void details() // Instance Method
        {
            System.out.println("Name: " + name);
            System.out.println("Salary: " + salary);
        }
    }
  class Demo
  {
        public static void main(String[] args)
```

```
        {
        Employee ram = new Employee();
                // may be done on two lines.
                //Employee ram;     //Object declaration
                //ram = new Employee();   // Instantiation
                ram.name = "Ram"; // initialization
                ram.salary = 32000;
                // Now print out ram information using details()
                ram.details();
        }
}
```

The output produced by this program is shown here:

Name:Ram

Salary:32000

new Operator:

- The new operator dynamically allocates memory for an object. The general form is:

  class-var = new *classname*( );

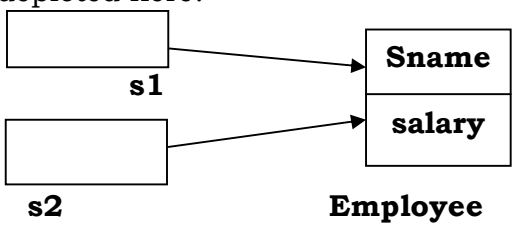
- *class-var* is a variable of the class type being created.
- The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class.
- A constructor defines what occurs when an object of a class is created.
- Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor.
- In example, JVM will initialize the instance variables using a default constructor.
- Java's primitive types are not implemented as objects and so new operator is not required for them.
- new allocates memory for an object during run time, so can create as many or as few objects as needed during the execution.

Assigning Object Reference Variables:

```
Employee s1=new Employee();
Employee s2=s1;
```

- s2 is assigned a reference to a copy of the object referred to by s1. s1 and s2 will both refer to the *same* object.
- With this assignment, s2 refers to the same object as s1. Thus, any changes made to the object through s2 will affect the object to which s1 is referring, since they are the same object.

This situation is depicted here:



- A subsequent assignment to s1 will simply unhook s1 from the original object without affecting the object or affecting s2.

For example:

```
Employee s1=new Employee();
Employee s2=s1;
// ...
s1 = null;
```

Here, s1 has been set to null, but s2 still points to the original object.

➢ METHODS:

- Classes usually consist of two things: instance variables and methods.

General form of a method:

type *name*(*parameter-list*) {

```
        // body of method
}
```

- *type* specifies the type of data returned by the method. This can be any valid type, including class types that we create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name.* This can be any legal identifier other than those already used by other items within the current scope.

- The *parameter-list* is a sequence of type and identifier pairs separated by commas.
- Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called.
- If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than void, return a value to the calling routine using the following form of the return statement:

  return *value*;

Here, *value* is the value returned.

➢ **CONSTRUCTORS:**

- Whenever an object is created for a class, the instance variables of the class needs to be given initial values.
- Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- A *constructor* is a special method which initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.
- When a constructor is not defined for a class, Java compiler provides a default constructor automatically initializes all instance variables to their default values.
- The constructor is automatically invoked as soon as the object is instantiated with the new keyword.
- Constructors have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.
- If any constructor is defined in the class then the JVM will not provide any constructor.
- Example, a simple constructor that simply sets the dimensions of each Box to the same values.

```
/* Here, Box uses a constructor to initialize the dimensions of a box.
class Box {
```

```
        double width;
        double height;
        double depth;
        Box() {                         // This is the constructor for Box.
            System.out.println("Constructing Box");
            width = 10;
            height = 10;
            depth = 10;
        }
      double volume() {          // compute and return volume
            return width * height * depth;
        }
}
class BoxDemo6 {
        public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            double vol;
            vol = mybox1.volume();              // get volume of first box
            System.out.println("Volume is " + vol);
            vol = mybox2.volume();              // get volume of second box
            System.out.println("Volume is " + vol);
        } }
```

When this program is run, it generates the following results:

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

They initialize the details of employee to both mybox1and mybox2**.**

Parameterized Constructors:

- The Box( ) constructor in the preceding example initializes all boxes with the same dimensions.
- Parameters can be passed to a constructor, similar to having parameters for a method. This makes them much more useful.

Example:

```
// Here, Box uses a parameterized constructor to initialize the dimensions
   of a box.
class Box {
        double width;
        double height;
        double depth;
        Box(double w, double h, double d)//This is the constructor for Box
        {
            width = w;
            height = h;
            depth = d;
         }
        double volume() {                          // compute and return volume
            return width * height * depth;
        }
}
class BoxDemo7 {
        public static void main(String args[]) {
            // declare, allocate, and initialize Box objects
            Box mybox1 = new Box(10, 20, 15);
            Box mybox2 = new Box(3, 6, 9);
            double vol;
            vol = mybox1.volume();  //// get volume of first box
            System.out.println("Volume is " + vol);
            vol = mybox2.volume();  // get volume of second box
            System.out.println("Volume is " + vol);
        }
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

- Each object is initialized with values specified in the parameters to its constructor.
  For example, in the following line,
  Box mybox1 = new Box(10, 20, 15);
- The values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object.Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15,respectively.

➢ **'this' KEYWORD:**

- Java defines the '**this**' keyword. **this** can be used inside any instance method to refer to the *current* object.
- **this** is always a reference to the object on which the method was invoked.
- Example:

```
Box(double w, double h, double d) {  // A redundant use of this.
        this.width = w;
        this.height = h;
        this.depth = d;
 }
```

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- But the names of local variables, including formal parameters to methods, may overlap with the names of the class' instance variables.
- So, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.
- This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class.
- **Example:**

Here is another version of Box( ), which uses width, height, and depth for parameter names and then uses 'this' to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
    Box(double width, double height, double depth) {
```

```
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
```

- 'this' can be used for constructor chaining, means a constructor can be called from another constructor.

```
/* First Constructor */
Box()
{
    //constructor chained
   this(14,12,10);
}
/*   Second Constructor  */
Box(double w, double h, double depth) {
        width = w;
        height = h;
        depth = d;
    }
```

➢ **OVERLOADING METHODS AND CONSTRUCTORS:**

- Method overloading is one way of achieving polymorphism in java.
- Each method in a class is uniquely identified by its name and parameter list, means two or more methods with same name, but with a different parameter list. This feature called as method overloading.
- Overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose number and type of parameters match the arguments used in the call.

```
// Demonstrate method overloading.
class OverloadDemo
{
        void test()
        {
                System.out.println("No parameters");
           }
        void test(int a)
        {                       // Overload test for one integer parameter.
                System.out.println("a: " + a);
    }
    void test(int a, int b)
    {                   // Overload test for two integer parameters.
                System.out.println("a and b: " + a + " " + b);
    }
    double test(double a)
    {                           // overload test for a double parameter

                System.out.println("double a: " + a);
                return a*a;
    }

     }

        class Overload

            {

        public static void main(String args[])
        {
                OverloadDemo ob = new OverloadDemo();
                double result;
                // call all versions of test()
                ob.test();
```

```
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }

    }
```

output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

- **test( )** is overloaded four times. The first version takes no parameters,the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter.
- The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.
- overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

Example:

```
        // Automatic type conversions apply to overloading.
class OverloadDemo
{
    void test()
    {
        System.out.println("No parameters");
    }
    void test(int a, int b)
    {   // Overload test for two integer parameters.
        System.out.println("a and b: " + a + " " + b);
```

```
        }
        void test(double a)
        {   // overload test for a double parameter
            System.out.println("Inside test(double) a: " + a);
        }
}
class Overload
{
        public static void main(String args[])
        {
            OverloadDemo ob = new OverloadDemo();
            int i = 88;
            ob.test();
            ob.test(10, 20);
            ob.test(i); // this will invoke test(double)
            ob.test(123.2); // this will invoke test(double)
        }
}
```

output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

- *OverloadDemo* does not define **test(int)**. So java will automatically convert int to **double**.

OVERLOADING CONSTRUCTORS:

- Similar to methods constructor can also be overloaded.
- Constructors for a class having the same name as that of class, but with different signatures I.e., different number of arguments or different types of arguments.

Example:

```
class Box
```

```
{
    double width;
    double height;
    double depth;
    Box(double w, double h, double d)
    {// This is the constructor for Box.
        width = w;
        height = h;
        depth = d;
    }
    double volume()
    {// compute and return volume
        return width * height * depth;
    } }
```

- The Box( ) constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box( ) constructor.
- For example,the following statement is currently invalid.

```
Box ob = new Box();
```

because Box( ) requires three arguments.

```
// Here, Box defines three constructors to initialize the dimensions of a box
various ways.
class Box
{
        double width;
        double height;
        double depth;
    Box(double w, double h, double d)
    {   //constructor used when all dimensions specified
            width = w;
            height = h;
            depth = d;
    }
    Box()
```

```
        {       // constructor used when no dimensions specified
                width = -1;         // use -1 to indicate
                height = -1;        // an uninitialized
                depth = -1;         // box
        }
        Box(double len)
        {       // constructor used when cube is created
                width = height = depth = len;
        }
        double volume()
        {// compute and return volume
                return width * height * depth;
        }
}
class OverloadCons
{
    public static void main(String args[])
    {// create boxes using the various constructors
            Box mybox1 = new Box(10, 20, 15);
            Box mybox2 = new Box();
            Box mycube = new Box(7);
            double vol;
            vol = mybox1.volume();      // get volume of first box
          System.out.println("Volume of mybox1 is " + vol);
            vol = mybox2.volume();      // get volume of second box
            System.out.println("Volume of mybox2 is " + vol);
            vol = mycube.volume();      // get volume of cube
            System.out.println("Volume of mycube is " + vol);
            }
      }
```

output:

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0.

Volume of mycube is 343.0

- The appropriate overloaded constructor is called based upon the parameters specified when **new** is executed.

➢ ACCESS CONTROL:

- Encapsulation links data with the code that manipulates it. Encapsulation provides another important attribute: ***access control***.
- Through encapsulation, we can control what parts of a program, can access the members of a class.
- How a member can be accessed is determined by the ***access specifier*** that modifies its declaration. *Java supplies a rich set of access specifiers*. Some aspects of access control are related mostly to inheritance or packages.
- Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level.
- **protected** applies only when inheritance is involved.
- A member of a class is modified by the **public** specifier, then that member can be accessed by any other code.
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- No access specifier is used, then by **default** the member of a class is **public** within its own package, but cannot be accessed outside of its package.

```
// This program demonstrates the difference betweenpublic and private.
class Test
{
        int a;                  // default access
        public int b;           // public access
        private int c;    // private access
        // methods to access c
        void setc(int i)
        {   // set c's value
                c = i;
        }
        int getc()
```

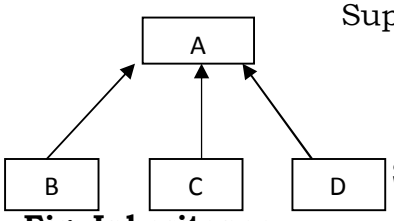
```
        {   // get c's value
                return c;
}}
class AccessTest
{
        public static void main(String args[])
        {
                Test ob = new Test();
                // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;
                // This is not OK and will cause an error
                // ob.c = 100; // Error!
                // You must access c through its methods
                ob.setc(100); // OK
                System.out.println("a, b, and c: " + ob.a + " " +
                ob.b + " " + ob.getc());
}}
```

- Inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**.
- Member **c** is given private access. This means that it cannot be accessed by code outside of its class.
- So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**.

INHERITANCE

➢ BASICS:

- Inheritance is the process by which one class acquires the properties (instance variables, methods) of another class.
- A deeply inherited subclass (descendent) inherits all of the properties from each of its ancestors in the class hierarchy.
- Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent.



Super class / Base class/ Parent class

Sub class / Derived class / Child class

Fig: Inheritance

Main purpose of Inheritance:

1. Reusability.
2. Abstraction.

- **"**extends" keyword is used to inherit the properties from one class to another class**.**

Member Access

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private** and accessing restrictions are as follows,
  - public: variable/method can be accessed anywhere
  - private: variable/method can be accessed only within this class (but NOT within subclasses)
  - protected: variable/method can be accessed:
    – Within this class.
    – Within any class/subclass in the same *package.*
    – Within any subclass of this class in other package.

Note: if you do not include an access specifier (default), the Variable or method has *package access.*

➢ **TYPES OF INHERITANCE:**

There are 5 types of Inheritance

- Single Inheritance.
- Multilevel Inheritance.
- Hierarchical Inheritance.
- Multiple Inheritance
  - JAVA does not support, need to use Interface. 'extends' can be used with only one class.
- Hybrid Inheritance

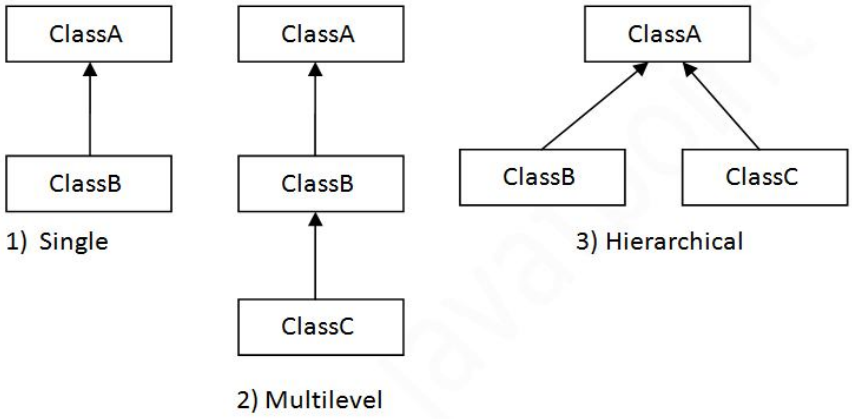


Fig: Types of Inheritance

- **Program for Single Inheritance:**

```
/* simple inheritance */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public void Adisplay()
    {
      System.out.println("\n \t in A class i= "+i);
    }
 }
class B extends A
 {
   public int j;
   public B()
    {
      System.out.println("\n \t default constructor B() is called");
```

```
        j=20;
    }
    public void Bdisplay()
    {
        j=i+1;
        System.out.println("\n \t in B class j= "+j);
    }
}
class Simple
 {
    public static void main(String ar[])throws IOException
     {
        System.out.println("\n \t start of main()");

        B b=new B();
        b.Adisplay();
        b.Bdisplay();
        System.out.println("\n \t end of main()");
     }
 }
```

OUTPUT:

```
start of main()
default constructor A() is called
default constructor B() is called
```

in A class i= 10

in B class j= 11

end of main()

- **Program for Multilevel Inheritance:**

```
/* multilevel inheritance */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public void Adisplay()
    {
      System.out.println("\n \t in A class i= "+i);
    }
 }
class B extends A
 {
   public int j;
   public B()
    {
      System.out.println("\n \t default constructor B() is called");
      j=20;
    }
   public void Bdisplay()
    {
      j=i+1;
```

```
        System.out.println("\n \t in B class j= "+j);
    }
}
class C extends B
{
   public int k;
   public C()
    {
        System.out.println("\n \t default constructor C() is called");
        k=30;
    }
   public void Cdisplay()
    {
        k=i+j;
        System.out.println("\n \t in C class k= "+k);
    }
}
 class MulLevel
 {
    public static void main(String ar[])throws IOException
     {
        System.out.println("\n \t start of main()");

        C c=new C();
        c.Adisplay();
        c.Bdisplay();
        c.Cdisplay();
        System.out.println("\n \t end of main()");
     }  }
```

OUTPUT:

```
   start of main()
```

```
default constructor A() is called
default constructor B() is called
default constructor C() is called
in A class  i= 10
in B class  j= 11
 in C class k= 21
end of main()
```

A Super class Variable Can Reference a Subclass Object

- A reference variable of a super class can be assigned a reference to any subclass derived from that super class.
- You will find this aspect of inheritance quite useful in a variety of situations . For example, consider the following:

```
// This program uses inheritance to extend Box.
class Box
 {
   double width;
   double height;
   double depth;

// construct clone of an object
   Box(Box ob)
   {
            // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
   }
// constructor used when all dimensions specified
  Box(double w, double h, double d)
  {
    width = w;
```

```
    height = h;
    depth = d;
  }
// constructor used when no dimensions specified
 Box()
 {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
 }
 // constructor used when cube is created
Box(double len)
 {
     width = height = depth = len;
 }
   // compute and return volume
double volume()
  {
      return width * height * depth;
  }
}
// Here, Box is extended to include weight.
class BoxWeight extends Box
 {
    double weight; // weight of box
    // constructor for BoxWeight
    BoxWeight(double w, double h, double d, double m)
    {
       width = w;
       height = h;
       depth = d;
```

```
        weight = m;
    }
}
class DemoBoxWeight
{
   public static void main(String args[])
    {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;
        vol = mybox1.volume();
         System.out.println("Volume of mybox1 is " + vol);
         System.out.println("Weight of mybox1 is " + mybox1.weight);
         System.out.println();
         vol = mybox2.volume();
         System.out.println("Volume of mybox2 is " + vol);
         System.out.println("Weight of mybox2 is " + mybox2.weight);
      }
 }
```

Output:

Volume of mybox1 is 3000.0

Weight of mybox1 is 34.3

Volume of mybox2 is 24.0

Weight of mybox2 is 0.076

- **BoxWeight** inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes.
- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.

- For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box
{
    int color; // color of box
    ColorBox(double w, double h, double d, int c)
    {
       width = w;
       height = h;
       depth = d;
       color = c;
   }
    }
```

- Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own, unique attributes. This is the essence of inheritance.

```
    class RefDemo
{
  public static void main(String args[])
   {
     BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
      Box plainbox = new Box();
      double vol;
      vol = weightbox.volume();
      System.out.println("Volume of weightbox is " + vol);
      System.out.println("Weight of weightbox is " +
      weightbox.weight);
      System.out.println();
```

```
        plainbox = weightbox;  // assign BoxWeight reference to Box
    reference
        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);
    /* The following statement is invalid because plainbox does not define a
       weight member. */
       // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}
```

- What members can be accessed is determined based on the type of the reference variable, not on the type of the object that it refers to.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

➢ **SUPER KEYWORD:**

- 'super' is used when a subclass wants to refer to its **immediate** super class members.
- 'super' has two general forms.
  - ✓ To make a call to the super class constructor from sub class constructor.
  - ✓ The second is used to access a member of the superclass that has been hidden by a member of a subclass.

program for super keyword:

```
/*  super keyword */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
```

```
        i=10;
    }
    public void display()
    {
        System.out.println("\n \t in A class i= "+i);
    }
}
class B extends A
{
    public int i;
    public B()
    {                       // invoking the super class constructor .
        super();            // always must be first statement and it is default
        System.out.println("\n \t default constructor B() is called");
        this.i=20;
        super.i=30;         //  points the super class instance variable.
    }
    public void display()
    {
        super.display();    // calling super class method.
        this.i=this.i+super.i;
        System.out.println("\n \t in B class subofi+supofi =  "+this.i);
    }
}
class Super
{
    public static void main(String ar[])throws IOException
    {
        System.out.println("\n \t start of main()");
        B b=new B();
        b.display();
        System.out.println("\n \t end of main()");
    }
}
```

OUTPUT:

```
start of main()
default constructor A() is called
default constructor B() is called
in A class i= 30
in B class subofi+supofi = 50
end of main
```

- **METHOD OVERRIDING:**
  - In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
  - When an overridden method is called within a subclass, it will always refer to the version of that method defined by the subclass. The version of ***the method defined by the superclass will be hidden***.

Program for Method Overriding:

```
/* overriding(Run-time polymorphism) */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public void display()
    {
      System.out.println("\n \t in A class i= "+i);
    }
 }
class B extends A
 {
```

```
    public int j;
    public B()
     {
        System.out.println("\n \t default constructor B() is called");
        j=20;
     }
    public void display()
     {
        j=i+1;
        System.out.println("\n \t in B class j= "+j);
     }
}
 class OverRiding
 {
    public static void main(String ar[])throws IOException
     {
        System.out.println("\n \t start of main()");
        B b=new B();
        b.display();   // display()  in B class overrides the display() in A class
        System.out.println("\n \t end of main()");
     }
 }
```

OUT PUT

```
start of main()
default constructor A() is called
default constructor B() is called
 in B class j= 21
end of main
```

- **DYNAMIC METHOD DISPATCH:**
  - Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.
  - Using this feature, java implements Runtime Polymorphism in Java.

- "A super class reference variable can refer to a sub class object", so resolves call to a overridden method during runtime.
- When an overridden method is called through a super class reference, java determines which version of overridden method to execute based upon the type of being refereed to at the time the call occurs.
- It is the type of object being referred to at the time of call occurs, not the type of reference variable that determines which version of an overridden method will be executed.

Program for Dynamic method dispatch:

```
// Dynamic Method Dispatch
class A
        {
            void callme()
              {
                    System.out.println("Inside A's callme method");
              }
         }
class B extends A
    {
          // override callme()
         void callme()
          {
                System.out.println("Inside B's callme method");
          }
    }
class C extends A
   {
        // override callme()
       void callme()
        {
```

```
            System.out.println("Inside C's callme method");
        }
    }
class Dispatch
    {
        public static void main(String args[]) {
            A a = new A();              // object of type A
            B b = new B();              // object of type B
            C c = new C();              // object of type C
              A r;                      // obtain a reference of type A
              r  = a;                   // r refers to an A object
              r.callme();               // calls A's version of callme
              r = b;                    // r refers to a B object
              r.callme();               // calls B's version of callme
              r = c;                    // r refers to a C object
              r.callme();               // calls C's version of callme
    }
}
```

Output:

Inside A's callme method
Inside B's callme method
Inside C's callme method

- This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**.
- Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared.
- The program then assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**.
- As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call, determined by

the type of the reference variable, **r**, from three calls to **A**'s **callme( )** method.

- **ABSTRACT CLASS:**
  - An abstract class contain one or more abstract methods.
  - An *abstract method* is method without a body, i.e., only declared but not defined.
  - The keyword "abstract" is used to indicate a method/class as abstract ones.
  - Abstract classes cannot be instantiated.
  - Abstract methods needs to be defined in subclasses of the abstract class.

Program for abstract class:

```
   /* abstract keyword */
import java.io.*;
abstract class A
{
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public abstract void add(); // no definition since abstract
   public void Adisplay()
    {
      System.out.println("\n \t in A class i= "+i);
    }
}
class B extends A
{
   public int j;
```

```
    public B()
     {
        System.out.println("\n \t default constructor B() is called");
        j=20;
     }
    public void Bdisplay()
     {
        System.out.println("\n \t in B class j= "+j);
     }
    public void add()
     {
         System.out.println("\n \t in B class add() is called");
         j=j+i;
     }
}
 class Abstract
 {
    public static void main(String ar[])throws IOException
     {
        System.out.println("\n \t start of main()");
        B b=new B();
        b.add();
        b.Bdisplay();
        System.out.println("\n \t end of main()");
     }
 }
```

OUTPUT:

start of main()

default constructor A() is called

default constructor B() is called

in B class add() is called

in B class j=30

end of main()

- Note: abstract classes must be inherited.

- Note: abstract classes must be override

➢ **USING FINAL WITH INHERITANCE:**

- The final keyword is used in three ways
  - To variables, those become constants.
  - To methods, those should not be override.
  - To the class, then that class should not be inherited.

Program which illustrates the usage of final keyword

```
  /* final keyword to variable, method */
import java.io.*;
class A
{
   public int i;
   final int SPEED_LIMIT=60;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public final void Aadd() // not overrided since final
    {
      System.out.println("\n \t in A class final add() method is called ");
      i=i+10;
    }
   public void Adisplay()
    {
      System.out.println("\n \t in A class i= "+i);
    }
}
class B extends A
{
```

```
    public int j;
    public B()
     {
        System.out.println("\n \t default constructor B() is called");
        j=20;
     }
    public void Badd()
     {
         System.out.println("\n \t in B class Badd() is called");
         j=j+i;
     }
    public void Bdisplay()
     {
        System.out.println("\n \t in B class j= "+j);
     }
}
 class FinalMethod
 {
    public static void main(String ar[])throws IOException
     {
        System.out.println("\n \t start of main()");
        B b=new B();
        b.Aadd();
        b.Adisplay();
        b.Badd();
        b.Bdisplay();
        System.out.println("\n \t end of main()");
     }
 }
```

OUTPUT

```
    start of main()
```

default constructor A() is called

default constructor B() is called

in A class final add() method is called

in A class i=20

in B class Badd() is called

in B class j=40

end of main()

1. /* final keyword to class */

```
import java.io.*;
final class A       // not inherited
 {
   public int i;
   public A()
    {
      System.out.println("\n \t default constructor A() is called");
      i=10;
    }
   public final void Aadd() // not overrided since final
    {
      System.out.println("\n \t in A class final add() method is called ");
      i=i+10;
    }
   public void Adisplay()
    {
      System.out.println("\n \t in A class i= "+i);
    }
 }
 class FinalCV
  {
```

```
    public static void main(String ar[])throws IOException
    {
       System.out.println("\n \t start of main()");
       final int f=100;  // like const variable
       System.out.println("\n \t in main() the value of final f= "+f);
       A ob=new A();
       ob.Aadd();
       ob.Adisplay();
       System.out.println("\n \t end of main()");
    }
}
```

OUT PUT:

start of main()

in main() the value of final f=100

default constructor A() is called

in A class final add() method is called

in A class i=20

end of main()

➢ **THE OBJECT CLASS:**

- There is one special class, **Object**, defined by Java.
- All other classes are subclasses of **Object**, **Object** is a superclass of all other classes.
- This means that a reference variable of type **Object** can refer to an object of any other class

METHOD	PURPOSE
protected Object cone()	Creates a new object that is same as the object being cloned
boolean equals(Object ob)	Determines whether one object is equal to another
protected void finalize()	Called before an unused object is recycled

final class getClass()	Obtains the class of an object at runtime
int hashCode()	Returns the hashcode associated with the invoking object
void notify()	Resumes execution of a thread waiting on the invoking object
void notifyAll()	Resumes execution of all threads waiting on the invoking object
String toString()	Returns a string that describes the object
void wait()	Waits on another thread of execution
void wait(long milliseconds)	
void wait(long milliseconds, int nanoseconds)	

UNIT-II
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1) Which of the following is the correct syntax for creating Object [ ]

A)Classname objName=new Classname

B)Classname objName=new Classname();

C)Classname objName=Classname();

D)objName classname=new objName();

2) _____is a keyword that refers to the current object that invoked the method.

3) _____ is the process of reclaiming the runtime unused memory automatically.

4) _____is the process of defining 2 or more methods within same class that have same name but different parameter declarations. [ ]

A) Method overriding B) Method overloading

C) Method hiding D) None of the above

5) 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 {}

6) Run-time polymorphism is achieved by using_____ [ ]

A) Method Overloading B) Constructor Overloading

C) Method Overriding D) this keyword

7) _____ is the Super class for all the classes in Java

8) What is the output of this program? [ ]

```
class box
        {
```

```
        int width;
        int height;
        int length;
        int vol;
        box()
        {
    width = 5;
        height = 5;
    length = 6;
    }
void volume()
{
    vol = width*height*length;
}
}
class constructor_output
{
public static void main(String args[])
{
    box obj = new box();
    obj.volume();
    System.out.println(obj.vol);
```

```
}
}
```

A) 100 B) 150 C) 200 D) 250

9) Consider the following code [ ]

```
class A
{
private int i;
public int j;
}
class B extends A
{
int k;
void show()
{
k=i+j;
System.out.println("sum of " +i+ "and" +j+"="+k);
}
public static void main(String arg[])
    {
B b1=new B();
    }
```

```
}
```

A) B gets only the member j through inheritance from A

B) B gets both i, j through inheritance from A

C) A is the sub class and B is the super class

D) None of the above

10) what is the output of this program? [ ]

```
class overload
    {
    int x;
    int y;
    void add(int a)
        {
        x =  a + 1;
        }
        void add(int a, int b)
            {
           x =  a + 2;
            }
    }
class Overload_methods
  {
    public static void main(String args[])
```

```
{
        overload obj = new overload();
        int a = 0;
            obj.add(6,7);
            System.out.println(obj.x);
}
}
```

a) 5 b)8 c)7 d) 6

11 The following code prints_____ [ ]

```
class A
    {
    int i;
    int j;
    A()
    {
        i = 1;
        j = 2;
    }
  }
class Output
{
    public static void main(String args[])
    {
        A obj1 = new A();
```

```
            System.out.print(obj1.toString());
        }
    }
```

a. true
b. false
c. String associated with object
d. Compilation Error

12 Predict the output of following Java Program. [ ]

```
class Grandparent
{
    public void Print( )
    {
        System.out.println("Grandparent's Print()");
    }
}
class Parent extends Grandparent
{
    public void Print( )
    {
        System.out.println("Parent's Print()");
        System.exit(0);
    }
}
class Child extends Parent
```

```
{
        public void Print()
    {
                super.Print();
                System.out.println("Child's Print()");
        }
}
public class Main
{
        public static void main(String[] args)
        {
                Child c = new Child();
                c.Print();
        }
}
```

A) Grandparent's Print()
B) Parent's Print()
C) Child's Print()
D) Runtime Error

13 What is the output of the following Java program? [ ]

```
class Test
{
        int i;
```

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

(A)0 (B) garbagevalue
(C) compilererror (D) runtime error

14 What is the output of the following Java program? [ ]

```
class Point
{
        int m_x, m_y;
        public Point(int x, int y)
                {
                m_x = x;    m_y = y;
            }
    public static void main(String args[])
        {
            Point p = new Point();
```

}}

(A)1 (B) garbagevalue (C) compilererror (D) runtime error

SECTION-B

SUBJECTIVE QUESTIONS

1) Define class. Write the steps for creating class and object? Explain it with an example?
2) Define constructor? Can we overload a constructor? If so, explain with an example?
3) Explain the usage of following keywords with examples?
   a) this b) super c) final
4) List Different types of Inheritance? Explain with example programs?
5) To read an integer n and then print the n^{th} table as below:

   1 x n = n

   2 x n = 2n

   . . . . .

   10 x n = 10n

6) To read the details of a student like name, age, phone number in a method called getData() and then write another method called putData() to display the details.
7) To find factorial of a given number using recursion?
8) (a) Implement Method overloading with the following example?
   (a) To overload a method area() which computes the area of a geometrical figure based on number of parameters. If number of parameters is 1 and is of type float it should calculate the area of circle, if it is of type int it should calculate area of square. If the number of parameters is 2 and they are of type float calculate area of triangle, if they are of int calculate area of rectangle.
9) Implement dynamic method dispatch with an example.

10) Define Abstract class. Differentiate abstract method and concrete method?

UNIT – III

Objective:

- To get acquainted with the concepts of Interface and Packages.

Syllabus:

Interfaces: Defining an interface, Implementing interfaces, Nested interfaces, Variables in interfaces and extending interfaces.

Packages: Defining, Creating and Accessing a Package.

Learning Outcomes:

At the end of the unit student will be able to Understand:

- Define and Develop an interface
- Implement the Nested and extending Interfaces
- Describe the creation of Packages and its accessing
- Write a java program using interfaces
- Write a java program using packages
- Create a sample package and use it in another application

Learning Material

Interfaces:

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is a collection of abstract methods and constants that one or more classes of objects will use.

Defining an Interface: Interface definition is same as class except that it consists of the methods that are declared have no method body. Syntax for an interface is as follows:

Syntax:

```
<access specifier> interface <interface_name>
  {
        Type varname1=value;
        Type varname2=value;
              .
                .
        returntype methodname1(parameterlist);
        returntype methodname2(parameterlist);
                    .
                      .
                      .
                      .
}
```

Where,

- Access specifier is always public only, public access specifier indicates that the interface can be used by any class. Otherwise, the interface will accessible to class that are defined in the same package as in the interface.
- Interface keyword is used to declare the class as an interface
- Interface_name is the name of the interface and it is a valid identifier.

Why do we use interface ?

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

- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

Example:

```
public interface shape
{
        int radius=2;
    public void area(int a);
}
```

Implementing Interfaces:

Once an interface has been defined, one or more classes can implement that interface. "implements" keyword used for implementing the classes. The syntax of implements is as follows:

Syntax:

```
class class_name implements interface1, interface 2, …….. interface n
{
        ----------
        ----------      // interface body
        -------
}
```

If a class implements more than one interface, the interfaces are separated with a comma operator. The methods that implement an interface must be declared public. Also type signature of implementing method must match exactly the type signature specified in the interface definition.

Example:

```
interface  it1
{
        int x=10, y=20;
        public void add(int a, int b);
        public void sub(int a, int b);
}

Class demo implements it1
```

```
{
        public  void add(int s, int w)
        {
                System.out.println("Addition="+(s+w));
        }
        public void sub(int s, int w)
        {
                System.out.println ("Subtraction="+(s-w));
        }
public static void main(String args[])  {
demo2 obj=new demo( );
obj.add(3,4);
obj.sub(5,2);
System.out.println(obj.x + obj.y);
Obj.x=70;       // error since x is final variable in interface
}
```

Note:

1. interface methods are similar to the abstract classes so, that it cannot be instantiated.
2. interface methods can also be accessed by the interface reference variable which refer to the object of subclasses. The method will be resolved at run time. This process is similar to the "super class reference to access a subclass object".

Example:

```
interface it1
{
        int x=10, y=20;
        public void add(int a, int b);
        public void sub(int a, int b);
}
Class it2 implements it1
{
        public  void add(int s, int w)
        {
                System.out.println("Addition="+(s+w));
        }
        public void sub(int s, int w)
        {
```

```
        System.out.println ("Subtraction="+(s-w));
    }
    public static void main(String args[])
    {
                it2 obj=new it2( );
                it1 ref;
                ref=obj;
               System.out.println(ref.x + ref.y);
    }
```

3. If a class includes an interface but does not fully implement the methods defined by that interface, then the class becomes abstract class and must be declared as abstract in the first line of its class definition.

Example:

```
interface it1
{
    int x=10, y=20;
    public void add(int a, int b);
    public void sub(int a, int b);
}
abstract class it2 implements it1
{
    public  void add(int s, int w)
    {
        System.out.println("Addition="+(s+w));

    }
}
Class it3 extends it2
{
    public void sub(int s, int w)
    {
        System.out.println ("Subtraction="+(s-w));
    }
    public static void main(String args[])
     {
        it3 obj=new it2( );
        obj.add(5,6);
```

```
}
```

Nested interfaces:

An interface that 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 can't be accessed directly.

Points to remember for nested interfaces

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Syntax of nested interface:

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

Example:

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
interface Showable
{
    void show();
    interface Message
            {
                    void msg();
            }
}
class TestNestedInterface1 implements Showable.Message
{
```

```
public void msg()
{
                        System.out.println("Hello nested interface");
                }

                public static void main(String args[])
                {
                //upcasting here
                Showable.Message message=new TestNestedInterface1();
                message.msg();
                }
        }
```

Variables in interfaces:

Variables in an interface are implicitly public, final and static and there is no need to explicitly declare them as public, static and final. As they are final, they need to be assigned a value compulsorily. Being static, they can be accessed directly with the help of an interface name and as they are public, we can access them from anywhere. The following example program shows the usage of variables in an interface

Example:

```
interface test
{
        int lowerlimit=0;
        int upperlimit=100;
}
class Variable_Test implements test
{
  void limits(int a);
   {
      if(a>lowerlimit && a< upperlimit)
        System.out.println(a+ “lie in between” +Variable_test.lowerlimit +
                        “and”  +Variable_Test.upperlimit );
     else
              System.out.println(a+ “does not lie in between”
                +Variable_test.lowerlimit  +“and”  +Variable_Test.upperlimit);
        public static void main(String args[])
        {
              Variable_Test vt= new Variable_Test();
              vt.limits(23);
```

```
                vt.limits(233);
}       }
```

OUTPUT:
23 lie in between 0 and 100
233 does not lie in between 0 and 100

Extending interfaces:

Just like normal classes, interfaces can also be extended. An interface can inherit another interface using the same keyword **extends**, and not the keyword implements. The following example program shows how interfaces are extended.

Example:

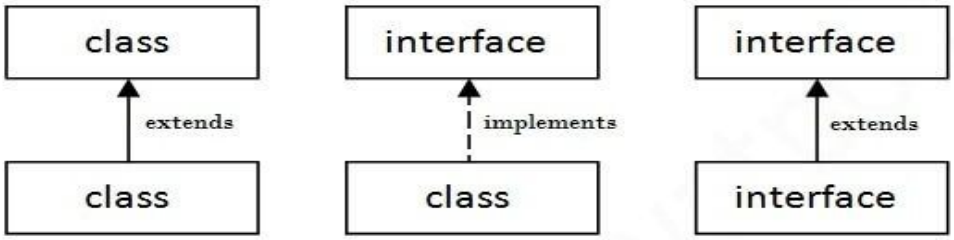
```
interface intfA
{
        Void showA();
}
interface B extends intfA
{
        Void showB();
}
class Demo implements B
{
        public void showA()
        {
                System.out.println("Overriden method of interface A");
        }


        public void showb()
        {
                System.out.println("Overriden method of interface B");
        }
        Public static void main(String args[])
        {
                Demo d = new Demo();
                d.showA();
                d.showB();
        }
}
```

OUTPUT:

Overriden method of interface A
Overriden method of interface B

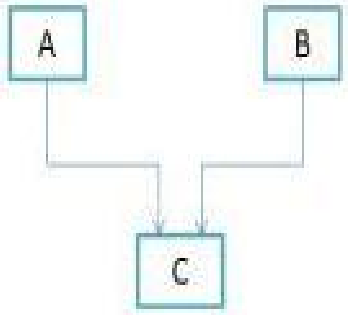
The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Multiple Inheritance using Interfaces:

Multiple inheritance enables to derive a class from multiple parent classes. Multiple Inheritance is not supported by java directly, need to use interfaces. Let A and B are parent classes and C is the derived class



Java provides interface approach to support the concept of multiple inheritance.

An interface can extend multiple interfaces and a class can implements multiple interfaces.

The following example program will explain the concept of Multiple inheritance.

Program: A program in java to show multiple inheritance

```
class student
{
        int rollNumber;
        void getNumber(int n)
        {
                rollNumber=n;
        }
        void printNumber()
        {
                System.out.println("RollNo is " +rollNumber);
        }
}


class test extends student
{
        float part1,part2;
        void getMarks(float a, float b)
        {
                part1=a;
                part2=b;
        }
        void putMarks()
        {
                System.out.println("Marks Part1 "+part1);
                System.out.println("Marks Part2 "+part2);
        }
}
interface sports
{
        float sportwt=6.0F;
        void putwt();
}
class results extends test implements sports
{
        float total;
        public void putwt()
        {
                System.out.println("Sports Marks "+ sportwt);
        }
        void display()
        {
```

```
                total=part1+part2+sportwt;
                System.out.println("Total marks of " +rollNumber+" is "+total);
        }
}
class mainClass
{
        public static void main(String srgs[])
        {
                results a=new results();
                a.getNumber(10);
                a.printNumber();
                a.getMarks(10.0F,25.5F);
                a.putMarks();
                a.putwt();
                a.display();
        }}
```

Packages

Package: Package is collection of related classes. Each class defines number of methods. Java packages are classified into 2 types

1. Java API(Application Program Interface) packages (or) Predefined packages (or) Built in packages. These packages are defined by the system. Some of the example for system defined packages are java.lang, java.util, java.io etc.,
2. User defined packages: These packages are defined by the users.

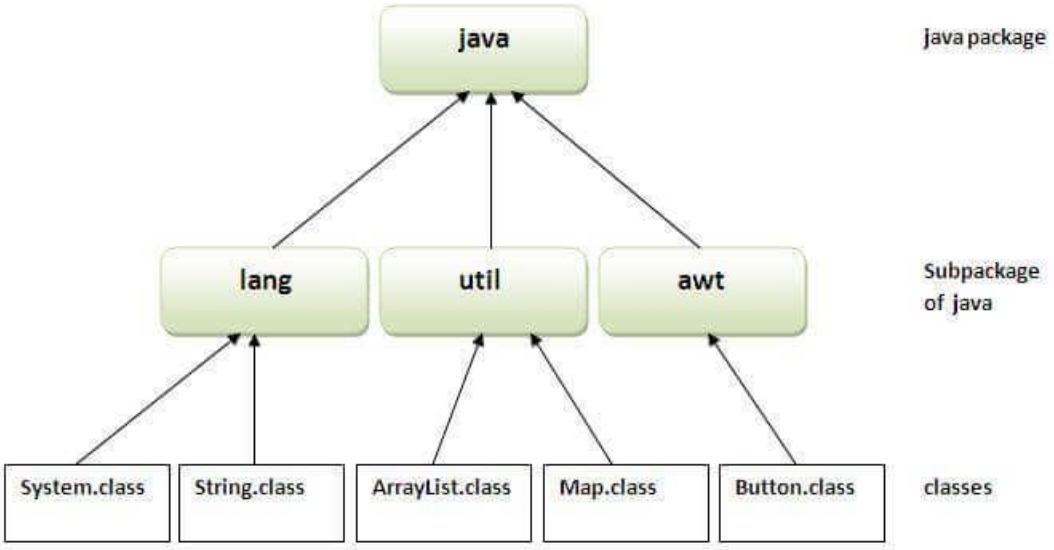
Advantage of Java Package

1.Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2.Java package provides access protection.

3. Java package removes naming collision.

Example Java Package:



Defining, creating and accessing a package:

To define a package, place "**package**" keyword as the **first statement** in the java source file. So, that any class declared within that file will belong to the specified package. The syntax of package creation is as follows:

Syntax: package package_name;

Where pack_name is the name of the package.

Example:

```
package mypack;
public class number
{
        public void add(int a, int b)
         {
                System.out.println("Sum="+(a+b));
          }
  }
```

- The class that is defined in the package must be start with the **public** access modifier. So, that it can be accessible by any another of them. If it is **not public, it is accessible only in that package**.

- Java uses file system directories to store packages. We save the program with number.java and compile the package is as javac –d number.java. Due to this compilation mypack directory is automatically created and .class file is stored in that directory.
- Package creation has completed. The package information is now including in our actual program by means of "import" statement**. "import"** is a keyword that links the package with our program. It is placed before the class definitions.

```
import mypack.*;

        Or

import mypack.number;
```

Example program for packages:

```
Package mypack;                 // Package Creation
Public class number
{
        Public void add(int a, int b)
        {
                System.out.println("Sum="+(a+b));
        }
    }
import mypack.*;                                // accessing package created
class pack
{
        public static void main(String args[])
        {
                Number obj=new number();
                Obj.add(3,4);
        }
}
```

Sub packages:

It is also possible to create sub packages for the main package like creating subfolders.

Syntax: **package pack1.pack2;**

Where pack1 is the main package and pack2 is the sub package.

Example:

```
package pack1;
public class x
{
        public void show()
        {
                System.out.println("Super");
        }
}


package pack1.pack2;        // creating pack2 under the package pack1
public class y
{
        public void display()
        {
                System.out.println("sub");
        }
}
import pack1.x;
import pack1.pack2.y;
class check {
        public static void main(String args[])
        {
                x obj=new x();
                obj.show();
                y obj1=new y();
                obj1.display();
}       }
```

Access Protection:

Java provides four types access modifiers as public, private, default and protected. Any variable declared as public, it could be accessed from anywhere. Any variable

declared as private cannot be seen outside of its class. Any variable declared as default, it is visible to subclasses as well as to other classes in the same package. Any variable declared as protected, it allows a member to be seen outside of current package, but only to classes that subclasses directly.

Package Access Location	Public	Private	Default	Protected
Same class	Yes	Yes	Yes	Yes
Sub-class in Same package	Yes	No	Yes	Yes
Non sub-class in Same package	Yes	No	Yes	Yes
Sub-class in Different package	Yes	No	No	Yes
non sub-class in Different package	Yes	No	No	No

Class Path:

A class path is an environmental variable, which tells the java virtual machine and other java tools(javac, java) where to find the class libraries, including user defined class libraries. By default java uses the class path as

C:\jdk1.2.1\libv\classes.zip

A user defined class path is set for the environment as

C:\>SET CLASSPATH=%CLASSPATH%;path to the created package

(or)

C:\>SET PATH = "jdk1.2.1\bin";

UNIT-III
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1) _____ keyword is used for implement the interface in JAVA

2) Which of the access specifier can be used for an Interface _____

3) Which of these keywords is used to define interfaces in JAVA [ ]

(a) implement (b) interface (c) Both a & b (d) None of these

4) The methods of interface are _____ by default. [ ]

(a)Abstract (b) static (c) final (d) none of these

5) The variables of interfaces are final and static by default (True / False)

6) A class can implements _____ interfaces [ ]

(a) only one (b) one or more than one (c) maximum two (d) minimum two

7) An interface contains_____ [ ]

(a) The method definitions (b) The method declaration

(c) Both a & b (d) None

8) Which of the following is correct way of implementing an interface [ ]
salary by class manager?

(a) class manager extends salary {} (b) class manager implements salary {}

(c) class manager imports salary {} (d) None of the mentioned

9) Is it possible to create object of an interface ? (True / False)

10) Which of these keyword is used to define packages in JAVA ? [ ]

(a) pkg (b) Pkg (c) package (d) Package

11) Which of the following is correct way of importing an entire package 'pkg' ?[ ]

(a) import pkg. (b) import Pkg. (c) import pkg.* (d) import Pkg.*

12) Package consists of ? [ ]

(1) classes (2) methods (3) variables (4) All of the above

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

13) Is it possible to access the private class outside the package ? (True / False)

14) Package is the first statement in java program ? (True / False)

15) What is the output of this program? [ ]

```
interface calculate {
   void cal(int item);
}
class display implements calculate {
   int x;
   public void cal(int item) {
      x = item * item;
   }
}
class interfaces {
   public static void main(String args[]) {
      display arr = new display;
      arr.x = 0;
      arr.cal(2);
      System.out.print(arr.x);
   }
}
```

a) 0 b) 2 c) 4 d) None of the mentioned

16) Determine output of the following code:

```
interface A { }
class C { }
class D extends C { }
class B extends D implements A { }
public class Test extends Thread{
      public static void main(String[] args){
```

```
        B b = new B();
        if (b instanceof A)
            System.out.println("b is an instance of A");
        if (b instanceof C)
            System.out.println("b is an instance of C");
    }
}
```

A) Nothing. B) b is an instance of A.

C) b is an instance of C D) b is an instance of A followed by b is an instance of C

17) Which of the above line will give compilation error? [ ]

```
interface Test
 {
    int p = 10; //line 1
    public int q = 20; //line 2
    public static int r = 30; //line 3
    public static final int s = 40; //line 4
}
```

a) 1 b) 3 c) 4 d) None of these

18) What is the output for the below code? [ ]

```
interface A{
public void printValue();
}
public class Test{
public static void main (String[] args){
A a1 = new A(){
public void printValue(){
System.out.println("A");
}
};

a1.printValue();

} }
```

A) Compilation fails due to an error on line 3 B) A

C) Compilation fails due to an error on line 8 D) null E) None of these

19) What will be the output for the below code ? [ ]

```
public interface TestInf  {
        int i =10;
```

```
}
public class Test{
public static void main(String... args) {
  TestInf.i=12;
    System.out.println(TestInf.i);
  }
}
```

A) Compile with error B) 10 C) 12 D) Runtime Exception

20) **What is the output of this program?** [ ]

```
package pkg;
class output {
public static void main(String args[])
{
StringBuffer s1 = new StringBuffer("Hello");
s1.setCharAt(1, x);
System.out.println(s1);
}
}
```

a) xello b) xxxxx c) Hxllo d) Hexlo

21) **What is the output of this program?** [ ]

```
package pkg;
class output {
        public static void main(String args[])
        {
StringBuffer s1 = new StringBuffer("Hello World");
s1.insert(6 , "Good ");
System.out.println(s1);}}
```

Note : Output.class file is not in directory pkg.

a) HelloGoodWorld b) HellGoodoWorld
c) Compilation error d) Runtime error

22) Which of the given statement is not true about an Java Package ? [ ]

A) A package can be defined as a group of similar types of classes and interface.

B) Package are used in order to avoid name conflicts and to control access of classes and interface.

C) A package cannot not have another package inside it.

D) Java uses file system directory to store package

23) You can import only static members of a class present in some other package using ?

A) import keyword B) import static keyword

C) package keyword D) static import keyword

24) Which is a valid declaration within an Interface ?

A) public static short stop = 23; B) protected short stop = 23;

C) transient short stop = 23; D) final void start(short stop);

SECTION-B

SUBJECTIVE QUESTIONS

1) What is an interface? How it is used to create constants and define functions.
2) Explain about defining and implementing interfaces with example program.
3) Differentiate between class and interfaces
4) Write a short note on i) variable in interfaces ii) Nested interfaces
5) Can we extend interfaces? Support your argument.
6) Can JAVA does support multiple inheritance? Justify your answer.
7) Explain the concept of applying interfaces?
8) Define package? Write the procedure to create and import user defined package.
9) Explain different access specifiers supported by JAVA with an example program.
10) What is CLASSPATH? Explain its role in finding packages.

Write a java program to

1) Compute the area of rectangle using interfaces
2) Implement Multiple Inheritance using interface
3) Show how a class implements two interfaces.
4) show that the variables in an interface are implicitly static and final and methods are automatically public
5) Implements the extended interfaces
6) Create a package to display the given string in reverse order.
7) Create a package for Book details giving Book Name, Author Name, Price and Year of Publishing.
8) Write a java program to Create and access a user defined package where the package contains a class named CircleDemo, which in turn contains a method called circleArea() which takes radius of the circle as the parameter and returns the area of the circle.

UNIT-IV-
Learning Material(R-17)
Exception Handling and Multithreading

Objective:

To familiarize the concepts of Exception Handling and Multithreading.

Syllabus:

Exception Handling- exception-handling fundamentals, uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws, finally, user-defined exceptions.

Multithreading-Introduction to multitasking, thread life cycle, creating threads, synchronizing threads, thread groups.

Learning Outcomes

Upon successful completion of the course, the students will be able to

- Understand the concepts and applications of exception handling.
- Apply exception handle mechanism to handle run time errors in java.
- Write a program to handle multiple exception.
- Create user defined exception.
- Understand threads concepts and its life cycle in java.
- Understand how multiple threads can be created within java program.
- Apply threads concept to an application.

Learning Material

➢ EXCEPTION-HANDLING FUNDAMENTALS:

- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.
- 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 is *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.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system.
- To manually throw an exception, use the keyword **throw**. Any exception that is not being handled must be specified as such by a **throws** clause.
- Any code that absolutely must be executed before a method returns is put in a **finally** block.

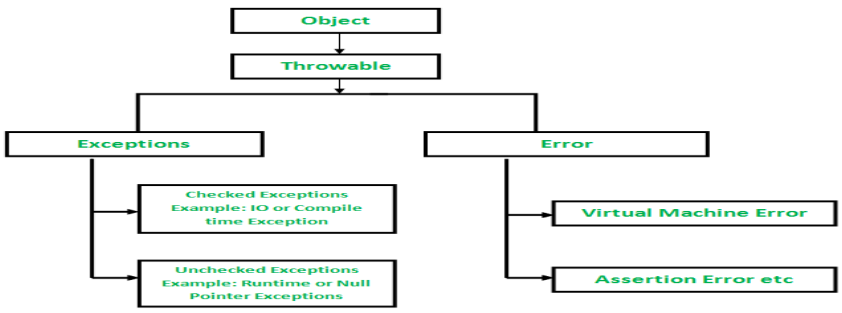
This is the general form of an exception-handling block:

```
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 before try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- 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**. 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.
- The other branch is **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.



- Exceptions are broadly classified into two categories
    - Checked Exceptions: Checked Exceptions are those for which the compiler checks to see whether they have been handled in your programs or not. These Exceptions are not sub classes of class RuntimeException.
    - Unchecked Exceptions: Run –Time exceptions are not checked by the compiler. These Exceptions are derived from class RuntimeException.

➢ UNCAUGHT EXCEPTIONS

example :

Program includes an expression that causes a divide-by-zero error.

```
class Exc0 {
public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- Notice how the class name, Exc0; the method name, main; the filename, Exc0.java; and the line number, 4, are all included in the simple stack trace.
- The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from main( ):

```
class Exc1 {
```

```
static void subroutine() {
        int d = 0;
        int a = 10 / d;
}
public static void main(String args[]) {
        Exc1.subroutine();
}
}
```

- The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

- As you can see, the bottom of the stack is main's line 7, which is the call to subroutine( ), which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

➢ USING TRY AND CATCH

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.
- To illustrate how easily this can be done, the following program includes a try block and a catch clause which processes the ArithmeticException generated by the division-by-zero error:

```
class Exc2 {
```

```
public static void main(String args[]) {
        int d, a;
try { // monitor a block of code.
        d = 0;
        a = 42 / d;
        System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
        System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

- This program generates the following output:

```
Division by zero.
After catch statement.
```

- Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Put differently, catch is not "called," so execution never "returns" to the try block from a catch.
- Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism. A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.

➢ MULTIPLE CATCH CLAUSES

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.
- The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

- This program will cause a division-by-zero exception if it is started with no command-line parameters, since a will equal zero. It will survive the division if you provide a commandline argument, setting a to something larger than zero.

- But it will cause an ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

Here is the output generated by running it both ways:

```
C:\\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:
42
After try/catch blocks.
```

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their super classes.
- This is because a catch statement that uses a super class will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its super class. Further, in Java, unreachable code is an error. For example, consider the following program:

```
class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
} catch(Exception e) {
System.out.println("Generic Exception catch.");
```

```
}
catch(ArithmeticException e) { // ERROR - unreachable
System.out.println("This is never reached.");
}
}
}
```

- If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable.

- Since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException.

- This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

➢ NESTED TRY STATEMENTS

- The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack.

- If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.

- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested try statements:

```
// An example of nested try statements.
class NestTry {
public static void main(String args[]) {
try {
        int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block

if(a==1) a = a/(a-a); // division by zero

if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);

}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

The program works as follows.

- When you execute the program with no command-line arguments, a divide-byzero exception is generated by the outer try block. Execution of the program by one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block,

where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block. Here are sample runs that illustrate each case:

```
C:\\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException: 42
```

- Nesting of try statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a try block. Inside that method is another try statement. In this case, the try within the method is still nested inside the outer try block, which calls the method. Here is the previous program recoded so that the nested try block is moved inside the method nesttry( ):

```
class MethNestTry {
static void nesttry(int a) {
try { // nested try block

if(a==1) a = a/(a-a); // division by zero

if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
```

```
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
}
public static void main(String args[]) {
try {
int a = args.length;

int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

➢ THROW:

- It is possible to throw an exception explicitly, not only catching exceptions that are thrown by the Java run-time system.using the throw statement. The general form of throw is shown here:

  throw *ThrowableInstance*;

- Here, *ThrowableInstance* must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter into a catch clause, or creating one with the new operator.

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception.

- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

- Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

- This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ).

- The demoproc( ) method then sets up another exception-handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown.

Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

throw new NullPointerException("demo");

- Here, new is used to construct an instance of NullPointerException. All of Java's built-in run-time exceptions have two constructors: one with no parameter and one that takes a string parameter.

- When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( ) or println( ).** It can also be obtained by a call to **getMessage( ),** which is defined by Throwable.

➢ THROWS:

- If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration.

- A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions(checked), except those of type **Error** or **RuntimeException**, or any of their subclasses.

- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

- To make this example compile, you need to make two changes. First, you need to declare that throwOne( ) throws IllegalAccessException. Second, main( ) must define a try/catch statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

output:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

➢ FINALLY:

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely.

- This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism.

- The finally keyword is designed to address this contingency. finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.

- The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

- The finally clause is optional. However, each try statement requires at least one catch or a finally clause. Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

class FinallyDemo
{
static void procA()
{
try {
	// Execute a try block normally.
	static void procC()
	{
	try {
	System.out.println("inside procC");

```
System.out.println("inside procA");
        throw new RuntimeException("demo");
    } finally
      {

System.out.println("procA's finally");
       } }
// Return from within a try block.

static void procB() {
try {
System.out.println("inside procB");
return;
} finally
 {
System.out.println("procB's finally");
}
}
```

```
} finally {
System.out.println("procC's finally");
} }
public static void main(String args[])
{
try {
procA();
}
catch (Exception e)
{
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

- In this example, procA( ) prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB( )'s try statement is exited via a return statement.

- The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed.

Note : If a finally block is associated with a try, the finally block will be executed upon conclusion of the try. Here is the output generated by the preceding program:

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally.

➢ USER-DEFINED EXCEPTIONS

- Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**.

- Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they need not be included in any method's throws list.
- In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed in the Table 1.

- Table 2 lists those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called

checked exceptions. Java defines several other types of exceptions that relate to its various class libraries.

Table-1

Java's **Unchecked** RuntimeException Subclasses

Exception	**Meaning**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string
UnsupportedOperationException	An unsupported operation was encountered

Table-2

Java's **Checked** Exceptions Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).

- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. 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. They are shown in Table below. these methods can be overridden in the created exception classes.

The Methods by Throwable

Method	Description

Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
String getLocalizedMessage( )	Returns a localized description of the exception
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
Void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
Void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
String toString( )	Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

- 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 the description of the exception to be displayed using println( ).

// This program creates a custom exception type. class MyException extends Exception	class ExceptionDemo { static void compute(int a) throws MyException {

```
{
private int detail;
MyException(int a)
{
detail = a;
}

public String toString()
 {
return "MyException[" + detail + "]";
}

}
```

```
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);
}}}
```

- This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays the value of the exception.

- The ExceptionDemo class defines a method named compute( ) that throws a MyException object. The exception is thrown when compute( )'s integer parameter is greater than 10. The main( ) method sets up an exception handler for MyException, then calls compute( ) with a legal value (less than 10) and an illegal one to show both paths through the code.

Output:

Called compute(1)

Normal exit
Called compute(20)
Caught MyException[20]

➢ INTRODUCTION TO MULTITASKING

- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- However, there are two distinct types of multitasking: process-based and thread-based.
- A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently.

- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.

- Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces.

- Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

➢ THREAD LIFECYCLE:

A Thread in its lifetime goes through various states.

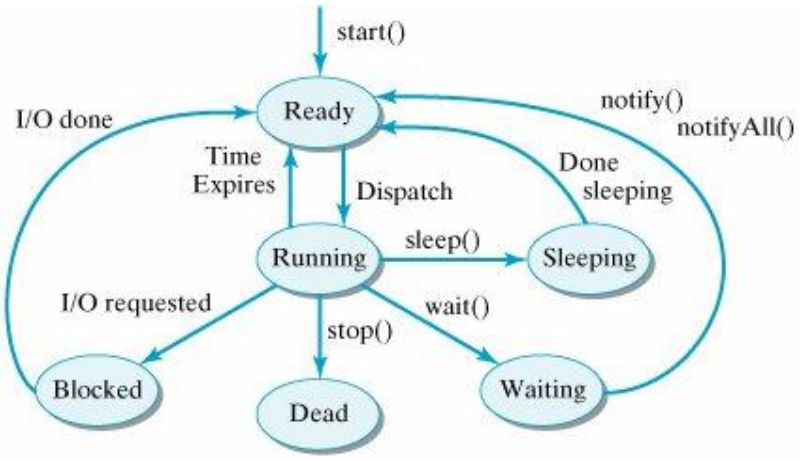
New : When we create a thread naturally, it is in "new" state. The thread is not yet ready to run. The only method can be called from this state is start(). This method moves to the ready state from which it is automatically moved to runnable state by thread scheduler.

Ready: The thread is ready to run (runnable) and waiting to be assigned to a processor by the scheduler. When the thread enters this state first time, it calls start() method from New state.

Running: A thread executing in the JVM is in running state. The state can be entered from ready state only when scheduled by the scheduler.

Blocked:A thread is blocked waiting for a monitor lock is in this state .A thread can enter waiting state from running state on any of the following events, like suspend, sleeping, waiting, joining and blocked.

Dead: The thread is destroyed when its run() method completes either normally or abnormally or destroy() or stop() method is called from any state.



Thread Life Cycle

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.

- There are 2 ways to create a new thread, either extend the Thread class or implement the Runnable interface.

- The Thread class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

➢ CREATING A THREAD:

A thread is created by instantiating an object of type **Thread**.

Java defines two ways in which this can be accomplished:

• implement the **Runnable** interface.

• extend the **Thread** class.

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

  public void run( )

- Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns. After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors.

  Thread(Runnable *threadOb*, String *threadName*)
- In this constructor, *threadOb* is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*. After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ).

  The start() method is shown here: void start( )

Here is an example that creates a new thread and starts it running:

```
class NewThread implements Runnable
{
Thread t;
NewThread()
{
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start();
}

public void run() {
try {
for(int i = 5; i > 0; i—) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
}
catch (InterruptedException e)
{
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

```
class ThreadDemo
{
public static void main(String args[])
{
new NewThread();
try {
for(int i = 5; i > 0; i—)
{
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e)
{
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Inside NewThread's constructor, a new Thread object is created by the following statement:

t = new Thread(this, "Demo Thread");

- Passing this as the first argument indicates that you want the new thread to call the run() method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread's for loop to begin.

- After calling start(), NewThread's constructor returns to main( ). When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

The output produced by this program is as follows:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

- In a multithreaded program, the main thread must be the last thread to finish running. If the main thread finishes before a child thread has completed, then the Java run-time system may "hang."

Extending Thread

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

Here is the preceding program rewritten to extend **Thread**:

```
class NewThread extends Thread
 {
NewThread()
{
super("Demo Thread");
System.out.println("Child thread: " +
this);
start();
}
public void run() {
try {
for(int i = 5; i > 0; i—) {
System.out.println("Child Thread: " +
i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
```

```
class ExtendThread
{
public static void main(String args[])
{
new NewThread();
try {
for(int i = 5; i > 0; i—)
{
System.out.println("Main Thread: " +
i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread
interrupted.");
}
System.out.println("Main thread
```

```
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

```
exiting.");
}
}
```

- This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of NewThread, which is derived from Thread.

- The call to super( ) inside NewThread invokes the following form of the Thread constructor:

  public Thread(String *threadName*)

  Here, *threadName* specifies the name of the thread.

The Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

  • It is the thread from which other "child" threads will be spawned.
  • It must be the last thread to finish execution.

- The main thread is created automatically when program is started, which can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread.

Its general form is shown here**: static Thread currentThread( )**

- This method returns a reference to the thread in which it is called.

- By default, the name of the main thread is main. Its priority is 5, which is the default value, and main is also the name of the group of threads to which this thread belongs.

- A *thread group* is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment

- The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

static void sleep(long *milliseconds*) throws InterruptedException

- The number of milliseconds to suspend is specified in *milliseconds.* This method may throw an **InterruptedException**.

static void sleep(long *milliseconds*, int *nanoseconds*) throws InterruptedException

- The sleep( ) method has a second form, which allows you to specify the period in terms of milliseconds and nanoseconds.

You can set the name of a thread by using **setName( )**.

You can obtain the name of a thread by calling **getName( )**

These methods are members of the **Thread** class and are declared like this:

final void setName(String *threadName*)

final String getName( )

Creating Multiple Threads

- In addition to the main thread and one child thread, a program can spawn as many threads as it needs.

For example, the following program creates three child threads:

```
class NewThread implements Runnable
{
String name; // name of thread
Thread t;
NewThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
public void run()
 {
try {
for(int i = 5; i > 0; i—) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{ System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}
}
class MultiThreadDemo
{
public static void main(String args[])
{
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
 try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

The output from this program is shown here: New thread: Thread[One,5,main] New thread: Thread[Two,5,main] New thread: Thread[Three,5,main] One: 5 Two: 5 Three: 5 One: 4 Two: 4 Three: 4 One: 3 Three: 3 Two: 3	One: 2 Three: 2 Two: 2 One: 1 Three: 1 Two: 1 One exiting. Two exiting. Three exiting. Main thread exiting.

- As you can see, once started, all three child threads share the CPU. Notice the call to sleep(10000) in main( ). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive() and join()

- Two ways exist to determine whether a thread has finished. First, you can call isAlive( ) on the thread. This method is defined by Thread, and its general form is shown here:

  final boolean isAlive( )

- The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.
- The method more commonly used to wait for a thread to finish is called join( ), shown here:

final void join( ) throws InterruptedException

- This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.

```
class DemoJoin
{
public static void main(String args[])
 {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();        ob2.t.join();              ob3.t.join();
}
catch (InterruptedException e)
{    System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
```

```
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

- As you can see, after the calls to join( ) return, the threads have stopped executing.

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- Higher-priority threads get more CPU time than lower-priority threads. But in practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (Ex: OS, CPU time.etc)
- To set a thread's priority, use the setPriority( ) method, which is a member of Thread. This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread.

- The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively.

- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as final variables within Thread. You can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

final int getPriority( )

➢ SYNCHRONIZATION

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.*
- A *monitor* is an object that is used as a mutually exclusive lock, or *mutex*. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

Using Synchronized Methods

- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.
- The following program has three simple classes. The first one, Callme, has a single method named call( ).
- The call( ) method takes a String parameter called msg. This method tries to print the msg string inside of square brackets. The interesting thing to notice is that after call( ) prints the opening bracket and the msg String, it calls Thread.sleep(1000), which pauses the current thread for one second.
- The constructor of the next class, Caller, takes a reference to an instance of the Callme class and a String, which are stored in target and msg, respectively. The constructor also creates a new thread that will call this object's run( ) method.

- The thread is started immediately. The run( ) method of Caller calls the call( ) method on the target instance of Callme, passing in the msg string.

- Finally, the Synch class starts by creating a single instance of Callme, and three instances of Caller, each with a unique message string. The same instance of Callme is passed to each Caller.

```
// This program is not synchronized.
class Callme {
void call(String msg) {
```

```
public void run()
 {
target.call(msg);
}
```

```
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted"
);
}
System.out.println("]");
}
}
class Caller implements
Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String
s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
```

```
}
class Synch {
public static void main(String
args[]) {
Callme target = new Callme();
Caller ob1 = new
Caller(target, "Hello");
Caller ob2 = new
Caller(target, "Synchronized");
Caller ob3 = new
Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException
e) {
System.out.println("Interrupt
ed");
}
}
}
```

Output: Hello[Synchronized[World]

- By calling sleep( ), the call( ) method allows execution to switch to another thread.

- This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the

same method, on the same object, at the same time. This is known as **a *race condition***, because the three threads are racing each other to complete the method.

- To fix the preceding program, you must *serialize* access to call( ). That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede call( )'s definition with the keyword synchronized, as shown here:

```
class Callme {
synchronized void call(String msg) {
                                                    …}
```

- This prevents other threads from entering call( ) while another thread is using it. After synchronized has been added to call( ), the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

- Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the synchronized keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, non synchronized methods on that instance will continue to be callable.

The synchronized Statement

- While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not

work in all cases. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods.

- Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add synchronized to the appropriate methods within the class. How can access to an object of this class be synchronized? Solution is to place calls to the methods defined by this class inside a synchronized block. This is the general form of the synchronized statement:

```
synchronized(object) {
// statements to be synchronized
}
```

- Here, *object* is a reference to the object being synchronized. If you want to synchronize only a single statement, then the curly braces are not needed. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

- Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```
// This program uses a synchronized block.

class Callme
{
void call(String msg)
{
System.out.print("[" + msg);
```

```
// synchronize calls to call()
public void run()
{
synchronized(target)
 { // synchronized block
target.call(msg);
}
}
```

```
try {
Thread.sleep(1000);
} catch (InterruptedException e)
{
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s)
{
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
```

```
}
class Synch1
{
public static void main(String args[])
{
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e)
 {
System.out.println("Interrupted");
}
}
}
```

- Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Inter Thread Communication

- Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time

- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized method.

• wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).

• notify( ) wakes up the first thread that called wait( ) on the same object.

• notifyAll( ) wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

These methods are declared within Object, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

- Additional forms of wait( ) exist that allow you to specify a period of time to wait. The following sample program incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: Q, the queue that you're trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

// An incorrect implementation of	public void run()

```
a producer and consumer.
class Q
 {
int n;
synchronized int get()
{
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n)
 {
this.n = n;
System.out.println("Put: " + n);
} }
class Producer implements
Runnable
 {
Q q;
Producer(Q q)
 {
this.q = q;
new Thread(this,
"Producer").start();
}
public void run()
{
int i = 0;
while(true)
 {
q.put(i++);
}}}
{
while(true)
{
q.get();
}
}
}


class PC
 {
public static void main(String args[])
{
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C
to stop.");
}
}
OUTPUT:
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
```

class Consumer implements Runnable
{
Q q;
Consumer(Q q)
{
this.q = q;
new Thread(this, "Consumer").start(); }
	Put: 6
	Put: 7
	Got: 7

- Although the put( ) and get( ) methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

- As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.
- The proper way to write this program in Java is to use wait( ) and notify( ) to signal in both directions, as shown here:

// A correct implementation of a producer and consumer.
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if(!valueSet)
	class Consumer implements Runnable {
	Q q;
	Consumer(Q q) {
	this.q = q;
	new Thread(this, "Consumer").start();

```
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException
caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
if(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException
caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}

public void run() {
while(true) {
q.get();
}
}
}
class PCFixed {
public static void main(String
args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press
Control-C to stop.");
}
}
```

OUTPUT:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

```
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
```

- Inside get( ), wait( ) is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside get( ) resumes. After the data has been obtained, get( ) calls notify( ). This tells Producer that it is okay to put more data in the queue. Inside put( ), wait( ) suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and notify( ) is called. This tells the Consumer that it should now remove it. Here is some output from this program, which shows the clean synchronous behavior:

➢ THREADGROUP

- **ThreadGroup** creates a group of threads. It defines these two constructors:

  ThreadGroup(String *groupName*)

  ThreadGroup(ThreadGroup *parentOb*, String *groupName*)

- For both forms, *groupName* specifies the name of the thread group. The first version creates a new group that has the current thread as its parent. In the second form, the parent is specified by *parentOb.*
- ThreadGroup also included the methods stop( ), suspend( ), and resume( ).
- These have been deprecated by Java 2 because they were inherently unstable. Thread groups offer a convenient way to manage groups of

threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads.

- For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience.

UNIT-IV
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Identify the parent class of all the exception in java is [ ]
   a)Throwable b)Throw c) Exception d)Throws

2. What are the two types of exception available in java ? [ ]
   a)Checked and compiled b) Un Checked and compiled
   c)Checked and Un Checked d) Compiled and non- compiled

3. The two subclasses of Throwable are [ ]
   a)Error and AssertionError
   b)Error and Exception
   c)Checked and UnChecked Exception
   d)Error and Runtime Exception

4. Choose the correct option regarding notifyAll() method. [ ]
   a) Wakes up one threads that are waiting on this object's monitor
   b) Wakes up all threads that are not waiting on this object's monitor
   c)Wakes up all threads that are waiting on this object's monitor
   c) None of the above

5. Identify the keyword when applied on a method indicates that only one thread should execute the method at a time. [ ]
   a)volatile b) synchronized c) native d) static

6. The built-in base class in Java, which is used to handle all exceptions is [ ]
   a)Raise
   b)Exception
   c)Error
   d)Throwable

7. Which of the following exceptions is thrown when one thread has been interrupted by another thread? [ ]
a)ClassNotFoundException
b)IllegalAccessException
c)InstantiationException
d)InterruptedException
e)NoSuchFieldException

8. Which of the following Exception classes in Java is used to deal with an exception, where an assignment to an array element is of incompatible type? [ ]
a)ArithmeticException
b)ArrayIndexOutOfBoundsException
c)IllegalArgumentException
d)ArrayStoreException
e)IllegalStateException

9. A programmer has created his own exception for balance in account <1000. The exception is created properly, and the other parts of the programs are correctly defined. Though the program is running but error message has not been displayed. Why did this happen? [ ]
a)Because of the Throw portion of exception.
b)Because of the Catch portion of exception.
c)Because of the main() portion.
d)Because of the class portion.
e)None of the above

10. Choose the correct option for the following program [ ]

```
class demo
{
        void show() throws CalssNotFoundException{}
}
class demo2 extends demo
{
void show() throws IllegalAccessException, classNotFoundException,
ArithmeticException
{
        System.out.println("In Demo1 show");
}
         public static void main(String arg[])
{
try{
        demo2   d=new demo2();
        d.show();
  }
```

```
catch(Exception e)  {}
        }
 }
```

a.Does not compile
b.Compiles successfully
c.Compiles successfully and prints "In Demo1 show"
d.Compiles but does not execute.

11. If the assert statement returns false, what is thrown? [ ]
a)Exception b) Assert c) assertion d) assertion Error

12. Choose the best possible answer for the following program [ ]

```
class demo
{
        void show() throws ArithmeticException
          { }
}
class demo2 extends demo
{
        void show()
        {
        System.out.println("In Demo1 show");
}
public static void main(String arg[])
{
        demo2 d=new demo2();
        d.show();
}}
```

a.Does not compile
b.Compiles successfully
c.Compiles successfully and prints "In Demo1 show"
b.Compiles but does not execute.

13. How can Thread go from waiting to runnable state? [ ]
a)notify/notifAll
b)bWhen sleep time is up
c)Using resume() method when thread was suspended
d)All

14. Predict the output of the following program [ ]

```
class A implements Runnable{
    public void run(){
        try{
            for(int i=0;i<4;i++){
                Thread.sleep(100);
                System.out.println(Thread.currentThread().getName());
            }
        }catch(InterruptedException e){
```

```
            }
        }
    }

    public class Test{
        public static void main(String argv[]) throws Exception{
            A a = new A();
            Thread t = new Thread(a, "A");
            Thread t1 = new Thread(a, "B");
            t.start();
            t.join();
            t1.start();
        }
    }
```

a) A A A A B B B B b) A B A B A B A B

c) Output order is not guaranteed d) Compilation succeed but Runtime Exception

15. What will be output of the following program code? [ ]

```
public class Test implements Runnable{
  public void run(){
 System.out.print("go");
 }
public static void main(String arg[]) {
     Thread t = new Thread(new Test())
    t.run();
        t.run();
        t.start();
    }
}
```

a) Compilation fails.

b) An exception is thrown at runtime

c) go" is printed

d) “gogo" is printed

16. Choose the correct option for Deadlock situation [ ]

a) Two or more threads have circular dependency on an object

b) Two or more threads are trying to access a same object

c)Two or more threads are waiting for a resource

d) None of these

17. Predict the output of following Java program [ ]

```
class Main {
  public static void main(String args[]) {
    try {
      throw 10;
    }
    catch(int e) {
      System.out.println("Got the  Exception " + e);
    }
  }
}
```

a) Got the Exception 10

b)Got the Exception 0

c)Compiler Error

d)None of the above

18. What is the output of the following program [ ]

```
class Test extends Exception {}
  class Main {
   public static void main(String args[]) {
    try {
      throw new Test();
    }
    catch(Test t) {
      System.out.println("Got the Test Exception");
    }
    finally {
      System.out.println("Inside finally block ");
    }
  }
}
```

a)Got the Test Exception Inside finally block

b)Got the Test Exception
c)Inside finally block
d)Compile error.

19. What is the output of the following program [ ]

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            int a[]= {1, 2, 3, 4};
            for (int i = 1; i <= 4; i++)
            {
                System.out.println ("a[" + i + "]=" + a[i] + "n");
            }
        }

        catch (Exception e)
        {
            System.out.println ("error = " + e);
        }

        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println ("ArrayIndexOutOfBoundsException");
        }
    }
}
```

a) Compiler error
b)Run time error
c)ArrayIndexOutOfBoundsException
d)Error Code is printed
e)Array is printed

20. Predict the output of the following program. [ ]

```
class Test
{   int count = 0;

    void A() throws Exception
    {
        try
        {
            count++;

            try
            {
                count++;
```

```
            try
            {
               count++;
               throw new Exception();

            }

            catch(Exception ex)
            {
               count++;
               throw new Exception();
            }
         }

         catch(Exception ex)
         {
            count++;
         }
      }

      catch(Exception ex)
      {
         count++;
      }

   }

   void display()
   {
      System.out.println(count);
   }

   public static void main(String[] args) throws Exception
   {
      Test obj = new Test();
      obj.A();
      obj.display();
   }
}
```

a)4 b)5 c)6 d)Compile Error

SECTION-B

Descriptive Questions

1. Define Exception? What are the three categories of exceptions? Also discuss the advantages of exception handling

2. Explain the keywords used in exception handling.
3. Implement a multiple exception handling for the following problem
   Read n+1 strings to string array and prints their lengths to get ArrayIndexOutOfBoundsException and NullPointerException
4. Write a java program to calculate the student total marks and percentage for class test with six subjects. The marks should be 0 to 10 only, if marks entered not in the range then raise an exception MarksNotInRangeException.**(**Create user defined exception and throw it).
5. Can a try block be written without a catch block? Justify.
6. Can we nest a try statement inside another try statement. Write the necessary explanation and example for this.
7. Differentiate multi tasking and multithreading.
8. Draw a neat sketch of thread life cycle.
9. What is synchronization and how do we use it in java.
10. Write a Java program to create two threads from main such that one thread calculates the factorial of a given number and another thread checks whether the given number is prime or not.
11. Write a Java program to print the messages in the following sequence

    For every 3 seconds " Welcome" message

    For every 2 seconds "Hello" message

    For every 5 seconds " Bye" message

APPLETS AND EVENT HANDLING

Objective:

- To get acquainted with the concepts of Applet and Event Handling.

Syllabus:

Applets: Concepts of Applets, Differences between applets and applications, life cycle of an applet, creating applets.

Event Handling: Events, Event sources, Event classes, Event Listeners, Delegation event model, Handling mouse and keyboard events, Adapter classes.

Learning Outcomes:

Students will be able to

- Understand the concept of Applet.
- Differentiate between Applets and Application.
- Understand the lifecycle of an applet
- Learn how applets are Created and executed.
- Understand what are events, sources and listeners.
- Know about their event classes and associated listeners.
- Know the Fundamentals of Event Handling
- Understand how the Mouse and Keyboard Events are handled.
- Know the uses of Adapter Classes.

Learning Material

5.1 Applets:

- **Applets** are small programs that are primarily used in internet programming.
  - Applet programs are either developed in local systems or in remote systems
  - Applet programs are executed by either a java compatible "web browser" or "appletviewer".
- Applets are classified into two types as Local Applet and Remote Applet.
- An applet developed locally and stored in a local system is known as a **Local Applet.**
  - When a Web page is trying to find a local applet, it does not need to use the Internet and therefore the local system does not require the Internet connection.
- An Applet developed by someone else and stored on a remote computer is known as **Remote Applet**.
  - If our system is connected to the Internet, we download the remote applet onto our system via Internet and run it.
  - In order to locate the remote applet, we must know the applets address on the Web. This address is known as Uniform Resource Locator (URL).

5.1.1 Advantages of Applet:

There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

5.1.2 Drawback of Applet :

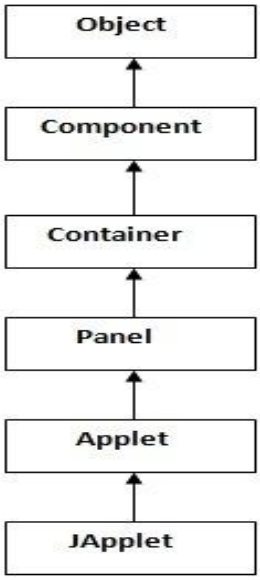
- Plug-in is required at client browser to execute applet.

5.1.3 Differences between applets and applications, life cycle of an applet

Applet	Application
1. Small Program	Large Program
2. Used to run a program on client Browser	Can be executed on standalone computer system
3. Applet is portable and can be executed by any JAVA supported browser.	Need JDK, JRE, JVM installed on client machine.
4. Applet applications are executed in a Restricted Environment	Application can access all the resources of the computer
5. Applets are created by extending the java.applet.Applet	Applications are created by writing public static void main(String[] s) method.
6. Applet application has 5 methods which will be automatically invoked on occurrence of specific event	Application has a single start point which is main() method

7. Example:
`import java.awt.*;`
`import java.applet.*;`
`public class Myclass extends Applet`
`{`
`public void init() { }`
`public void start() { }`
`public void stop() { }`
`public void destroy() { }`
`public void paint(Graphics g) { }`
`}`
	`public class MyClass`
	`{`
	`public static void main(String args[])`
	`{`
	`}`
	`}`

5.1.4 Hierarchy of Applet:



5.1.5 Life cycle of an applet:

- Every java applet inherits a set of default behaviors from the Applet class defined in java.applet package.
- When an applet is loaded, it undergoes a series of changes in its states.
- The important states of the Applet cycle is

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

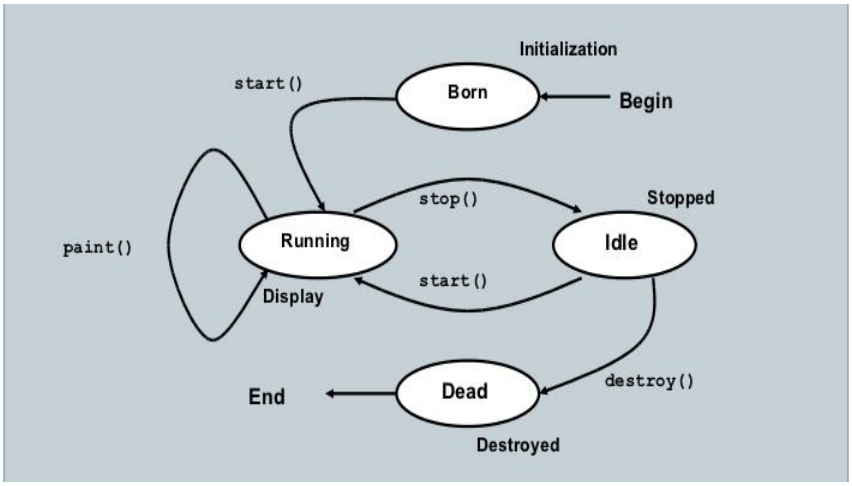


Figure 5.1 Applet Life Cycle

- Five methods in the Applet class give you the framework on which you build any serious applet:
  - **Initialization State:** Applet enters the initialization state when it is first loaded. This is achieved by calling **init( )** method of **Applet** class. The initialization occurs only once in the applet life cycle. Generally all the initialization variables are to be placed in the init () method.

    Syntax:

    ```
    public void init ( )
    {
            ------(Action)
    }
    ```

  - **Running State:** Applet enters the running state when the system calls the **start( )** method of **Applet** class. This occurs automatically after the applet is initialized. Starting can also occurs if the applet is already in "Stopped State". The start () method may be called more than once.

    Syntax:

    ```
    public void start( )
    {
            ----- (Action)
    }
    ```

- **Idle or Stopped State:** An Applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. This can also done by calling the stop () method explicitly.

  Syntax:

  ```
  public void stop ()
  {
        ----- (Action)
  }
  ```

- **Dead or Destroyed State:** An applet is said to be dead when it is removed from memory. This occurs automatically by invoking the destroy() method when we quit the browser. Destroying stage occurs only once in the applet life cycle.

  Syntax:

  ```
  public void destroy ()
  {
        ------- (Action)
  }
  ```

- **Display state:** Display state is useful to display the information on the output screen. This happens immediately after the applet enters into the running state. The paint () is called to accomplish this task. Almost every applet will have a paint () method.

  Syntax:

  ```
  public void paint(Graphics g)
  {
        ------
        ------ (Display statements)
  }
  ```

5.1.6 Creating applets:

- To create an Applet program follow the steps:
  1. Building an applet code (.java file)
  2. Creating an executable applet (.class file)
  3. Create HTML page with the <APPLET>tag.
  4. Testing the Applet code with applet viewer or browser.

Example

//First.java

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
  public void paint(Graphics g)
  {
    g.drawString("welcome",150,150);
  }
}
```

//myapplet.html

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

- To run the Applet in browser:
  - create an applet that contains applet tag in comment and compile it.

    javac First.java
  - Double click on the HTML file. It will open the Applet in browser.

//First.java

```
/*
<applet code="First.class" width="300" height="300">
</applet>
*/  import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet
{
  public void paint(Graphics g)
  {
    g.drawString("welcome to applet",150,150);
  }
}
```

- To execute the applet by appletviewer tool:
  - create an applet that contains applet tag in comment and compile it.

javac First.java

- After that run it by:

appletviewer First.java.

5.2 Event Handling:

5.2.1 Events:

- An event is an object that describes a state change in a source.
- Even driven is a consequence interaction of the user with the GUI. Some of the common interactions are moving the mouse, clicking the mouse, clicking a button, typing in a textfield etc.,

5.2.2 Event Sources:

- A source is an object that generates an event. This occurs when the internal state of that object changes in some way.
- Source may generate more than one type of event.

5.2.3 Event Classes:

- Event classes are used to handle java event handling mechanism.
- At the root of the java event class hierarchy is **EventObject,** which is in java.util package. It is the superclass for all events. It provides a constructor as **EventObject(Object x)**
- Methods provided by the class are
  1. **Object getSource():** returns the source of the event
  2. **String toString ():** returns the string equivalent of the event.
- The class AWTEvent, defined within the java.awt package, is subclass of EventObject.
- It is the super class of all AWT-based events used by the delegation model.
- Commonly used Event classes are

  ActionEvent,
  AdjustmentEvent,
  ComponentEvent,
  ContainerEvent,
  FocusEvent,
  InputEvent,
  ItemEvent,

KeyEvent,
MouseEvent,
MouseWheelEvent,
TextEvent,
WindowEvent.

5.2.4 Event Listeners:

- A listener is an object that is notified when an event occurs. It has two requirements
- It must have been registered with one or more sources to receive notifications about the specific type of events.
- It must implement methods to receive and process the notifications. The methods that receive and process events are defined in a set of interfaces found in java.awt.event package.
- Commonly used Event Listener Interfaces are

ActionListener,
AdjustmentListener
ComponentListener
ContainerListener
FocusListener
ItemListener
KeyListener
MouseListener
MouseMotionListener
TextListener
WindowFocusListener
WindowListener

5.2.5 Delegation Event Model:

- The Event Delegation Model defines a consistent mechanism to generate and process the events.
- The process is: **a source generates an event and sends it to one or more listeners.**
- In this scheme, the listener simply waits until it receives an event.
- Once an event is received, the listener processes the event and then returns.

5.2.6 Mouse Events:

5.2.6.1 MouseEvent Class:

- The MouseEvent class defines 8 types of mouse events. It defines the following integer constants that can be used to identify them:
  1. MOUSE_CLICKED : The user clicked the mouse
  2. MOUSE_DRAGGED : The user dragged the mouse
  3. MOUSE_ENTERED : The mouse entered a component
  4. MOUSE_EXITED : The mouse exited from a component
  5. MOUSE_MOVED : The mouse moved
  6. MOUSE_PRESSED : The mouse was pressed
  7. MOUSE_RELEASED : The mouse was released
  8. MOUSE_WHEEL : The mouse wheel was moved
- **Methods** of MouseEvent Class**:**
  1. **int getX():** returns the X coordinates of the mouse when an event occurred
  2. **int getY():** returns the Y coordinates of the mouse when an event occurred.
  3. **Point getPoint()**: returns the coordinates of the mouse.

5.2.6.2 MouseListener Interface:

- This interface defines five methods:
  1. **void mouseClicked(MouseEvent me):** it invokes if the mouse is pressed and released at the same point
  2. **void mouseEntered(MouseEvent me):** it invokes when the mouse enters a component.
  3. **void mouseExited(MouseEvent me):** it invokes when the mouse leaves the component.
  4. **void mousePressed(MouseEvent me):** it invokes when the mouse is pressed.
  5. **void mouseReleased(MouseEvent me):** it invokes when the mouse is released.

5.2.6.3 MouseMotionListener Interface:

- This interface defines two methods:
  1. **void mouseDragged(MouseEvent me):** it invokes when multiple times as the mouse is dragged.
  2. **void mouseMoves(MovesEvent me):** it invokes when multiple times as the mouse is moved

Example program to Display the position of x and y co-ordinates of the cursor movement using mouse

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="Mouse" width=500 height=500>
   </applet> */
public class Mouse extends Applet implements MouseListener,
   MouseMotionListener
{
   int X = 0, Y = 20;
   String msg = "MouseEvents";
   public void init( )
   {
     addMouseListener(this);
     addMouseMotionListener(this);
     setBackground(Color.black);
     setForeground(Color.red);
   }
   public void mouseEntered(MouseEvent m)
   {
     setBackground(Color.magenta);
     showStatus("Mouse Entered");
     repaint( );
   }
   public void mouseExited(MouseEvent m)
   {
     setBackground(Color.black);
     showStatus("Mouse Exited");
     repaint( );
   }
   public void mousePressed(MouseEvent m)
   {
```

```
        X = 10;
        Y = 20;
        msg = "GEC";
        setBackground(Color.green);
        repaint( );
    }
    public void mouseReleased(MouseEvent m)
    {
        X = 10;
        Y = 20;
        msg = "Engineering";
        setBackground(Color.blue);
        repaint( );
    }
    public void mouseMoved(MouseEvent m)
    {
        X = m.getX( );
        Y = m.getY( );
        msg = "College";
        setBackground(Color.white);
        showStatus("Mouse Moved");
        repaint( );
    }
    public void mouseDragged(MouseEvent m)
    {
        msg = "CSE";
        setBackground(Color.yellow);
        showStatus("MouseMoved" + m.getX( ) + " " + m.getY( ));
        repaint( );
    }
    public void mouseClicked(MouseEvent m)
    {
        msg = "Students";
        setBackground(Color.pink);
        showStatus("Mouse Clicked");
        repaint( );
    }
    public void paint(Graphics g)
    {
        g.drawString(msg, X, Y);
    }
}
```

5.2.7 KeyEvents:

5.2.7.1 KeyEvent Class:

- A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants:
    1. KEY_PRESSED
    2. KEY_RELEASED
    3. KEY_TYPED
- The first two events are generated when any key is pressed or released. The last event occurs when a character is generated.

- **Methods** of KeyEvent Class**:**
    1. **char getKeyChar()** : returns the character that was enterd
    2. **int getKeyCode() :** returns the character code.

5.2.7.2 KeyListener Interface:

- The interface defines three methods:
    1. **void keyPressed(KeyEvent ke):** it invokes when a key is pressed.
    2. **void keyReleased(KeyEvent ke):** it invokes when a key is released.
    3. **void keyTyped(KeyEvent ke):** it invokes when a key is typed.

Example program to handle keyboard events, which echoes keystrokes to the applet window and shows the status of each key event in the status bar

```
/* <applet code="Key.class" width=4000 height=4000>
   </applet> */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class Key extends Applet implements KeyListener
{
   String msg = "";
```

```
    public void init( )
    {
      setBackground(Color.green);
      addKeyListener(this);
    }
    public void keyPressed(KeyEvent k)
    {
      showStatus("Key pressed");
      if(k.getKeyCode( ) == KeyEvent.VK_ENTER)
      showStatus("Enter key is pressed:");
      repaint( );
    }
    public void keyReleased(KeyEvent k)
    {
      showStatus("Key Up or Key Released");
    }
    public void keyTyped(KeyEvent k)
    {
      msg += k.getKeyChar( );
      repaint( );
    }

    public void paint(Graphics g)
    {
      g.setFont(new Font("Arial", Font.BOLD, 30));
      g.drawString("Key Typed is:" + msg, 20, 300);
    }
}
```

5.2.8 Adapter classes:

- An adapter class provides an empty implementation of all methods in an event listener interface.
- Adapter classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface.
- For this, we can define a new class to act as an event listener by extending one of the adapter classes and the **MouseMotionAdapter** class has two methods, **mouseDragged( ) and mouseMoved( ).**
- If we are interested in only mouse drag events, then we extend **mouseMotionAdapter** and implement **mouseDragged( ).** The empty

implementation of mouseMoved( ) would handle the mouse motion events.

- Adapter classes are provided by java.awt.event package. Some of the Adapter classes are

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	FocusListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Example program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
    <applet code="AdapterDemo.class" width=100 height=100>
    </applet>
*/
public class AdapterDemo extends Applet
{
    public void init( )
    {
      addMouseListener(new my(this));
      addMouseMotionListener(new mtadd(this));
    }
}
class My extends MouseAdapter
{
    AdapterDemo a;
    public My(AdapterDemo p) {
      This.a = p;
    }
    public void mouseClicked(MouseEvent me)
    {
```

```
        a.showStatus("Mouse Clicked");
    }
}
class MyAdd extends MouseMotionAdapter
{
    AdapterDemo a;
    public MyAdd(AdapterDemo p)
    {
        this.p;
    }
    public void mouseDragged(MouseEvent me)
    {
        a.showStatus("Mouse Dragged");
    }
}
```

UNIT-V

Assignment-Cum-Tutorial Questions

Section - A

Objective Questions

1) A Java _____ is a program that is executed by a Web browser
2) An HTML document uses the _____ tag to identify Java applets
3) What is the name of the method that is only called once whenever an applet is loaded into the Java Virtual Machine? [ ]

A. start B. Applet C. ActionEvent D. init

4) The _____ method of an applet is used to draw graphics and is invoked automatically when the applet runs.
5) A _____ has methods that tell what will happen when it receives an event
6) When the user clicks a button, the event will be handled by an object of type _____. [ ]

A)ActionListener B)EventHandler
C) ButtonListener D)ActionHandler

7)_____ class provides an empty implementation of all methods in an event listener interface.

8)Which of these packages contains all the event handling interface [ ]

A) java.lang B) java.awt C) java.awt.event D) java.event

9)The Applet class is in_package [ ]

A) java.applet B) java.awt C) java.io D) java.util

10) Which of these methods are used to register a keyboard event listener? [ ]

A) KeyListener() B) addKistener()

C) addKeyListener() D) eventKeyboardListener()

11) Which of these methods are used to register a mouse motion listener

A) addMouse() B) addMouseListener()

C) addMouseMotionListner() D) eventMouseMotionListener()

12) Which of these events will be generated if we close an applet's window?

A) ActionEvent B) ComponentEvent [ ]

C) AdjustmentEvent D) WindowEvent

13) Which of these is super class of all Adapter classes? [ ]

A) Applet B) ComponentEvent

C) Event D) InputEvent

SECTION -B

SUBJECTIVE QUESTIONS

1) Explain about the lifecycle of an applet with an example

2) Differentiate between applets and applications.

3) Write the steps involved in creating an applet with an example

4) Can we pass parameters to an Applet? If so, justify your answer with an example.

5) Explain in detail about Delegation event model and various events, event sources that are available in Java?

6) Demonstrate keyboard event handling with an example

7) Explain MouseEvent Class in detailed with an example

8) Write about Adapter classes and their importance in Event Handling

9) Write a program to Pass the parameters: Employee Name and ID Number to an applet

10) Create an Applet that displays the message like ”Hai Friends How are you..?” using <param >tag.

11) Create an applet having the background color as black and the foreground color as white.

UNIT –VI

AWT

Syllabus: The AWT class hierarchy, User interface components- label, button, Checkbox, checkboxgroup, Choice, list, textfield, Scrollbar. Layout managers- Flow, Border , Grid, Card , GridBag layout.

Objective: Design and implement an effective GUI for various applications

Learning Outcomes:

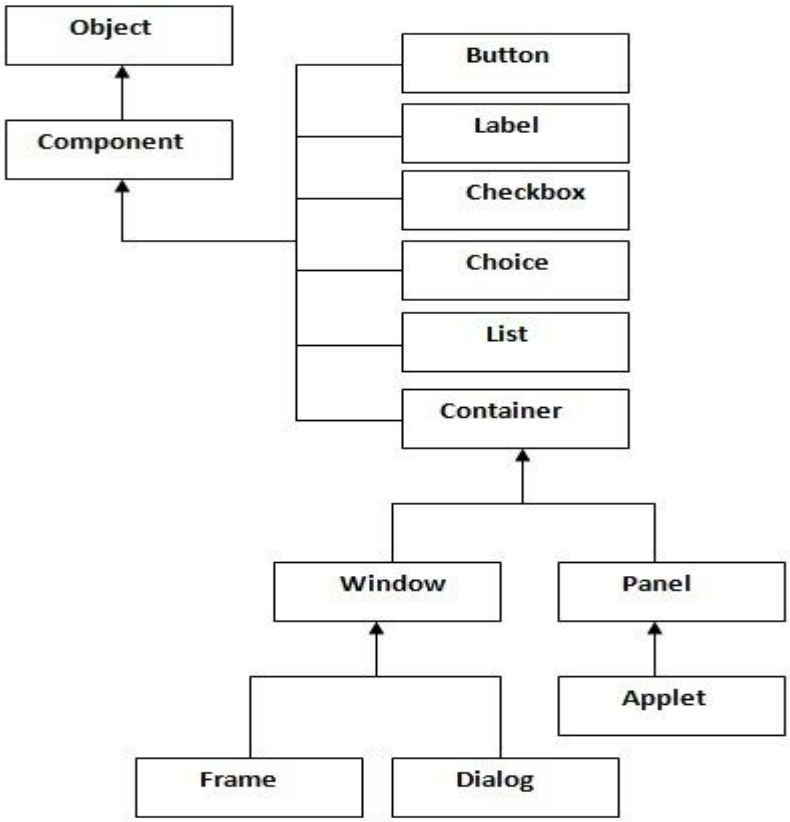
Students will be able to

- Create various AWT components like Buttons, Chechboxes, and List etc.
- Design a GUI containing various AWT Components
- Write Code to Handle events raised by the AWT components
- Apply Various Layout Managers to the GUI being designed.

6.1 Introduction to AWT:

- **Java AWT** (Abstract Window Toolkit) is *an API to develop GUI or window-based application in java.*
- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- AWT is heavyweight i.e. its components uses the resources of system.
- The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy



a. Component

- At the top of the AWT hierarchy is the Component class.
- Component is an abstract classthat encapsulates all of the attributes of a visual component.
- Except for menus, all user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.

- **Methods of Component class :**

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width, int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

b. Container

- The Container class is a subclass of Component.
- A container is responsible for laying out (that is, positioning) any components that it contains

c. Window

- The Window class creates a top-level window.
- A top-level window is not contained within any other object; it sits directly on the desktop.
- Generally, Window objects are not created directly.

d. Panel

- The Panel class is a concrete subclass of Container.
- Panel is a window that does not contain a title bar, menu bar, or border.
- Panel is the superclass for Applet.
- Other components can be added to a Panel object by its add( ) method (inherited from Container).
- Once these components have been added, you can position and resize them manually using the setLocation( ), setSize( ), setPreferredSize( ), or setBounds( ) methods defined by Component.

e. Frame

- Frame encapsulates what is commonly thought of as a “window.”
- It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.
- The precise look of a Frame will differ among environments

6.2 User Interface Components

- The **java.awt** package provides an integrated set of classes to manage user interface-components.
- The simplest form of Java AWT component is the basic User Interface Component.
- We can create and add these to your applet as it is an AWT container.
- We can put other AWT components, such as UI components or other containers, in it.
- The following table gives a list of all the Controls in Java AWT and their respective functions.

AWT Component	Function
Label	
Button	
TextField	
Checkbox	
CheckboxGroup	
Choice	
List	
Scrollbar	

- In order to add a control to a container, you need to perform the following two steps.
  1. Create an object of the control by passing the required arguments to the constructor.
  2. Then add it to the window by calling add( ).

6.2.1 Label:

- A label is an object of type Label, and it contains a string, which it displays.
- Labels are passive controls that do not support any interaction with the user.

Constructors:

Label( ) throws HeadlessException	creates a blank label
Label(String str) throws HeadlessException	creates a label that contains the string specified by str. This string is left-justified.
Label(String str, int how) throws HeadlessException	creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three integer constants: • Label.LEFT, • Label.RIGHT, or • Label.CENTER.

Methods:

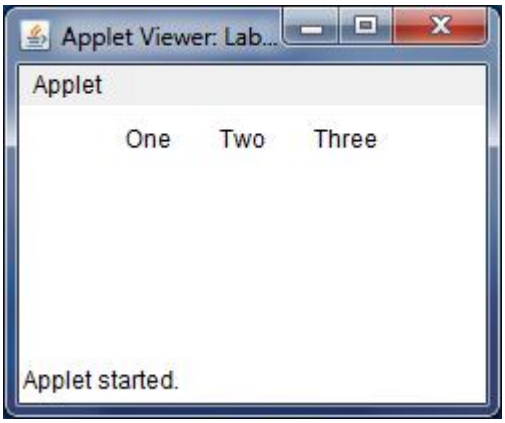
void setText(String str)	to change the label of the button
String getText()	to retrieve the caption.

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code= “LabelDemo.class” Width = 500 Height = 100> </applet>
*/
public class LabelDemo extends Applet
{
        Label l1,l2,l3;
        public void init( )
        {
                l1 = new Label("One",Label.LEFT);
                l2 = new Label("Two",Label.CENTER);
                l3 = new Label("Three", Label.RIGHT);
                add(l1);
                add(l2);
                add(l3);
```

```
        }
    }
```

OUTPUT:



6.2.2 Button:

- A button is a component that contains a label and that generates an event when it is pressed.
- Each time a button is pressed, an action event is generated.
- This is sent to ActionListener which calls actionPerformed(ActionEvent ae) method.
- It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button.
- By default, the action command string is the label of the button.

Constructors:

Button( ) throws HeadlessException	creates an empty button
Button(String str) throws HeadlessException	creates a button that contains str as a label.

Methods:

void setLabel(String str)	creates an empty button
String getLabel()	creates a button that contains str as a label.
String getActionCommand()	

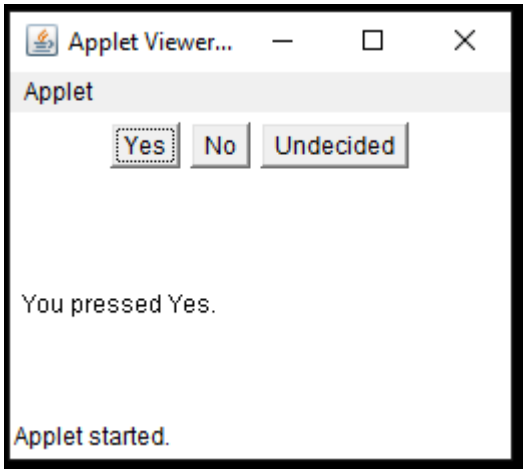
voidsetActionCommand(String str)	

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener
{
        String msg = "";
        Button yes, no, maybe;
        public void init( )
        {
                yes = new Button("Yes");
                no = new Button("No");
                maybe = new Button("Undecided");
                add(yes);
                add(no);
                add(maybe);
                yes.addActionListener(this);
                no.addActionListener(this);
                maybe.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae)
        {
                String str = ae.getActionCommand( );
                if(str.equals("Yes"))
                        msg = "You pressed Yes.";
                else if(str.equals("No"))
                        msg = "You pressed No.";
                else
                        msg = "You pressed Undecided.";
                repaint();
        }
```

```
    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

OUTPUT:



6.2.3 TextField:

- The TextField class implements a single-line text-entry area, usually called an *Edit Control*.
- Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- It is able to generate ActionEvent and TextEvent.

Constructors:

TextField( ) throws HeadlessException	creates a default text field.
TextField(int numChars) throws HeadlessException	creates a text field that is numChars characters wide.
TextField(String str) throws HeadlessException	initializes the text field with the string contained in str.
TextField(String str, int numChars) throws HeadlessException	initializes a text field and sets its width.

Methods:

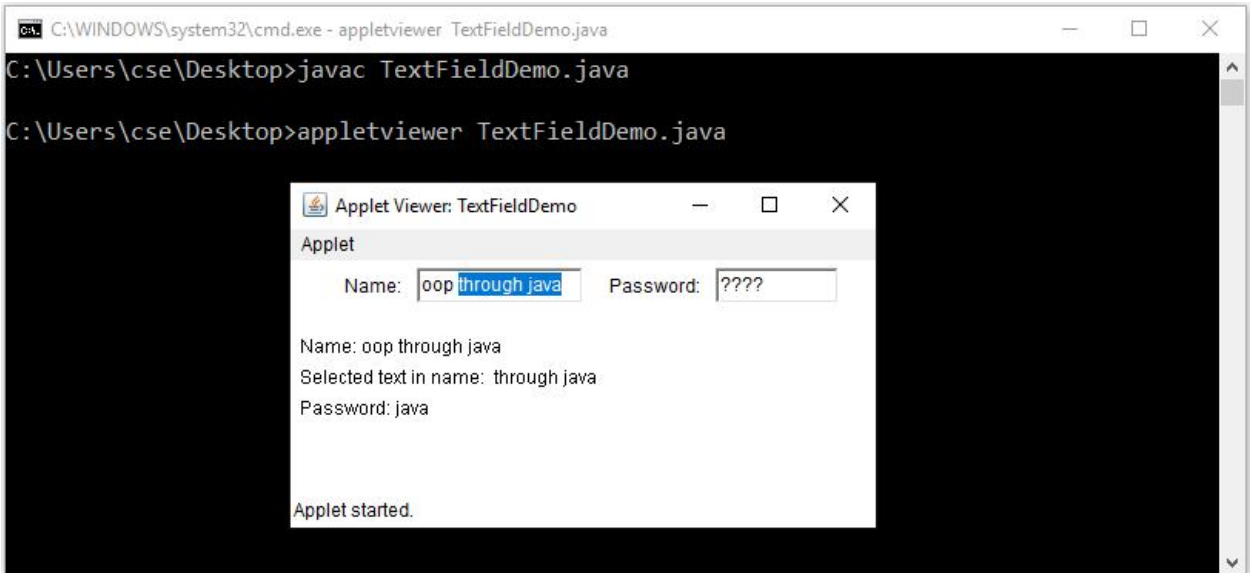
Method	Description
String getText()	Gets the number of columns in this text field.
void setText(String str)	Gets the character that is to be used for echoing.
String getSelectedText()	Sets the number of columns in this text field.
void select(int startIndex, int endIndex)	Sets the echo character for this text field.
booleanisEditable()	Sets the text that is presented by this text component to be the specified text.
void setEditable(Boolean canEdit)	
void setEchoChar(char ch)	
booleanechoCharIsSet()	
char getEchoChar()	

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet implements ActionListener {
      TextField name, pass;
      public void init() {
        Label namep = new Label("Name: ", Label.RIGHT);
        Label passp = new Label("Password: ", Label.RIGHT);
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');
        add(namep);
        add(name);
        add(passp);
```

```
        add(pass);
        // register to receive action events
        name.addActionListener(this);
        pass.addActionListener(this);
      }
      // User pressed Enter.
      public void actionPerformed(ActionEvent ae) {
        repaint();
      }
      public void paint(Graphics g) {
        g.drawString("Name: " + name.getText(), 6, 60);
        g.drawString("Selected text in name: " + name.getSelectedText(), 6,80);
        g.drawString("Password: " + pass.getText(), 6, 100);
      }
}
```

OUTPUT:



6.2.4 Checkbox:

- A check box is a control that is used to turn an option on or off.
- It consists of a small box that can either contain a check mark or not.
- There is a label associated with each check box that describes what option the box represents.
- The state of a check box can be changed by clicking on it.
- Each time a check box is selected or deselected, an item event is generated
- The ItemListener interface defines the itemStateChanged(ItemEvent ie) method which contains information about the event.

Constructors

Checkbox( ) throws HeadlessException	creates an unchecked check box whose label is initially blank
Checkbox(String str) throws HeadlessException	creates an unchecked check box whose label is specified by str.
Checkbox(String str, boolean on) throws HeadlessException	It allows you to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared

Methods

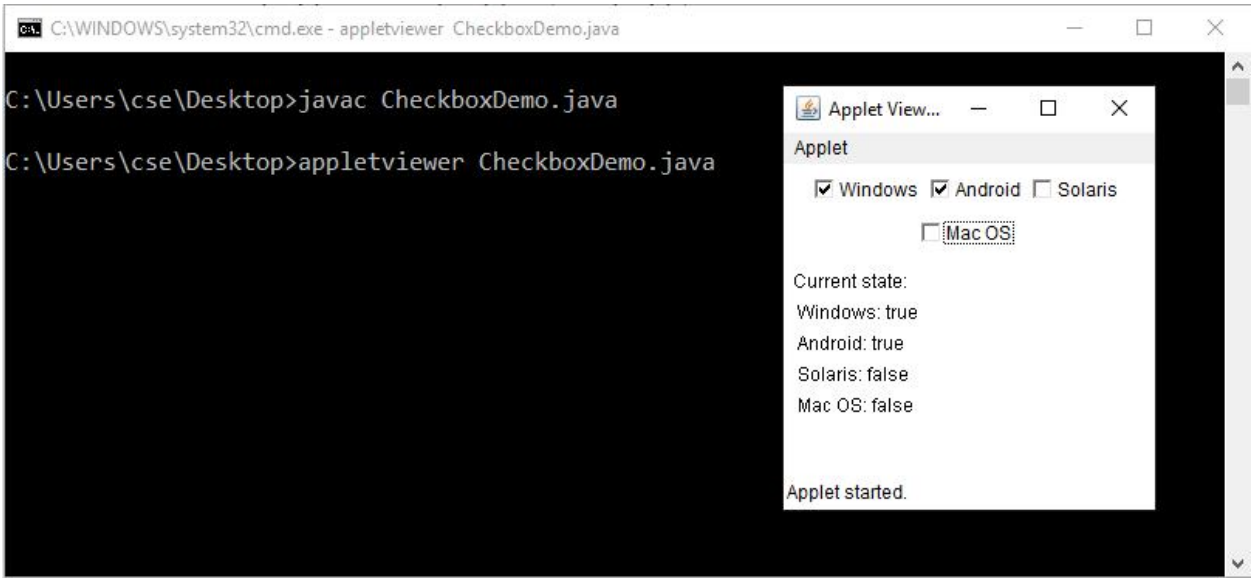
Boolean getState()	
void setState(boolean on)	
String getLabel( )	
void setLabel(String str)	

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=240 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener
{
 String msg = "";
 Checkbox windows, android, solaris, mac;
 public void init( )
 {
  windows = new Checkbox("Windows", null, true);
  android = new Checkbox("Android");
  solaris = new Checkbox("Solaris");
  mac = new Checkbox("Mac OS");
  add(windows);
  add(android);
```

```
 add(solaris);
 add(mac);
 windows.addItemListener(this);
 android.addItemListener(this);
 solaris.addItemListener(this);
 mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
 repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g)
{
 msg = "Current state: ";
 g.drawString(msg, 6, 80);
 msg = " Windows: " + windows.getState();
 g.drawString(msg, 6, 100);
 msg = " Android: " + android.getState();
 g.drawString(msg, 6, 120);
 msg = " Solaris: " + solaris.getState();
 g.drawString(msg, 6, 140);
 msg = " Mac OS: " + mac.getState();
 g.drawString(msg, 6, 160);
}
}
```

OUTPUT:



6.2.5 CheckboxGroup:

- It is a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These are called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time.
- Check box groups are objects of type CheckboxGroup.
- Only the **default constructor** is defined, which creates an empty group.

Methods:

CheckboxGroup()	Creates a new instance of CheckboxGroup.
Checkbox getSelectedCheckbox()	Gets the current choice from this check box group.
Void setSelectedCheckbox(Checkbox box)	Sets the currently selected check box in this group to be the specified check box.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
```

```
*/
public class CBGroup extends Applet implements ItemListener
{
        String msg = "";
        Checkbox winXP, winVista, solaris, mac;
        CheckboxGroup cbg;
        public void init( )
         {
                cbg = new CheckboxGroup( );
                winXP = new Checkbox("Windows XP", cbg, true);
                winVista = new Checkbox("Windows Vista", cbg, false);
                solaris = new Checkbox("Solaris", cbg, false);
                mac = new Checkbox("Mac OS", cbg, false);
                add(winXP);
                add(winVista);
                add(solaris);
                add(mac);
                winXP.addItemListener(this);
                winVista.addItemListener(this);
                solaris.addItemListener(this);
                mac.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {
                repaint( );
        }
        public void paint(Graphics g)
        {
                msg = "Current selection: ";
                msg += cbg.getSelectedCheckbox( ).getLabel( );
                g.drawString(msg, 6, 100);
        }
}
```

OUTPUT:

6.2.6 Choice:

- Choice control is a form of menu
- The Choice class is used to create a pop-up list of items from which the user may choose.
- When inactive, a Choice component takes up only enough space to show the currently selected item.
- When the user clicks on it, the whole list of choices pops up, and a new selection can be made.
- Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object.
- Only one item from the list can be selected at one time and the currently selected element is displayed.
- Choice defines only the default constructor, which creates an empty list.
- Each time a choice is selected, an item event is generated.
- This is sent to ItemListener interface which defines the itemStateChanged(ItemEvent ie) method

Methods:

Choice( )	Creates an instance of Choice
add(String item)	Adds an item to this choice menu
getItem(int index)	Gets the string at the specified index in this choice menu

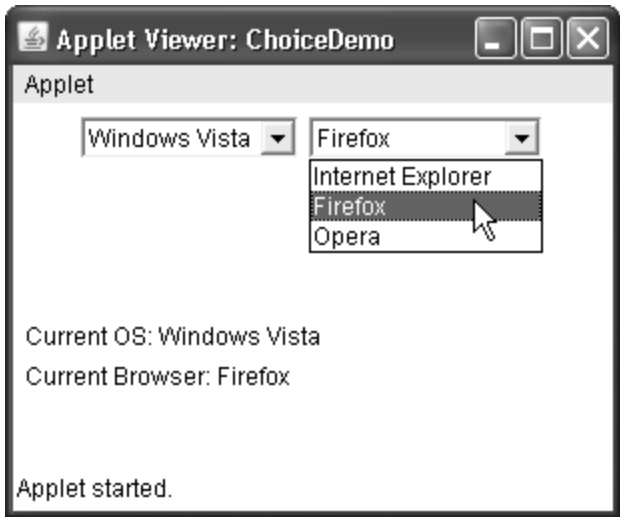
getItemCount()	Gets the number of items in this choice menu
getSelectedIndex()	Gets the index of the currently selected item
getSelectedItem()	Gets the string representation of the currently selected item
insert(String item,int index)	Inserts an item into this choice menu at the specified position
remove(int position)	Removes an item from this choice menu (overloaded)
select(int position) void select(String name)	Selects the specified item within this choice menu (overloaded)

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener
 {
    Choice os, browser;
    String msg = "";
    public void init( )
    {
          os = new Choice();
        browser = new Choice();
        os.add("Windows XP");
        os.add("Windows Vista");
        os.add("Solaris");
        os.add("Mac OS");
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Opera");

        add(os);
        add(browser);

       os.addItemListener(this);
       browser.addItemListener(this);
     }
```

```
    public void itemStateChanged(ItemEvent ie)
    {
          repaint( );
    }
    public void paint(Graphics g)
    {
        msg = "Current OS: ";
        msg += os.getSelectedItem( );
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg+=browser.getSelectedItem( );
        g.drawString(msg, 6, 140);
     }
}
```

OUTPUT:



6.2.7 List:

- The List class provides a compact, multiple-choice, scrolling selection list.
- The list can be configured to that user can choose either one item or multiple items.
- Each time a List item is double-clicked, an ActionEvent object is generated. Its getActionCommand( ) method can be used to retrieve the name of the newly selected item.
- Also, each time an item is selected or deselected with a single click, an ItemEvent object is generated. Its getStateChange( ) method can be used to determine whether a selection or deselection triggered this event.

Constructors :

List( ) throws HeadlessException	creates a List control that allows only one item to be selected at any one time.
List(int numRows) throws HeadlessException	the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed).
List(int numRows, boolean multipleSelect) throws HeadlessException	If multipleSelect is true, then the user may select two or more items at a time.

Methods:

void add(String item)	Adds a new item to the list.
void add(String item, int index)	Adds a new item at the specified position.
void addItem(Sring item)	Adds a new item to the list.
void addItem(String item, int index)	Adds a new item at the specified position.
String getItem(int index)	Gets the item at the specified index.
int getItemCount( )	Gets the number of elements in the list
String [] getItems( )	Gets an array of the element names.
int getRows( )	Gets the number of lines that are currently visible.
int getSelectedIndex( )	Gets the index of the selected item.
int [] getSelectedIndexes( )	Gets a list of items that are all selected.
String getSelectedItem( )	Gets the string that represents the text of the selected item.
String [] getSelectedItems( )	Gets a list of the strings that represent the selected items.
Object [] getSelectedObjects( )	Gets a list of selected strings as Objects.
int getVisibleIndex( )	Gets the index of the item that was last made visible by the makeVisible() method.
boolean isIndexSelected (int index)	Indicates if the specified index represents a selected element.

boolean isMultipleMode( )	Indicates if multiple elements can be simultaneously selected.
void makeVisible(int index)	Makes the item at the specified index visible.
void remove(int position)	Removes the item at the specified position.
void remove(String item)	Removes the first occurrence of the item that matches the string.
void removeAll( )	Removes all items from this list.
void replaceItem(String newValue, int index)	Replaces the item at the specified position with the new item.
void select(int index)	Selects the item at the specified position.
void setMultipleMode (boolean b)	Makes this list use a multiple selection policy.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener
{
   List os, browser;
   String msg = "";
   public void init( )
    {
        os = new List(4, true);
        browser = new List(4, false);
        os.add("Windows XP");
        os.add("Windows Vista");
       os.add("Solaris");
       os.add("Mac OS");
       browser.add("Internet Explorer");
       browser.add("Firefox");
       browser.add("Opera");
       browser.select(1);
       add(os);
       add(browser);
       os.addActionListener(this);
       browser.addActionListener(this);
   }
```

```
    public void actionPerformed(ActionEvent ae)
    {
        repaint();
    }
    // Display current selections.
    public void paint(Graphics g)
    {
        int idx[];
        msg = "Current OS: ";
        idx = os.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
        msg += os.getItem(idx[i]) + " ";
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
```

OUTPUT:

6.2.8 Scroll Bars:

- Scroll bars are used to select continuous values between a specified minimum and maximum.
- Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.
- The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar.
- The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value.

- In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down.
- Scroll bars are encapsulated by the Scrollbar class.
- Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated.
- The types of adjustment events are as follows:

Adjustment Event	Description
BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

Constructors and Methods:

Scrollbar( )	Constructs a new vertical scroll bar.
Scrollbar(int)	Constructs a new scroll bar with the specified orientation.
Scrollbar(int, int, int, int, int)	Constructs a new scroll bar with the specified orientation, initial value, page size, and minimum and maximum values.
int getMaximum( )	Gets the maximum value of this scroll bar.
int getMinimum( )	Gets the minimum value of this scroll bar.
int getValue( )	Gets the current value of this scroll bar.
void setMinimum(int newMaximum)	Sets the minimum value of this scroll bar.
void setMaximum(int newMaximum)	Sets the maximum value of this scroll bar.
void setValue(int newValue)	Sets the value of this scroll bar to the specified value.
void setValues(int value,int visible, int minimum, int maximum)	Sets the values of four properties for this scrollbar: value, visibleAmount, minimum,

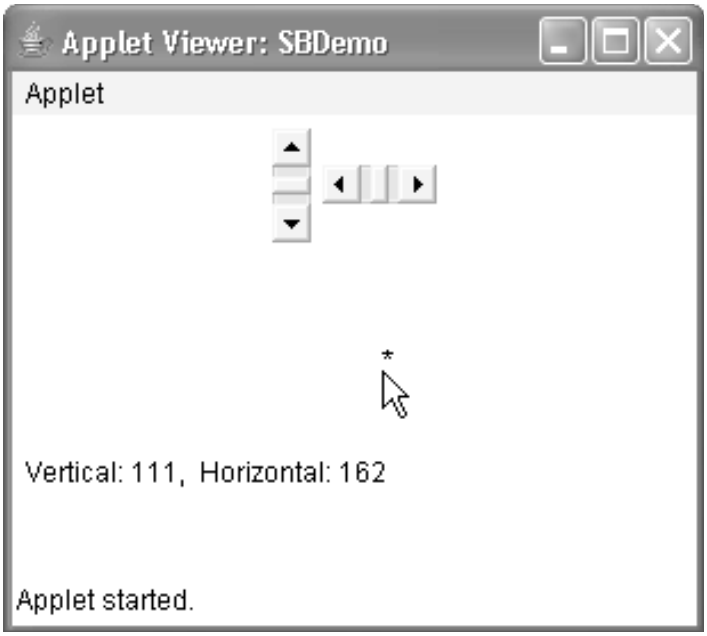
	and maximum.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet implements AdjustmentListener,
MouseMotionListener {
        String msg = "";
        Scrollbar vertSB, horzSB;
        public void init( )
        {
                int width = Integer.parseInt(getParameter("width"));
                int height = Integer.parseInt(getParameter("height"));
                vertSB = new Scrollbar(Scrollbar.VERTICAL,0, 1, 0, height);
                horzSB = new Scrollbar(Scrollbar.HORIZONTAL,0, 1, 0, width);
                add(vertSB);
                add(horzSB);
                vertSB.addAdjustmentListener(this);
                horzSB.addAdjustmentListener(this);
                addMouseMotionListener(this);
        }
        public void adjustmentValueChanged(AdjustmentEvent ae)
        {
                repaint( );
        }
        // Update scroll bars to reflect mouse dragging.
        public void mouseDragged(MouseEvent me)
        {
                int x = me.getX();
                int y = me.getY();
                vertSB.setValue(y);
                horzSB.setValue(x);
                repaint( );
        }
        // Necessary for MouseMotionListener
        public void mouseMoved(MouseEvent me) {
        }
        // Display current value of scroll bars.
        public void paint(Graphics g)
```

```
        {
            msg = "Vertical: " + vertSB.getValue( );
            msg += ", Horizontal: " + horzSB.getValue( );
            g.drawString(msg, 6, 160);
            g.drawString("*", horzSB.getValue( ), vertSB.getValue( ));
        }
}
```

OUTPUT:



6.3 LAYOUT MANAGER:

- 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:

  void setLayout(LayoutManager *layoutObj*)
- Here, *layoutObj* is a reference to the desired layout manager.
- To disable the layout manager and position components manually, pass **null** for *layoutObj.*

- Each layout manager keeps track of a list of components that are stored by their names.
- The layout manager is notified each time you add a component to a container.
- Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize( )** and **preferredLayoutSize( )** methods.
- Each component that is being managed by a layout manager contains the **getPreferredSize( )** and **getMinimumSize( )** methods.
- Layouts are following types:
    - Flow Layout.
    - Border Layout
    - Grid Layout
    - Card Layout
    - Grid Bag Layout

6.3.1 FlowLayout:

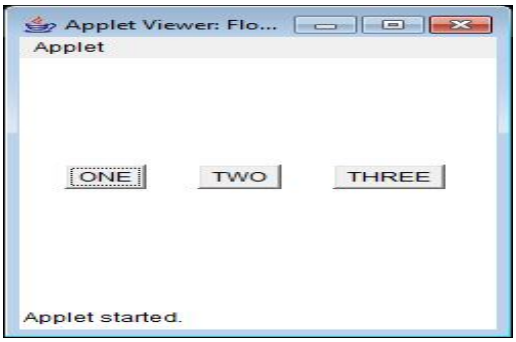
- **FlowLayout** is the default layout manager.
- It implements a simple layout style, which is similar to how words flow in a text editor.
- Components are laid out from the upper-left corner, left to right and top to bottom.
- When no more components fit on a line, the next one appears on the next line.
- A small space is left between each component, above and below, as well as left and right.
- The constructors for **FlowLayout:**
    - FlowLayout( )
    - FlowLayout(int *how*)
    - FlowLayout(int *how*, int *horz*, int *vert*)
- The first form creates the default layout, which centers components and leaves fivepixels of space between each component.
- The second form lets you specify how each line is aligned.
- Valid values for how are as follows:
    - FlowLayout.LEFT
    - FlowLayout.CENTER
    - FlowLayout.RIGHT
- These values specify left, center, and right alignment respectively.

- The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert* respectively.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code="FlowDemo" width=233 height=232 >
</applet>*/
public class FlowDemo extends Applet
{
        Button b1,b2,b3;
        public void init( )
        {
                b1=new Button("ONE");
                b2=new Button("TWO");
                b3=new Button("THREE");
                setLayout(new FlowLayout(FlowLayout.RIGHT,25,100));
                add(b1);
                add(b2);
                add(b3);
        }
}
```

Output:



6.3.2 Border LayOut:

- 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, south, east, and west. The middle area is calledthe center.
- Here are the constructors defined by **BorderLayout:**
  - BorderLayout( )
  - BorderLayout(int *horz*, int *vert*)
- The first form creates a **default** border layout.
- The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively.
- **BorderLayout** defines the following constants that specify the regions:
  - BorderLayout.CENTER
  - BorderLayout.SOUTH
  - BorderLayout.EAST
  - BorderLayout.WEST
  - BorderLayout.NORTH
- When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container:**

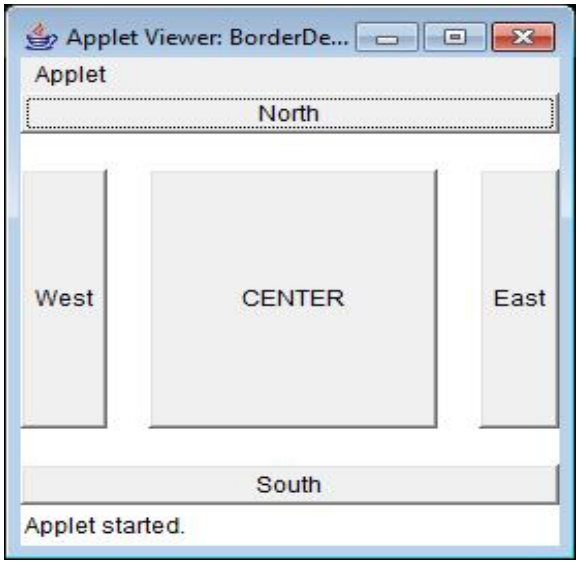
  void add(Component *compObj*, Object *region*);
- Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

Example:

```
import java.awt.*;
import java.applet.*;
/*<applet code="BorderDemo" width=233 height=232 ></applet>*/
public class BorderDemo extends Applet
{
    Button b1,b2,b3,b4,b5;
    public void init( )
    {
        b1=new Button("North");
        b2=new Button("South");
        b3=new Button("East");
        b4=new Button("West");
        b5=new Button("CENTER");
        setLayout(new BorderLayout(20, 20) );
        add(b1, BorderLayout.NORTH);
        add(b2, BorderLayout.SOUTH);
        add(b3, BorderLayout.EAST);
        add(b4, BorderLayout.WEST);
        add(b5, BorderLayout.CENTER);
```

```
        }
    }
```

Output:



6.3.3 GridLayout:

- GridLayoutlays out components in a two-dimensional grid.
- When you instantiate a GridLayout, you define the number of rows and columns.
- The constructors supported by GridLayout are shown here:
  - GridLayout( )
  - GridLayout(int numRows, int numColumns)
  - GridLayout(int numRows, int numColumns, int horz, int vert)
- The first form creates a single-column grid layout.
- The second form creates a grid layout with the specified number of rows and columns.
- The third form allows you 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. Specifying numColumns as zero allows for unlimited-length rows.
- Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons

Example:

```
import java.awt.*;
import java.applet.*;
/*<applet code="GridDemo" width=233 height=232 ></applet>*/
public class GridDemo extends Applet {
        public void init( ) {
                int k=1;
                setLayout(new GridLayout(4, 4) );
                for(int i=0;i<4;i++) {
                        for(int j=0;j<4;j++) {
                                add(new Button(""+k) );
                                k++;
                        }
                }
        }
}
```

Output:

6.3.4 CardLayout

- The **CardLayout** class is unique among the other layout managers in that it stores several different layouts.
- Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.
- This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.

- You can prepare the other layouts and have them hidden, ready to be activated when needed.
- CardLayout provides these two constructors:
  - CardLayout( )
  - CardLayout(int horz, int vert)
- The first form creates a default card layout.
- The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.
- When card panels are added to a panel, they are usually given a name.
- Thus, most of the time, you will use this form of add( ) when adding cards to a panel:

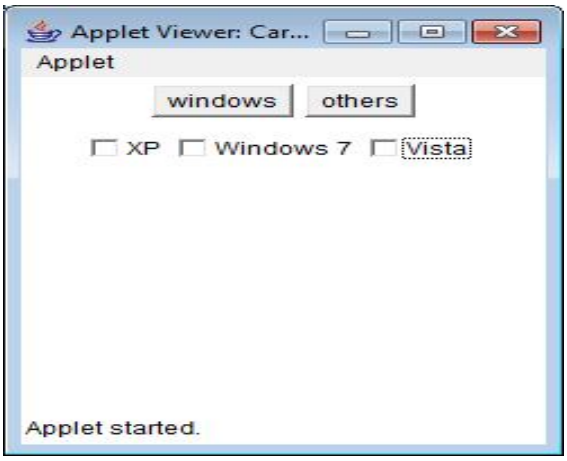
  void add(Component panelObj, Object name);
- Here, name is a string that specifies the name of the card whose panel is specified by panelObj.
- After you have created a deck, your program activates a card by calling one of thefollowing methods defined by CardLayout:
  - void first(Container deck)
  - void last(Container deck)
  - void next(Container deck)
  - void previous(Container deck)
  - void show(Container deck, String cardName)
- Here, deck is a reference to the container (usually a panel) that holds the cards, andcardNameis the name of a card.
- Calling first( ) causes the first card in the deck to be shown.
- To show the last card, call last( ).
- To show the next card, call next( ).
- To show the previous card, call previous( ).
- Both next( ) and previous( ) automatically cycle back to the top or bottom of the deck, respectively.
- The show( ) method displays the card whose name is passed in cardName.

Example:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code="CardDemo" width=233 height=232 ></applet>*/
public class CardDemo extends Applet implements ActionListener {
        Button b1, b2;
        CardLayout cl;
```

```
Checkbox c1, c2, c3, c4, c5;
Panel p;
public void init( ) {
        b1=new Button("windows");
        b2=new Button("others");
        add(b1);
        add(b2);
        cl=new CardLayout( );
        p=new Panel( );
        p.setLayout(cl);
        Panel p1=new Panel( );
        c1=new Checkbox("XP");
        c2=new Checkbox("Windows 7");
        c3=new Checkbox("Vista");
        p1.add(c1);
        p1.add(c2);
        p1.add(c3);
        Panel p2=new Panel( );
        c4=new Checkbox("mac Os");
        c5=new Checkbox("Linux");
        p2.add(c4);
        p2.add(c5);
        p.add(p1,"wcard");
        p.add(p2,"ocard");
        add(p);
        b1.addActionListener(this);
        b2.addActionListener(this);
}
public void actionPerformed(ActionEvent ae) {
        String s=ae.getActionCommand( );
        if(s.equals("windows") ) {
                cl.first(p);
        }
        else if(s.equals("others") )
                cl.show(p,"ocard");
}
}
```

Output:



6.3.5 GridBagLayout

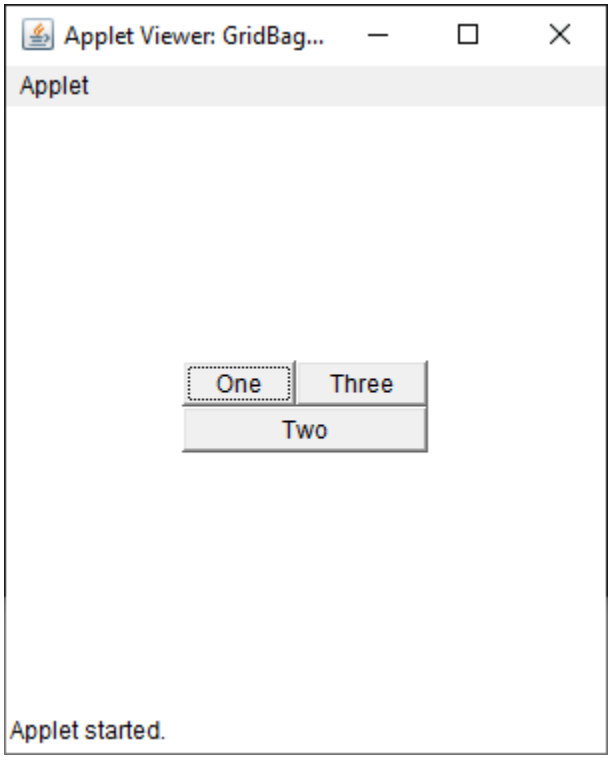
- GridBagLayout is a layout manager that lays out a container's components in a grid of cells with each component occupying one or more cells, called its **display area**.
- The display area aligns components vertically and horizontally, without requiring that the components be of the same size.
- GridBagLayout constructor:
  - GridBagLayout( )

Example :

```
public class GridBagDemo extends Applet {
        Button b1,b2,b3;
        public void init( ) {
                b1=new Button("One");
                b2=new Button("Two");
                b3=new Button("Three");
                GridBagLayout gb= new GridBagLayout( );
                setLayout(gb);
                GridBagConstraints gbc = new GridBagConstraints( );
                gbc.ipadx=20;
                gbc.gridx = 0;
                gbc.gridy = 0;
                l.setConstraints(b1, gbc);
                gbc.gridx = 1;
                gbc.gridy = 0;
                l.setConstraints(b3, gbc);
                gbc.fill = GridBagConstraints.HORIZONTAL;
                gbc.gridx = 0;
                gbc.gridy = 1;
```

```
                gbc.gridwidth=2;
                gb.setConstraints(b2, gbc);
                add(b1);
                add(b2);
                add(b3);
            }
        }
```

Output:



UNIT-VI

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. AWT stands for [ ]

(a) Applet Windowing Toolkit

(b) Abstract Windowing Toolkit

(c) Absolute Windowing Toolkit

(d) None of the above

2. Which object can be constructed to show any number of choices in the visible window? [ ]

(a) Labels (b) Choice (c**)** List (d) Checkbox

3. Which class provides many methods for graphics programming?

(a) java.awt (b) java.Graphics

(c**)** java.awt.Graphics (d) None of the above

4. _____Layout arranges the components as a deck of cards such that only one component is visible at a time [ ]

(a) CardLayout (b)Borderlayout (c)FlowLayout (d)GridLayout

5. At the top of the AWT hierarchy is the_____ class. [ ]

(a) Window (b) Component (c) Panel (d)Frame

6. AWT classes are contained in the _____ package [ ]

(a) java.awt (b) java.Awt

(c) java.classes.awt (d) java.pacakge.awt

7. BorderLayout class has __regions to add components to it [ ]

(a) 4 (b)7 (c)5 (d)8

8. By default FlowLayout uses _____justification. [ ]

(a)Left (b)Right (c)Center (d)Top

9. By default page-up and page-down increment of scrollbar is__ [ ]

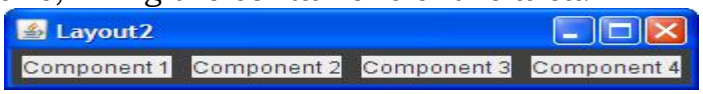
(a)5 (b)10 (c)7 (d)6

In each of the following questions, choose the layout manager(s) most naturally suited for the described layout.

11) The container has one component that should take up as much space as possible [ ]

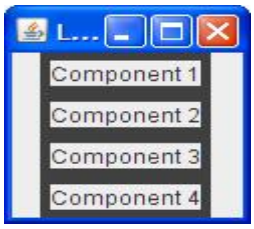
a). BorderLayout b). GridLayout c). GridBagLayout d). a and b e). b and c

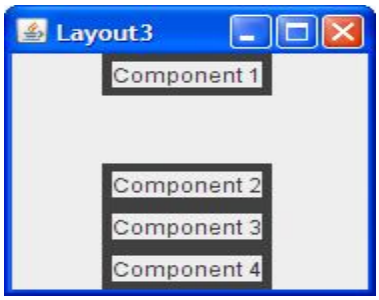
12). The container has a row of components that should all be displayed at the same size, filling the container's entire area. [ ]



a). FlowLayout b). GridLayout c). BoxLayout d). a and b

13) The container displays a number of components in a column, with any extra space going between the first two components. [ ]



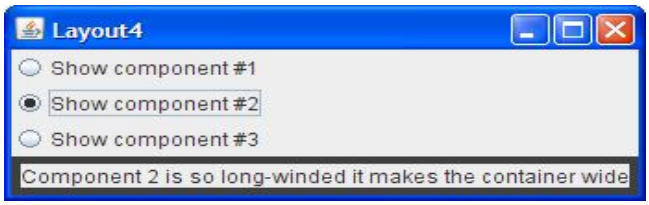


a). FlowLayout b). BoxLayout

c). GridLayout d). BorderLayout

14) The container can display three completely different components at different times, depending perhaps on user input or program state. Even if the components' sizes differ, switching from one component to the next shouldn't change the amount of space devoted to the component.

[ ]



a). SpringLayout b). BoxLayout

c). CardLayout d). GridBagLayout

SECTION-B

Descriptive Questions

1) Explain in detail AWT class hierarchy.
2) Explain the following Components with an example

   A) Label B) Button
3) Differentiate between Checkbox and ChechboxGroup. Explain them with an Example

4) Explain with an example how to add Choice and List Controls to the container.
5) Explain with an example the following Scrollbar user Interface component
6) What are layout managers? Explain their importance and List them.
7) Explain with an example Border Layout layout Manager
8)Write a short note on Flow and Card Layouts. Give examples
9)Write an AWT program to create checkboxes for different courses belonging to a university such that the courses selected would be displayed.
10)Create a list of vegetables. If you click on one of the items of the list, the item should be displayed
11)Write a java program to show how the radio buttons can be used to change the background color of the applet window