INTORDUCTION TO OOP AND JAVA

1.Explain the features and characteristics of java(13)

OOPs simplify the software development and maintenance by providing some concepts:

1. Class - Blue print of Object
2. Object - Instance of class
3. Encapsulation - Protecting our data
4. Polymorphism - Different behaviors at different instances
5. Abstraction - Hiding irrelevant data
6. Inheritance - An object acquiring the property of another object

1. Class:

A class is a collection of similar objects and it contains data and methods that operate on that data. In other words

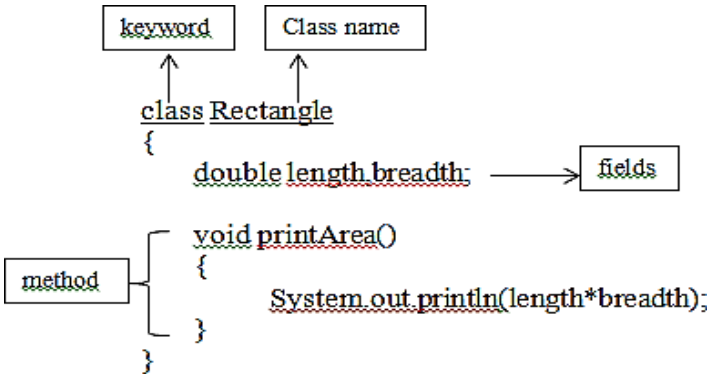
A class in Java can contain:

- **fields**
- **methods**
- **constructors**
- **blocks**
- **nested class and interface**

Syntax to declare a class:

Example:

```
class <class_name>
{
   field;
  method;
}
```



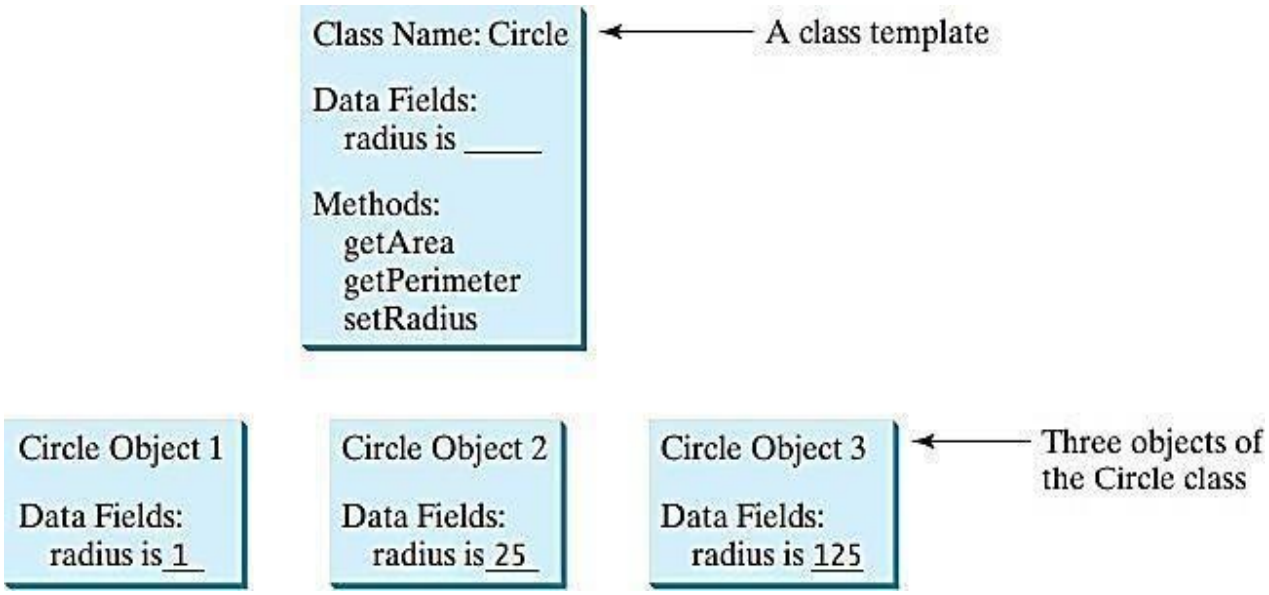
2. Object:

Any entity that has state and behavior is known as an object. **Object is an instance of a class.**

- ✓ For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.
- ✓ The object of a class can be created by using the **new** keyword in Java Programming language.

```
class_name object_name = new class_name;
                (or)
class_name object_name;
object_name = new class_name();
```

Syntax to create Object in Java:



An object has three characteristics:

- **State:** represents data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **Identity:** Object identity is an unique ID used internally by the JVM to identify each object uniquely.
- For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Difference between Object and Class

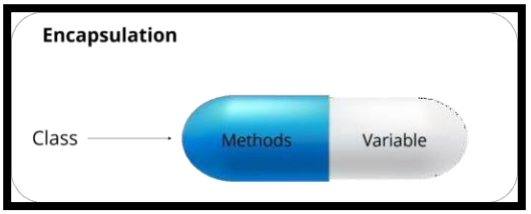
S.No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g.class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.

6)	Object **allocates memory when it iscreated**.	Class **doesn't allocated memory when it iscreated**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

3. Encapsulation:

Wrapping of data and method together into a single unit is known as Encapsulation.

For example: capsule, it is wrapped with different medicines.



- ✓ **The insulation of the data from direct access by the program is called —data hiding.**

4. Polymorphism:

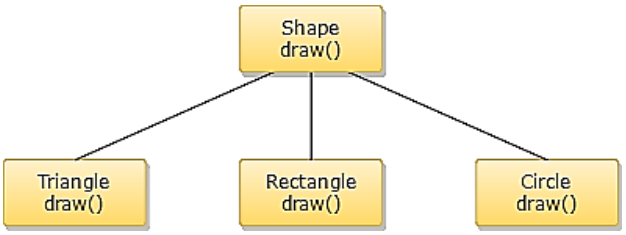
- ✓ Polymorphism is a concept by which we can perform a single action by different ways.
- ✓ It is the ability of an object to take more than one form.
- ✓ The word "poly" means many and "morphs" means forms. So polymorphism meansmany forms.
- ✓ An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation.

- Two types of polymorphism:

1. **Compile time polymorphism / Method Overloading: -** In this method, object is bound to the function call at the compile time itself.
2. **Runtime polymorphism / Method Overriding: -** In this method, object is bound to the function call only at the run time.

- **<u>In java, we use method overloading and method overriding to achievepolymorphism.</u>**
- **Example:**

1. draw(int x, int y, int z)
2. draw(int l, int b)
3. draw(int r)



5. Abstraction:

- ✓ Abstraction refers to the act of representing essential features without including the background details or explanations**.**
- ✓ **i.e., *Abstraction* means hiding lower- level details and exposing only the essential and relevant details to the users.**
- ✓ ForExample:-ConsideranATMMachine;
- ✓ Abstractionprovidesadvantageofcodereuse.
- ✓ Abstraction enables program open for extension.
- ✓ **In java, abstract classes and interfaces are used to achieve Abstraction.**

6. Inheritance:

- ✓ **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of another object.
- ✓ Inheritance represents the **IS-A relationship**, also known **as *parent-child* relationship**.
- ✓ For example:- In a child and parent relationship, all the properties of a father areinherited by his son.
- ✓ **Syntax of Java Inheritance**

```
class Subclass-name extends Superclass-name
{
//methods and fields
}
```

7. Message Passing: Message Communication:

- ✓ Objects interact and communicate with each other by sending ***messages*** to eachother. This informationispassedalongwiththemessageasparameters.

- ✓ Amessageforanobjectisarequestforexecutionofaprocedure andtherefore will invoke a method (procedure) in the receiving object that generates the desired result.
- ✓ Message passing involves specifying the name of the object, the name of themethod (message) and the information to be sent.
- ✓ **Example:**

  Employee.getName(name); Where,
  Employee – object name
  tName – method name (message) name - information

The following are the features of the Java language:

1. ObjectOriented
2. Simple
3. Secure
4. Platform Independent
5. Robust
6. Portable
7. Architecture Neutral
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

1. Object Oriented:

- ✓ Java programming is pure object-oriented programming language.
- ✓ Example: **Printing "Hello" Message.**

C++ (can be without class)	Java – No programs without classes and objects

With Class:
#include<iostream.h>
class display {
public:
void disp()
{
cout<<"Hello!";
}
};
main()
{
display d;
d.disp();
}
 	
Without class: #include<iostream.h>
void main()
{
clrscr();
cout<<"\n Hello!";
getch();
}
	With class:
	import java.io.*;
	class Hello {
	public static void main(String args[])
	{
	System.out.println("Hello!");
	}
	}
	
	Without class is not possible

2. Simple:

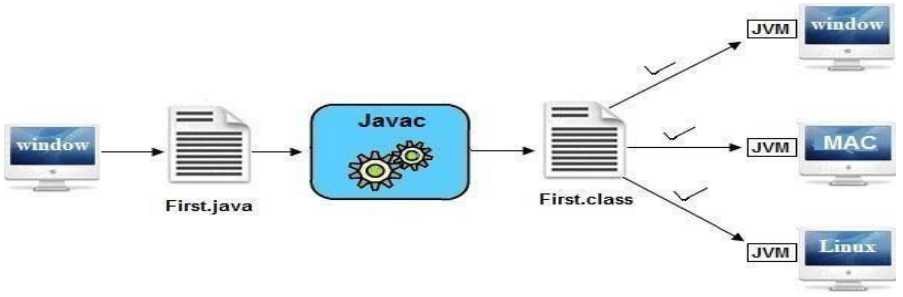
- ✓ Java is Easy to write and more readable and eye catching.
- ✓ Most of the concepts are drew from C++ thus making Java learning simpler.

3. Secure :

- ✓ Java provides a secure means of creating Internet applications and to access web applications.
- ✓ Java enables the construction of secured, virus-free, tamper-free system.

4. Platform Independent:

- ✓ The Java bytecodes are not specific to any processor. They can be executed in any computer without any error.
- ✓ Because of the bytecode, Java is called as **Platform Independent.**



5. Robust:

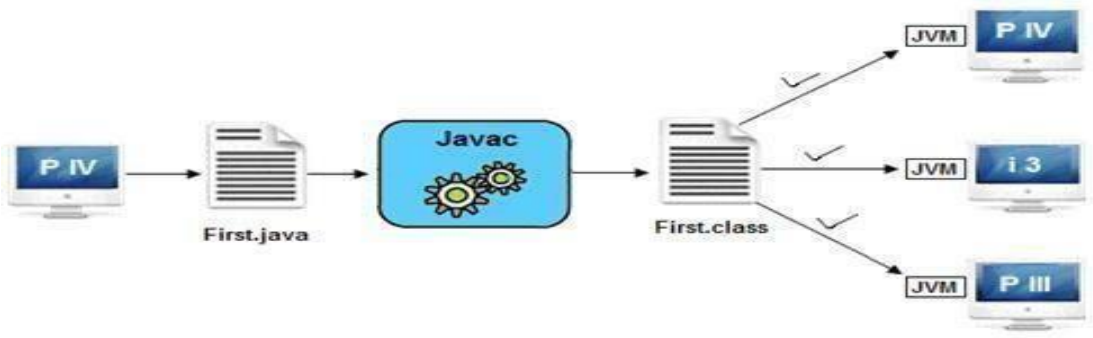
- ✓ Java encourages error-free programming by being strictly typed and performing run- time checks.

6. Portable:

- ✓ Java bytecode can be distributed over the web and interpreted by **Java Virtual Machine (JVM)**
- ✓ Java programs can run on any platform (Linux, Window, Mac)
- ✓ Java programs can be transferred over world wide web (e.g applets)

7. Architecture Neutral:

- ✓ Java is not tied to a specific machine or operating system architecture.
- ✓ Machine Independent i.e Java is independent of hardware.
- ✓ Bytecode instructions are designed to be both easy to interpret on any machine and easily translated into native machine code.



8. Dynamic and Extensible:

- ✓ Java is a more dynamic language than C or C++. It was developed to adapt to an evolving environment.
- ✓ Java programs carry with them substantial amounts of run-time information that are used to verify and resolve accesses to objects at run time.

9. Interpreted:

- ✓ Java supports cross-platform code through the use of Java bytecode.
- ✓ The Java interpreter can execute Java Bytecodes directly on any machine to which the interpreter has been ported.

10. High Performance:

- ✓ Bytecodes are highly optimized.
- ✓ JVM can execute the bytecodes much faster.
- ✓ With the use of Just-In-Time (JIT) compiler, it enables high performance.

11. Multithreaded:

- ✓ Java provides integrated support for multithreaded programming.
- ✓ Using multithreading capability, we can write programs that can do many tasks simultaneously.
- ✓ The benefits of multithreading are better responsiveness and real-time behavior.

12. Distributed:

- ✓ Java is designed for the distributed environment for the Internet because it handles TCP/IP protocols.
- ✓ Java programs can be transmit and run over internet.

2.Explain the structure of java program(13)

Java program may contain many classes of which only one class defines the main method.
A Java program may contain one or more sections.

Documentation Section
Package Statement
Import Statements
Interface Statements
Class Definitions

main Method Calss
{
Main Method Definition
}

Documentation Section

- ✓ It Comprises a Set of comment lines giving the name of the program, the authorand other details.

- ✓ Comments help in Maintaining the Program.
- ✓ Java uses a Style of comment called *documentation comment.*
  /* * …… */
- ✓ This type of comment helps is generating the documentation automatically.
- ✓ **Example:**

```
/*
 * Title: Conversion of Degrees
 * Aim: To convert Celsius to Fahrenheit and vice versa
 *Date: 31/08/2000
 * Author: tim
*/
```

Package Statement

- ✓ The first statement allowed in a Java file is a package statement.
- ✓ It declares the package name and informs the compiler that the classes defined belong to this package.
- ✓ **Example :**

  package student;

  package basepackage.subpackage.class;
- ✓ It is an optional declaration.

Import Statements

- ✓ The statement instructs the interpreter to load a class contained in a particular package.
- ✓ Example :

  import student.test;

  Where, student is the package and test is the class.

Interface Statements

- ✓ An interface is similar to classes which consist of group of method declaration.
- ✓ To link the interface to our program, the keyword **implements** is used.
- ✓ **Example:**

  public class xx extends Applet implements ActionListener

  where, xx – classname (subclass of Applet) Applet – Baseclassname ActionListener – interface Extends & implements - keywords

Class Definitions

- ✓ A Java Program can have any number of class declarations.
- ✓ The number of classes depends on the complexity of the program.

Main Method Class

- ✓ **Syntax for writing main:**

  public static void main(String arg[])

  where,

  public – It is an access specifier to control the visibility of class members. main() must be declared as public, since it must be called by code outside of its class when the program is started.

  static – this keyword allows main() method to be called without having to instantiate the instance of the class.

  void – this keyword tells the compiler that main() does not return any value.

  main() – is the method called when a Java application begins.

  String arg[] – arg is an string array which receives any command-line arguments present when the program is executed.

<u>Rules to be followed to write Java Programs:</u>

About Java programs, it is very important to keep in mind the following points.

- ***Case Sensitivity - Java is case sensitive, which means identifier Hello and hello would have different meaning in Java.***
- ***Class Names - For all class names the first letter should be in Upper Case.***
  Example class MyFirstJavaClass
- ***Method Names - All method names should start with a Lower Case letter.***
  Example public void myMethodName()
- ***Program File Name - Name of the program file should exactly match the class name.***
  Example : Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as

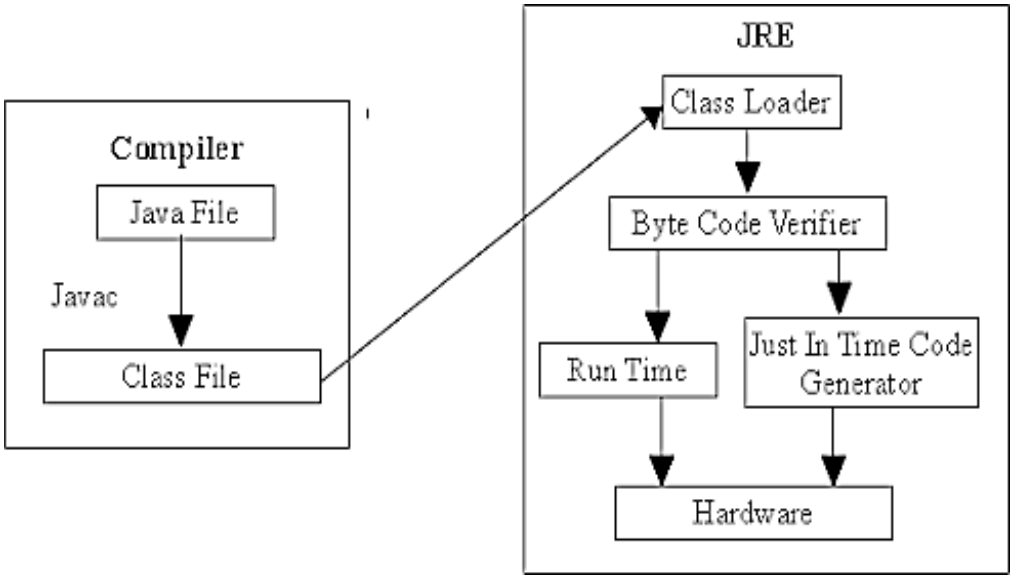
'MyFirstJavaProgram.java'

- ***public static void main(String args[]) - Java program processing starts from the main() method which is a mandatory part of every Java program.***

Compiling and running a java program in command promptSTEPS:

1. Set the path of the compiler as follows (type this in command prompt):
   Set path="C:\Program Files\Java\jdk1.6.0_20\bin";
2. To create a Java program, ensure that the name of the class in the file is the same as the name of the file.
3. Save the file with the extension .java (Example: HelloWorld.java)
4. To compile the java program use the command javac as follows:
   javac HelloWorld.java
   This will take the source code in the file HelloWorld.java and create the java bytecode in a file HelloWorld.class
5. To run the compiled program use the command java as follows:
   java HelloWorld
   (Note that you do not use any file extension in this command.)

   At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



Class Loader : is the subsystem of JVM that is used to load class files.
Bytecode Verifier : checks the code fragments for illegal code that can violate access right to objects
Interpreter : read bytecode stream then execute the instructions.

Example 1: A First Java Program:

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

Save: HelloWorld.java**Compile:** javac HelloWorld.java**Run:** javaHelloWorld
Output: Hello World

3.Explain the data types of java(13)

Data type is used to allocate sufficient memory space for the data. Data types specify the different sizes and values that can be stored in the variable.

- ***Java is a strongly Typed Language.***

Data types in Java are of **two types**:

1. **Primitive data types (Intrinsic or built-in types) :- :** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types (Derived or Reference Types):** The non-primitive data types include Classes, Interfaces, Strings and Arrays.

1. Primitive Types:

Primitive data types are those whose variables allow us to store only one value and never allow storing multiple values of same type.

There are eight primitive types in Java:

Integer Types:

1. int
2. short
3. lomg
4. byte

Floating-point Types:

5. float
6. double

Others:

7. char
8. Boolean

- **Integer Types:**

The integer types are form numbers without fractional parts. Negative values are allowed.Java provides the four integer types shown below:

Type	Storage Requirement	Range	Example	Default Value
int	4 bytes	-2,147,483,648(-2^31) to 2,147,483,647(2^31-1)	inta=100000, intb =-200000	0
short	2 bytes	-32,768(-2^15)to32,767(2^15-1)	short s = 10000, short r = -20000	0
long	8 bytes	-9,223,372,036,854,775,808 (-2^63) to 9,223,372,036,854,775,808 (2^63-1)	long a = 100000L, int b =-200000L	0L
byte	1 byte	-128 (-2^7) to 127 (2^7-1)	byte a=100, byte b=-50	0

- **Floating-point Types:**

The floating-point types denote numbers with fractional parts. The two floating-pointtypes are shown below:

Type	Storage Requirement	Range	Example	Default Value
float	4 bytes	Approximately ±3.40282347E+38F (6-7 significant decimal digits)	floatf1 =234.5f	0.0f
double	8 bytes	Approximately ±1.79769313486231570E+308 (15 significant decimal digits)	double d1 = 123.4	0.0d

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

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

2. Derived Types (Reference Types):

- **Derived data types are those whose variables allow us to store multiple values of same type. But they never allow storing multiple values of different types.**

- **The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable.**
- Example

```
inta[]={10,20,30};                    // valid
intb[]={100,'A',"ABC"};               //invalid
Animal animal = new Animal("giraffe"); //Obj
```

4.Explain the variables of java(13)

- **A Variable is a named piece of memory that is used for storing data in java Program.**
- A variable is an identifier used for storing a data value.
- **Syntax to declare variables:**

  datatype identifier [=value][,identifier [=value] …];

- Example of Variable names:

  int average=0.0, height, total height;

- **Rules followed for variable names (consist of alphabets, digits, underscore anddollar characters)**
  1. A variable name must begin with a letter and must be a sequence of letter or digits.
  2. They must not begin with digits.
  3. Uppercase and lowercase variables are not the same.
     a. **Example:** Total and total are two variables which are distinct.
  4. It should not be a keyword.
  5. Whitespace is not allowed.
  6. Variable names can be of any length.

- **Initializing Variables:**
  - ✓ After the declaration of a variable, it must be initialized by means of assignment statement.
  - ✓ Two ways to initialize a variable: 1.

Initialize after declaration:

Syntax: **variablename=value;**

```
int months;
months=1;
```

2. Declare and initialize on the same line:

Syntax: **Datatype variablename=value;**

```
int months=12;
```

- **dynamic Initialization of a Variable:**

Java allows variables to be initialized dynamically using any valid expression at the timethe variable is declared.

Example: Program that computes the remainder of the division operation:

```
class FindRemainer
{
public static void main(String arg[]) {int num=5,den=2;
int rem=num%den; System.out.println(-Remainder is –+rem);
}
}
```

Output:

Remainder is 1

JAVA - VARIABLE TYPES

There are three kinds of variables in Java:

1. Local variables
2. Instance variables
3. Class/static variables

Local Variables	Instance Variable	Class / Static Variables
Local variables aredeclared in methods, constructors, or blocks.	Instance variables are declared in a class, but outside a method, constructor or any block.	Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block. There would only be one copy of each class variable per class, regardless of how many objects are createdfromit.
Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.	Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when theobject is destroyed.	Static variables are created when the program starts and destroyed when the program stops.
Access modifiers cannot be usedfor local variables.	Access modifiers can be used forinstance variables.	Access modifiers can be used for class variables.

Local variables are visible only within the declared method, constructor or block.	The instance variables are visible for all methods, constructors and block in the class.	Visibility is similar to instance variables.
There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.	Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.	Default values are same as instance variables.
Local variables can only be access inside the declared block.	Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class should be called using the fully qualified name as follows: **ObjectReference.VariableName.**	Static variables can be accessed by calling with the class name. **ClassName.VariableName.**

Example program illustrating the use of all the above variables:

```
class area
{
int length=20; int
breadth=30;
staticicintclassvar=2500; void calc()
{
        int areas=length*breadth;
        System.out.println("The area is "+areas+" sq.cms");
}

public static void main(String args[])
{
  area a=new area();
  a.calc();
  System.out.println("Static Variable Value : "+classvar);
}
}
```

Output:

The area is 600 sq.cms Static Variable Value : 2500

5.explain single dimensional array with example(13)

Definition:

An array is a collection of similar type of elements which has contiguous memory location.

Advantage of Array:

- **Code Optimization:** It makes the code optimized; we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Array:

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime.

Types of Array:

There are two types of array.

1. One-Dimensional Arrays
2. Multidimensional Arrays

1. One-Dimensional Array:

Definition: One-dimensional array is an array in which the elements are stored in one variable name by using only one subscript.

- **Creating an array:**

Three steps to create an array:

1. Declaration of the array
2. Instantiation of the array
3. Initialization of arrays

1. Declaration of the array:

Declaration of array means the specification of array variable, data_type and array_name.

Syntax to Declare an Array in java:

```
dataType[] arrayRefVar;   (or)
dataType []arrayRefVar;   (or)
dataType arrayRefVar[];
```

Example:

int[] floppy; (or) int []floppy (or) int floppy[];

2. Instantiation of the array:

Syntax:

arrayRefVar=new datatype[size];

Example: floppy=newint[10];

3. Initialization of arrays:Definition:

Storing the values in the arrayelement is called as **Initialization of arrays.**

Syntax to initialize values to array element:

arrayRefVar[indexvalue]=constantorvalue;

Example:

floppy[0]=20;

SHORTHAND TO CREATE AN ARRAY OBJECT:

Javahasshorthandtocreateanarrayobjectandsupply initial values at the same timewhen it is created.

dataType[] arrayRefVar={list of values};
(or) dataType
[]arrayRefVar={list of values};
(or) dataType
arrayRefVar[]={list of values};
(or) dataType
arrayRefVar[]=arrayVariable;

Example 1:

int regno[]={101,102,103,104,105,106};
int reg[]=regno;

Example 2: double[] myList = new double[10];

ARRAY LENGTH:

The variable **length** can identify the length of array in Java. To find the number of elements of an array, use **array.length.**

Example1:

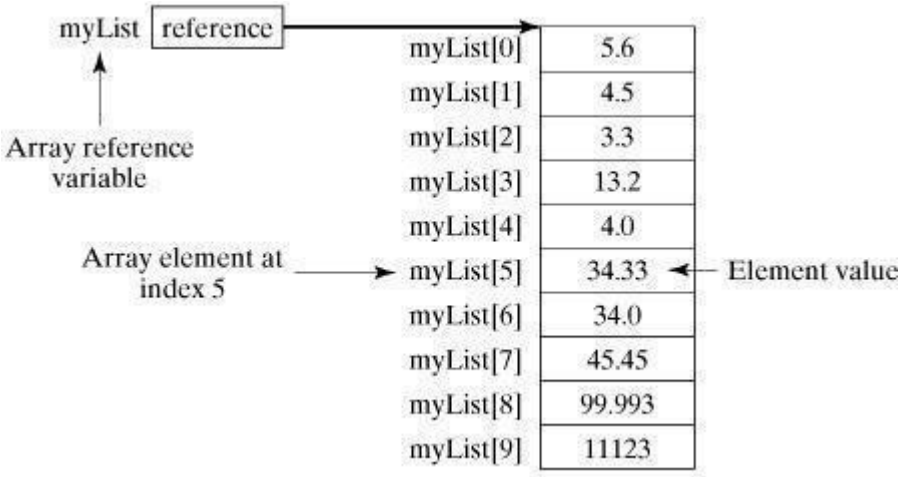
int regno[10]; len1=regno.length;

Example 2:

for(int i=0;i<reno.length;i++) System.out.println(regno[i]);

Following picture represents array myList. Here, myList holds ten double values and the indicesare from 0 to 9.

Example: (One-DimensionalArray)




```
class Array
{
public static void main(String[] args)
{
        int month_days[]; month_days=new
        int[12]; month_days[0]=31;
        month_days[1]=28;
        month_days[2]=31;
        month_days[3]=30;
        month_days[4]=31;
        month_days[5]=30;
        month_days[6]=31;
        month_days[7]=31;
        month_days[8]=30;
        month_days[9]=31;
        month_days[10]=30;
        month_days[11]=31;

        System.out.println("April has "+month_days[3]+ " days.");
}
}
```

Output:

April has 30 days.

Total is11.7
Max is 3.5

6.Explain multidimensional array with example(13)

Uses of Multidimensional Arrays:

- ✓ Used fortable
- ✓ Used for more complex arrangements

Syntax to Declare Multidimensional Array in java:

```
1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in java:

```
int[][]arr=newint[3][3];
```

Example to initialize Multidimensional Array in java:

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;

arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Examples to declare, instantiate, initialize and print the 2Dimensional array:

```
class twoDarray
{
public static void main(String args[])
{
int array1[][]=new int[4][5];// declares an 2D array.
```

```
intarray2[][]={{1,2,3},{2,4,5},{4,4,5}};//declaring and initializing 2D arrayint i,j,k=0;

  // Storing and printing the values of Array1

System.out.println("-------Array 1 ----------" );
for(i=0;i<4;i++)
{
      for(j=0;j<5;j++)
      {
            array1[i][j]=k;k++;
            System.out.print(array1[i][j]+ " ");
      }
      System.out.println();
}

// printing 2D array2
System.out.println("-------Array 2 ----------");
for(int i=0;i<3;i++)
      {
      for(int j=0;j<3;j++)
      {
      System.out.print(array2[i][j]+
      }
      System.out.println();
      }
      }
}
```

Output:

----------Array1----------

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

----------Array2----------

1 2 3
2 4 5
4 4 5

7.Write a program for Unary Operator (7)

```
class OperatorExample
{
public static void main(String args[])
 {
 int x=10;
 System.out.println(x++);
System.out.println(++x);
System.out.println(x--);
 System.out.println(--x);
}
}
```

Output:

```
12
12
10
```

Java Unary Operator Example: ~ and !

```
1.  class OperatorExample{
2.  public static void main(Stringargs[]){
3.      int a=10;
4.      int b=-10;
5.  boolean c=true;
6.  boolean d=false;
7.  System.out.println(~a);     //-11 (minus of total positive value which starts from 0)
8.  System.out.println(~b);     //9 (positive of total minus, positive starts from 0)
9.  System.out.println(!c);     //false (opposite of boolean value)
10.     System.out.println(!d);     //true
11.     }
12.     }
```

Output:

```
-11
```

9
False
true

8.Explain the operators of java with example(13)

1. **Assignment**
2. **Arithmetic**
3. **Relational**
4. **Logical**
5. **Bitwise**
6. **Compound assignment**
7. **Conditional**
8. **Type**.

Assignment Operators	**=**
Arithmetic Operators	**- + * / % ++ --**
Relational Operators	**> < >= <= == !=**
Logical Operators	**& || &	! ^ &**
Bit wise Operator	**&	^ >> >>>**
Compound Assignment Operators	**+= -= *= /= %= <<= >>= >>>=**
Conditional Operator	**?:**

1. Java Assignment Operator

The java assignment operator statement has the following syntax:

<variable> = <expression>

If the value already exists in the variable it is overwritten by the assignment operator (=).

Java Assignment Operator Example

```
1. class OperatorExample{
2. public static void main(Stringargs[])
3. {
4. int a=10;
5. int b=20;
6. a+=4;                    //a=a+4 (a=10+4)
7. b-=4;                    //b=b-4 (b=20-4)
8. System.out.println(a);
9. System.out.println(b);
10. }
11. }
```

Output:

```
14
16
```

2. Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

Java Arithmetic Operator Example: Expression

```
1.  class OperatorExample 2.
    {
3.  public static void main(String args[]) 4.
    {
5.      System.out.println(10*10/5+3-1*4/2);4.
6.  }
7.  }
```

Output:

```
21
```

3. Relational Operators

Relational operators in Java are used to compare 2 or more objects. Java provides six relational operators: Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example:

```
public RelationalOperatorsDemo()
{
      int x = 10, y = 5; System.out.println("x>y:
      "+(x>y));
      System.out.println("x<y:"+(x<y));
      System.out.println("x>=y:"+(x>=y));
      System.out.println("x<=y:"+(x<=y));
      System.out.println("x==y:"+(x==y));
      System.out.println("x!=y:"+(x!=y));

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

Output:

```
$java RelationalOperatorsDemo
```

x>y:truex
<y:falsex
>= y : true
x <= y : false x
==y:false x!=
y:true

4. **Logical Operators**

Logical operators return a true or false value based on the state of the Variables. Given that x and y represent boolean expressions, the boolean logical operators are defined in the Tablebelow.

x	y	!x	x & y x && y	x | y x || y	x ^ y
true	true	false	true	true	False
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Example:

```
public class LogicalOperatorsDemo
{
    public LogicalOperatorsDemo()
    {
        boolean x = true;
        boolean y = false;
        System.out.println("x&y:"+(x&y));
        System.out.println("x && y : " + (x && y));
        System.out.println("x|y:"+(x|y));
        System.out.println("x||y:"+(x||y));
        System.out.println("x^y:"+(x^y));
        System.out.println("!x : " + (!x));
    }
public static void main(String args[])
{
new LogicalOperatorsDemo();
}
}
```

Output:

$java LogicalOperatorsDemo

x&y:false x
&&y:false x|y
:true
x||y:true x^
y:true
!x :false

5. Bitwise Operators

Java provides Bit wise operators to manipulate the contents of variables at the bit level. The result of applying bitwise operators between two corresponding bits in the operandsis shown in the Table below.

A	B	~A	A & B	A | B	A ^ B
1	**1**	0	1	1	0
1	**0**	0	0	1	1
0	**1**	1	0	1	1
0	**0**	1	0	0	0

```
public classTest
{
public static void main(String args[])
{
    int a = 60; /* 60 = 0011 1100 */
    intb=13;    /*13=00001101*/intc=0; c=a&
    b;          /* 12 = 0000 1100 */
    System.out.println("a&b="+c);
    c=a|b;      /*61=00111101*/
    System.out.println("a|b="+c); c=a^b;
                /*49=00110001*/
    System.out.println("a^b="+c); c=~a;
                /*-61=11000011*/
    System.out.println("~a="+c);
    c=a<<2;     /* 240 = 1111 0000 */
    System.out.println("a<<2="+c); c=a>>2;
                /* 215 = 1111 */
    System.out.println("a>>2="+c); c=a>>>2;
                /*215=00001111*/
```

```
        System.out.println("a >>> 2 = " + c );
}
}
```

Output:

$java Test

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

6. Compound Assignment operators

The compound operators perform shortcuts in common programming operations. Javahas eleven compound assignment operators.

Syntax: argument1 **operator** = argument2.

Java Assignment Operator Example

```
1.  class OperatorExample
2. {
3. public static void main(String[] args)
4. {
5. int a=10;
6.   a+=3;        //10+3
7. System.out.println(a);
8.  a-=4;       //13-4
9. System.out.println(a);
10. a*=2;      //9*2
11. System.out.println(a);
12. a/=2;      //18/2
13. System.out.println(a);12.
14. }
15. }
```

Output:

13

```
9
18
9
```

7. Conditional Operators

The Conditional operator is the only ternary (operator takes three arguments) operator in Java. The operator evaluates the first argument and, if true, evaluates the second argument.

If the first argument evaluates to false, then the third argument is evaluated. The conditional operator is the expression equivalent of the if-else statement.

The conditional expression can be nested and the conditional operator associates from right to left: **(a?b?c?d:e:f:g) evaluates as (a?(b?(c?d:e):f):g)**

Example:

```
public class TernaryOperatorsDemo {

        public TernaryOperatorsDemo() {
                int x = 10, y = 12, z = 0;
                z = x > y ? x : y;
                System.out.println("z : " + z);
        }
        public static void main(String args[]) {
                new TernaryOperatorsDemo();
        }
}
```

Output:

```
$java TernaryOperatorsDemo
z : 12
```

8. instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as:

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the

Example:

```
public classTest
{
    public static void main(String args[])
   {
        String name = "James";
         // following will return true since name is type of String
         boolean result = name instanceof String;
         System.out.println( result );
     }
}
```

This would produce the following result:

True

OPERATOR PRECEDENCE:

The order in which operators are applied is known as precedence. Operators with a higher precedence are applied before operators with a lower precedence.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left toright
Unary	++ - - ! ~	Rightto left
Multiplicative	* /%	Left toright
Additive	+ -	Left toright
Shift	>> >>> <<	Left toright
Relational	> >= < <=	Left to right
Equality	== !=	Left toright
Bitwise AND	&	Left toright
Bitwise XOR	^	Left toright
Bitwise OR	|	Left toright

Logical AND	&&	Left toright
Logical OR	||	Left toright
Conditional	?:	Rightto left
Assignment	=+=-=*=/=%=>>=<<=&=^=|=	Rightto left
Comma	,	Left toright

9.Write a java program to print 10 numbers using do while statement(7)

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

        int count = 1;
        System.out.println("Printing Numbers from 1 to 10");
        do {
            System.out.println(count++);
        } while (count <= 10);
    }
}
```

Output:

```
Printing Numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
```

10. Write a java programfor finding the volume of box(7)

```
class box
{
        double width;
        double height;
        double depth;
void volume()
{
        System.out.print("\n Box Volume is : "); System.out.println(width*height*depth+" cu.cms");
}
}
public classBoxVolume
{
public static void main(String[] args)
{
      box b1=new box(); // creating object of type box b1.width=10.00;
                                // Accessing instance variables throughobject
      b1.height=10.00;
      b1.depth=10.00;
      b1.volume();              // Accessing method through object
}
}
```

Output:

Box Volume is: 1000.0 cu.cms

11.Explain constructor with example(13)

Definition:

Constructor is a **special type of method** that is used to initialize the object. Constructor is **invoked at the time of object creation**.

➢ Rules for creating constructor:

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type
3. Constructors can be declared public or private (for a Singleton)
4. Constructors can have no-arguments, some arguments and var-args;
5. A constructor is always called with the **new** operator
6. The default constructor is a no-arguments one;
7. If you don't write ANY constructor, the compiler will generate the default one;
8. Constructors CAN'T be **static**, **final** or **abstract**;
9. When overloading constructors (defining methods with the same name but with different arguments lists) you must define them with different arguments lists (as number or as type)

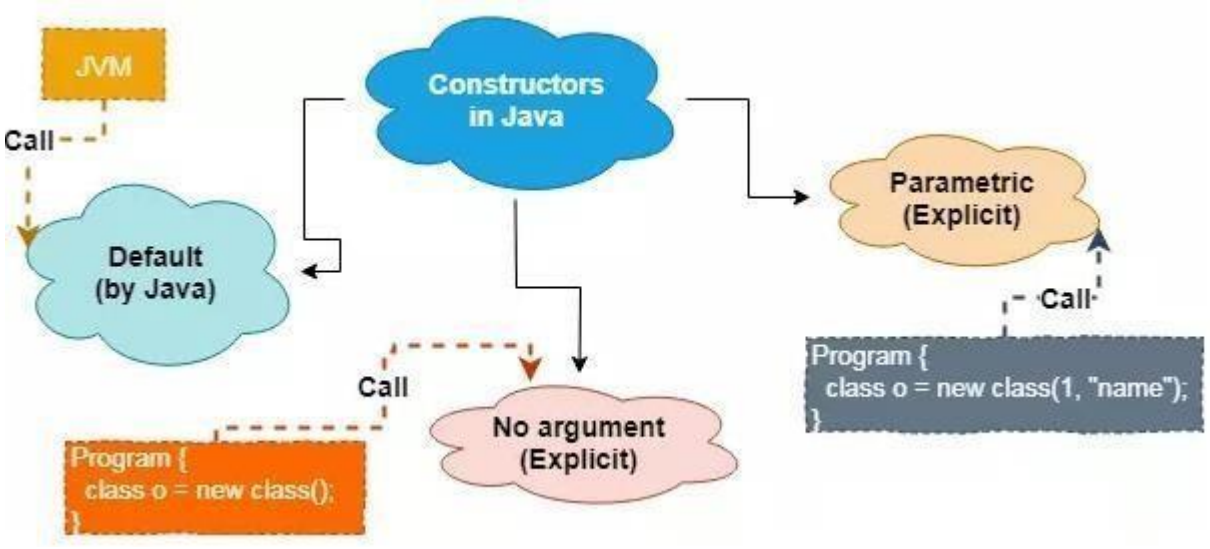
➢ What happens when a constructor is called?

1. All data fields are initialized to their default value (0, false or null).
2. All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.
3. If the first line of the constructor calls a second constructor, then the body of the second constructor is executed.
4. The body of the constructor is executed.

➢ Types of constructors

There are two types of constructors:

1. Default constructor
2. no-arg constructor
3. Parameterized constructor



1. Default Constructor

- **Default constructor refers to a constructor that is automatically created by compilerin the absence of explicit constructors.**

Rule: Ifthereisnoconstructorinaclass,compilerautomaticaalycreatesadefault constructor.
Purpose of Default Constructor: Itisusedtoprovidethedefaultvaluestotheobject members like 0, null etc. depending on the data type.

Example:

```
class student
{
        int id;
        String name;
        void display()
        {
                System.out.println(id+&amp;quot; &amp;quot;+name);
        }
        public static void main(String args[])
        {
                student s1=new student(); student
                s2=new student(); s1.display();
                s2.display();
        }
}
```

Output:null

0 null

2) No-Argument Constructor

- **Constructor without parameters is called no-argument constructor**.

Purpose of No-Arg Constructor: It is used to provide values to be common for all objects of the class.

Syntax of default constructor:

```
Classname()
{
```

Example:

```
/Constructor body
}

class Box
{
  double      width;
  double      height;
  doubledepth;

  // This is the constructor for Box Box()
  {
        System.out.println("Constructing Box…");
        width=10;
        height=10; depth=10;
  }
// Compute and return volume double
 volume()
 {
        return width*height*depth;
 }
}
class BoxDemo
{
        public static void main(String arg[])
        {
                // declare, allocate and initialize Box objects
                Box mybox1=new Box(); Box
                mybox2=new  Box();  double
                vol;

                // Get volume of first box
```

```
        vol=mybox1.volume();
        System.out.println("Volume is " +vol);

        // Get volume of second box
        vol=mybox2.volume(); System.out.println("Volume is "+vol);
    }
}
```

Output:

```
Constructing    Box
Constructing    Box
Volume is1000.0
Volume is1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume.

3 Parameterized Constructor

A constructor that takes parameters is known as parameterized constructor.

Purpose of parameterized constructor

Parameterized constructor is used to provide different values to the distinct objects.

Example:

```
class Box
{
  double    width;
  double    height;
  doubledepth;

  // This is the constructor for Box

  Box(double w, double h, double d)
  {
      width=w;height=h;depth=d;
  }

// Compute and return volume
  double volume()
  {
 return width*height*depth;
  }
}
class BoxDemo
```

```
{
public static void main(String arg[])
{
        // declare, allocate and initialize Box objects Box
        mybox1=new Box(10,20,15); Box
        mybox2=newBox(3,6,9);
        double vol;
        // Get volume of first box

        vol=mybox1.volume();
        System.out.println("Volume is " +vol);

        // Get volume of second box
        vol=mybox2.volume();
        System.out.println("Volume is " +vol);
}
}
```

Output:

Volume is3000.0
Volume is162.0

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

Box mybox1 = new Box(10, 20, 15);

Difference between constructor and method:

There are many differences between constructors and methods. They are given below

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

CONSTRUCTOR OVERLOADING:

Definition:

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading:

```
class Box
{
        double      width;
        double      height;
        doubledepth;

        // constructor used when all the dimensions are specified
        Box(double w, double h, double d)
        {
                width=w;
                height=h;
                depth=d;
        }

        // constructor used when no dimensions are specified
        Box()
        {
                width=-1;
                height=-1;
                depth=-1;
        }

       // constructor used when cube is created
       Box(double len)
       {
                width = height = depth = len;
        }

        // Compute and return volume
     double volume()
        {
                return width*height*depth;
        }
  }

  class ConsOverloadDemo
  {
  public static void main(String arg[])
  {
```

```
// declare, allocate and initialize Box objects
Box mybox1=new Box(10,20,15); Box
mybox2=new Box();
Box mybox3=new Box(7); double vol;

// Get volume of first box
vol=mybox1.volume();
System.out.println("Volume of Box1 is "+vol);

// Get volume of second box
vol=mybox2.volume();
System.out.println("Volume of Box2 is "+vol);
        System.out.println("Volume of Cube is "+vol);



}
}
```

Output:

lume of cube

Volume of Box1 is 3000.0
Volume of Box2 is -1.0
Volume of the cube is 343.0

As we can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

CONSTRUCTOR CHAINING:

Constructor chaining is the process of calling one constructor of a class from another constructor of the same class or another class using the current object of the class.

- It occurs through inheritance.

Ways to achieve Constructor Chaining:

We can achieve constructor chaining in two ways:

- **Within the same class:** If we want to call the constructor from the same class, then we use **this** keyword.
- **From the base class:** If we want to call the constructor that belongs to different classes (parent and child classes), we use the **super** keyword to call the constructor from the base class.

Rules of Constructor Chaining:

- ✓ An expression that uses **this** keyword must be the first line of the constructor.
- ✓ Order does not matter in constructor chaining.
- ✓ There must exist at least one constructor that does not use this

Advantage:

- ✓ Avoids duplicate code while having multiple constructors.
- ✓ Makes code more readable

Example

```
class Shape
{
   int radius,length,breadth;

   Shape(int radius)
   {
      this.radius=radius;
   }
```

```
    Shape(int r,int l,int b)
    {
       this(r); length=l;
       breadth=b;
    }

    void areaCircle()
    {
       System.out.println("Area of Circle is "+(3.14*radius*radius));
    }
    void areaRectangle()
    {
       System.out.println("Area of Rectangle is "+(length*breadth));
    }
}
public class ConstructorChaining
{
    public static void main(String arg[])
    {
    Shape s1=new Shape(5,10,50); s1.areaCircle();
    s1.areaRectangle();
    }
}
```

Output:

Area of Circle is 78.5 Area
of Rectangle is 500

13. Explain the acess specifiers of java with example(13)

Definition:

Access specifiers are used to specify the visibility and accessibility of a class constructors, member variables and methods.

Java classes, fields, constructors and methods can have one of four different accessmodifiers:

1. Public
2. Private
3. Protected
4. Default (package)

1. **Public (anything declared as public can be accessed from anywhere):**
   A variable or method declared/defined with the public modifier can be accessed anywhere in the program through its class objects, through its subclass objects andthrough the objects of classes of other packages also.

2. **Private (anything declared as private can't be seen outside of the class):**
   The instance variable or instance methods declared/initialized as private can be accessedonly by its class. Even its subclass is not able to access the private members.

3. **Protected (anything declared as protected can be accessed by classes in the samepackage and subclasses in the other packages):**
   The protected access specifier makes the instance variables and instance methods visibleto all the classes, subclasses of that package and subclasses of other packages.

4. **Default (can be accessed only by the classes in the same package):**
   The default access modifier is friendly. This is similar to public modifier except only theclasses belonging to a particular package knows the variables and methods.

	PRIVATE	DEFAULT	PROTECTED	PUBLIC
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes	Yes
Different package Non-subclass	No	No	No	Yes

Example: Illustrating the visibility of access specifiers:

Z:\MyPack\FirstClass.java

```
package MyPack;

public classFirstClass
{
   ublic String i="I am public variable"; protected String j="I
   am protected variable";
```

```
    private String k="I am private variable"; String
    r="Idonthaveanymodifier";
}
```

Z:\MyPack2\SecondClass.java

```
package MyPack2;
import MyPack.FirstClass;
class SecondClass extends FirstClass { void
    method()
    {
        System.out.println(i);          // No Error: Will print "I am public variable".
       System.out.println(j);          //No Error:Willprint“Iamprotectedvariable”.
       System.out.println(k); // Error: k has private access in FirstClass
       System.out.println(r); //Error: risnotpublicinFirstClass;cannotbeaccessed
                                  //          from outside package
    }

    public static void main(String arg[])
    {
        SecondClass obj=new SecondClass();
        obj.method();
    }
}
```

Output:

```
I am public variable
I am protected variable

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - k has private access
in MyPack.FirstClass
```

14.Explain the static members of java with example(13)

Static Members are data members (variables) or methods that belong to a static or non-static class rather than to the objects of the class. Hence it is not necessaryto create object of that class to invoke static members.

- ✓ The static can be:
    1. variable (also known as class variable)

2. method (also known as class method)
3. block
4. nested class

❖ **Static Variable:**

- ✓ **When a member variable is declared with the static keyword, then it is called static variable and it can be accessed before any objects of its class are created, and without reference to any object.**
- ✓ **Syntax** to declare a static variable:

[access_spefier] static data_type instance_variable;

- ✓ When a static variable is loaded in memory (static pool) it creates only a single copy of static variable and shared among all the objects of the class.
- ✓ A static variable can be accessed outside of its class directly by the class name anddoesn't need any object.

Syntax : <class-name>.<variable-name>

Advantages of static variable

- ✓ It makes your program **memory efficient** (i.e., it saves memory).

❖ **Static Method:**

If a method is declared with the static keyword , then it is known as static method.

- **Syntax: (defining static method)**

```
[access_specifier] static Return_type method_name(parameter_list)
{
  // method body
}
```

- Syntax to access static method:

```
<class-name>.<method-name>
```

✓ The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

- Methods declared as **static** have several restrictions:
  - ✓ They can only directly call other **static** methods.
- ✓ They can only directly access **static** data.
- ✓ They cannot refer to **this** or **super** in any way.

❖ **Static Block:**

Static block is used to initialize the static data member like constructors helps toinitialize instance members and it gets executed exactly once, when the class is first loaded.

- It is executed before main method at the time of class loading in JVM.

- **Syntax:**

```
class classname
{
   static
   {
      // block of statements
   }
}
```

The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

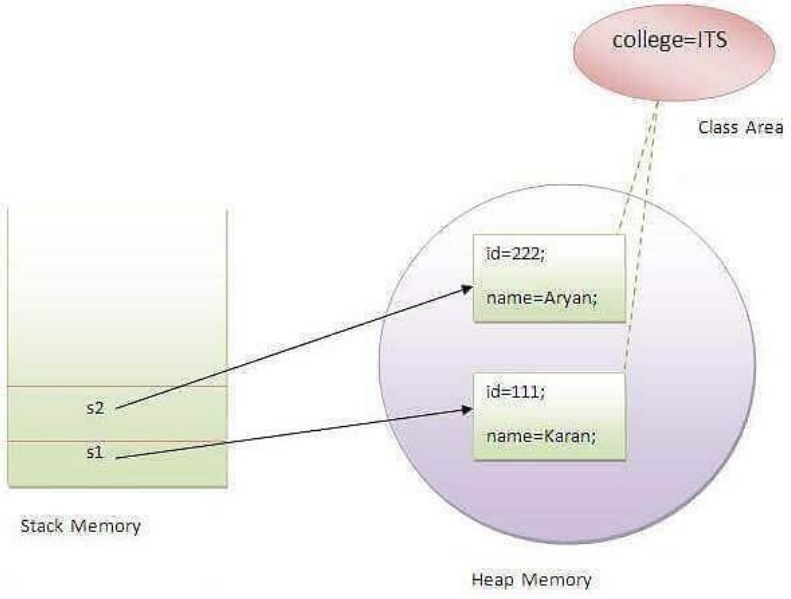
//Demonstrate static variables, methods, and blocks.

```
1.  class Student
2.  {
3.  int rollno;
4.  String name;
5.  static String college = "ITS";
6.  //static method to change the value of static variable
7.  static voidchange(){
8.  college ="BBDIT";
9.  }
10. //constructor to initialize the variable
11. Student(int r, String n){
12. rollno = r;
13. name=n;
14. }
15. //method to display values
16. void display()
17. {
18. System.out.println(rollno+" "+name+" "+college);
19. }
20. }
21. //Test class to create and display the values of object
22. public class TestStaticMembers
```

```
23.         {
24.         static
25.         {
26.         System.out.println(-*** STATIC MEMBERS – DEMO ***‖);
27.         }
28.
29.         public static void main(String args[])
30.         {
31.         Student.change(); //calling changemethod
32.         //creating objects
33.         Students1=newStudent(111,"Karan");
34.         Students2=newStudent(222,"Aryan");
35.         Students3=newStudent(333,"Sonoo");
36.         //calling display method
37.         s1.display();
38.         s2.display();
39.         s3.display();
40.         }
41.         }
```

Here is the output of this program:

```
*** STATIC MEMBERS – DEMO ***
111         Karan           BBDIT
222         Aryan           BBDIT
333         Sonoo           BBDIT
```



:

15.Explain in detail about java doc comments(13)

Definition:

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code. Java documentation can becreated as part of the source code.

Input: Java source files (.java)

- Individual sourcefiles
- Root directory of the source files

Output: HTML files documenting specification of java code

- One file for each class defined
- Package and overview files

HOW TO INSERT COMMENTS?

The javadoc utility extracts information for the following items:

- Packages
- Public classes and interfaces
- Public and protected methods
- Public and protected fields

Each comment is placed immediately above the feature it describes.

Format:

- ✓ A Javadoc comment precedes similar to a multi-line comment except that it beginswith a forward slash followed by two asterisks (/**) and ends with a */
- ✓ Each/**…*/documentation comment contains free-form text followed by tags.
- ✓ A tag starts with an @, such as @author or @param.
- ✓ The first sentence of the free-form text should be a summary statement.
- ✓ The javadoc utility automatically generates summary pages that extract thesesentences.
- ✓ In the free-form text, you can use HTML modifiers such as ... foremphasis, <code>...</code> for a monospaced —typewriter font, ... for strong emphasis, and even <img ...> to include an image.
- ✓ Example:

```
/**
This is a <b>doc</b> comment.
*/
```

TYPES OF COMMENTS:

1. Class Comments

The class comment must be placed *after* any import statements, directly before the class definition.
Example:

```
import java.io.*;
/** class comments should be written here */Public class sample
{
….
}
```

2. Method Comments

The method comments must be placed immediately before the method that it describes.

Tags used:

Tag	Description	Syntax
@param	It describes the method parameter	**@param name description**
@return	This tag describes the return value from a method with the exception void methods and constructors.	**@return description**
@throws	This tag describes the method that throws an exception.	**@throws class description**

Example:

```
/** adding two numbers
@param a & b are two numbers to be added
@return the result of addition
**/
public double add(int a,int b)
{
int c=a+b;
returnc;
}
```

3. ***Field Comments***

Field comments are used to document public fields—generally that means static constants.

For example:

```
/**
* Accountnumber
*/
public static final int acc_no = 101;
```

4. ***General Comments***

Tag	Description	Syntax
The following tags can be used in class documentation comments		
@author	This tag makes an —author entry. You can have multiple @author tags, one for each author.	**@author name**
@version	This tag makes a —version‖ entry. The text can be any description of the current version.	**@version text**
The following tags can be used in all documentation comments		
@since	This tag makes a —since‖ entry. The text can be any description of the version that introduced this feature. For example, @since version 1.7.1	**@since text**
@deprecate d	This tag adds a comment that the class, method, or variable should no longer be used. The text should suggest a replacement. For example: @deprecated Use <code>setVisible(true)</code>instead	**@deprecated text**
Hyperlinks to other relevant parts of the javadoc documentation, or to external documents,with the @see and @link tags.		
@link	This tag place hyperlinks to other classes or methods anywhere in any of your documentationcomments.	**{@link package.class#feature label}**

@see	This tag adds a hyperlink in the —see also section. It can be used with both classes and methods. Here, reference can be one of the following: package.class#feature label label "text" Example: @see -Core java 2 @see Core Java	**@see reference**

COMMENT EXTRACTION

Here, *docDirectory* is the name of the directory where you want the HTML files to go. Follow these steps:

1. Change to the directory that contains the source files you want to document.
2. To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the java file.

Here, docDirectory is the name of the directory where you want the HTML files to go.

Follow these steps:

1 Change to the directory that contains the source files you want to document.
2 Run the command

javadoc -d docDirectory nameOfPackage

for a single package. Or run

javadoc -d docDirectory nameOfPackage1 nameOfPackage2...

to document multiple packages.

If your files are in the default package, then instead run

javadoc -d docDirectory *.java

If you omit the -d docDirectory option, then the HTML files are extracted to the current directory.

UNIT II INHERITANCE, PACKAGES AND INTERFACES

1.Explain method overloading with example(13)

Method Overloading is a feature in Java that allows a class to have **more than one methods having same name**, but with **different signatures**

Advantage:

- ✓ Method Overloading increases the readability of the program.
- ✓ Provides the flexibility to use similar method with different parameters.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters. (Different number of parameters in argument list)

For example: This is a valid case of overloading add(int,
int)
add(int, int, int)

2. Data type of parameters. (Difference in data type of parameters)

For example:
add(int, int)
add(int, float)

3. Sequence of Data type of parameters.

For example:
add(int, float)
add(float, int)

Rules for Method Overloading:

1. First and important rule to overload a method in java is to change method signature.
2. Return type of method is never part of method signature, so only changing the return type of method does not amount to method overloading.

Example:

```
class Add
{
   static int sum(int a, int b)
  {
      return a+b;
  }
  static float sum(int a, int b)


 {
     return a+b;
 }
public static void main(String arg[])
{
    System.out.println(sum(10,20));
    System.out.println(sum(15,25));
}
}
```

Output:

Compile by: javac TestOverloading3.java

Add.java:7: error: method sum(int,int) is already defined in class Add static
float sum(int a, int b)
^
1 error

Method Overloading and Type Promotion

Type Promotion: When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion

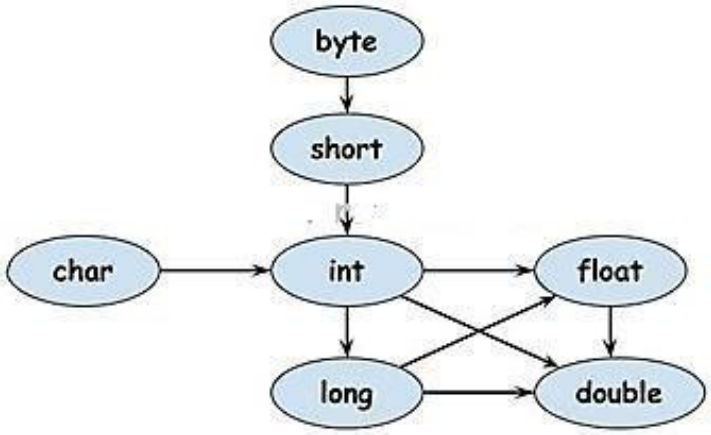
Type Promotion in Method Overloading:

One type is promoted to another implicitly if no matching datatype is found.

Type Promotion Table:

The **data type on the left side can be promoted to the any of the data type present in the right side of it**.

```
byte → short → int → long → double
short → int → long → float → double
int → long → float → double
float → double
long → float → double
char → int → long → float → double
```



Example: Method Overloading with Type Promotion:

```
class Overloading
{
     void sum(int a, float b)
    {
        System.out.println(a+b);
    }
     void sum(int a, int b, int c)
    {
        System.out.println(a+b+c);
    }

  public static void main(String args[])
{
        OverloadingCalculation1 obj=new OverloadingCalculation1();
        obj.sum(20,20);          //now second int literal will be promoted to float
        obj.sum(100,'A');        //Character literal will be promoted to float
        obj.sum(20,20,20);
  }
}
```

OUTPUT:

```
40.0
165.0
60
```

2.Explain object as a parameter with example(13)

Java is strictly pass-by-value. But the scenario may change when the parameter passed is of primitive type or reference type.

- If we pass a primitive type to a method, then it is called **pass-by-value** or call-by-value.
- If we pass an object to a method, then it is called **pass-by-reference** or call-by-reference.

Object as a parameter is a way to establish communication between two or more objects of the same class or different class as well.

Pass-by-value vs. Pass-by-reference:

Pass-by-value (Value as parameter)	**Pass-by-reference (Object as parameter)**
Only values are passes to the function parameters. So any modifications done in the formal parameter will not affect the value of actual parameter	Reference to the object is passed. So any modifications done through the object will affect the actual object.
Caller and Callee method will have two independent variables with same value.	Caller and Callee methods use the same reference for the object.
Callee method will not have any access to the actual parameter	Callee method will have the direct reference to the actual object
Requires more memory	Requires less memory

class CallByVal
{
void Increment(int count)
{
count=count+10;
}
}
public class CallByValueDemo { public
static void main(String arg[])
{
CallByVal ob1=new CallByVal();
int count=100;
System.out.println("Value of Count before method call = "+count);
	class CallByRef
	{
	int count=0;
	CallByRef(int c)
	{
	count=c;
	}
	static void Increment(CallByRef obj) {
	obj.count=obj.count+10;
	}
	
	public static void main(String arg[]) {
	CallByRef ob1=new CallByRef(10);
	System.out.println("Value of Count (Object 1) before

```
    ob1.Increment(count);
    System.out.println("Value of Count after
                        method call = "+count);
  }
}
```

OUTPUT:

Value of Count before method call = 100 Value of Count after method call = 100

```
                        method call = "+ob1.count);
    Increment(ob1);
    System.out.println("Value of Count (Object 1) after
                        method call = "+ob1.count);

  } }
```

OUTPUT:

Value of Count (Object 1) before method call = 10
Value of Count (Object 1) after method call = 20

Returning Objects:

In Java, a method can return any type of data. Return type may any primitive data type or class type (i.e. object). As a method takes objects as parameters, it can also return objects as return value.

Example:

```
class Add
{
  int num1,num2,sum;

  static Add calculateSum(Add a1,Add a2)
  {
    Add a3=new Add();
    a3.num1=a1.num1+a1.num2;
    a3.num2=a2.num1+a2.num2;
    a3.sum=a3.num1+a3.num2; return
    a3;
  }

  public static void main(String arg[])
  {
    Add ob1=new Add();
    ob1.num1=10;
    ob1.num2=15;

    Add ob2=new Add();
    ob2.num1=100;
    ob2.num2=150;

    Add ob3=calculateSum(ob1,ob2);
    System.out.println("Object 1 -> Sum = "+ob1.sum);
```

```
        System.out.println("Object 2 -> Sum = "+ob2.sum);
        System.out.println("Object 3 -> Sum = "+ob3.sum);
    }
}
```

OUTPUT:

```
Object 1 -> Sum = 0
Object 2 -> Sum = 0
Object 3 -> Sum = 275
```

3.Explain the inner class of java(13)

Definition:

An inner class is a class that is defined inside another class.
Inner classes let you make one class a member of another class. Just as classes have member variables and methods, a class can also have member classes.

Benefits:

1. Name control
2. Access control
3. Code becomes more readable and maintainable because it locally group related classes in one place.

Syntax: For declaring Inner classes

```
[modifier] class OuterClassName
{
      ---- Code -----
       [modifier] class InnerClassName
       {
       ---- Code ----
       }
}
```

- **Instantiating an Inner Class:**

Two Methods:

1. **Instantiating an Inner class from outside the outer class:**
   To instantiate an instance of an inner class, you must have an instance of the outer class.
   Syntax:
   OuterClass.InnerClass objectName=OuterObj.new InnerClass();

2. **Instantiating an Inner Class from Within Code in the Outer Class:**
   From inside the outer class instance code, use the inner class name in the normal way:

Syntax:

InnerClassName obj=new InnerClassName();

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested class **can access all the members (data members and methods) of outer class** including private.
2) Nested classes are used **to develop more readable and maintainable code**.
3) **Code Optimization**: It requires less code to write.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

1. Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer
{
 //code
    class Inner
    {
       //code
    }
}
```

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
1.  class TestMemberOuter1
2.  {
3.   private int data=30;
4.   class Inner
5.  {
6.    void msg()
7.  {
8.  System.out.println("data is "+data); 9.
    }
10.  }
11.  public static void main(String args[])
12.  {
13.   TestMemberOuter1 obj=new TestMemberOuter1();
14.   TestMemberOuter1.Inner in=obj.new Inner();
15.   in.msg();
16.  }
17.  }
```

Output:

data is 30

2. Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

```
1.  abstract class Person
2.  {
3.    abstract void eat();
4.  }
5.  class TestAnonymousInner
6.  {
7.   public static void main(String args[]) 8.
    {
9.    Person p=new Person()
10. {
11.   void eat()
12. {
13.     System.out.println("nice fruits");
14. }
15.   };
16.   p.eat();
17.  }
18. }
```

Output:

nice fruits

Java anonymous inner class example using interface

```
1.  interface Eatable
2.  {
3.   void eat();
```

```
4.    }
5.    class TestAnnonymousInner1
6.    {
7.     public static void main(String args[]) 8.
      {
9.     Eatable e=new Eatable()
10.    {
11.    public void eat(){System.out.println("nice fruits");
12.    }
13.   };
14.   e.eat();
15.   }
16.  }
```

Output:

```
nice fruits
```

3. Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
1.    public class localInner1
2.    {
3.     private int data=30;//instance variable
4.    void display()
5.    {
6.    int value=50;
7.     class Local
8.    {
9.      void msg()
10.       {
11.        System.out.println(data);
12.        System.out.println(value);
13.       }
14.   }
```

```
15.    Local l=new Local();
16.    l.msg();
17.   }
18.   public static void main(String args[])
19.   {
20.    localInner1 obj=new localInner1();
21.    obj.display();
22.   }
23.  }
```

Output:

```
30
50
```

Rules for Java Local Inner class

1. Local inner class cannot be invoked from outside the method.
2. Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.
3. Local variable can't be private, public or protected.

Properties:

1. Completely hidden from the outside world.
2. Cannot access the local variables of the method (in which they are defined), but the local variables has to be declared final to access.

4. Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java static nested class example with instance method

```
1.    class TestOuter1
2.    {
3.     static int data=30;
4.     static class Inner
5.    {
6.      void msg()
```

```
7.        {
8.          System.out.println("data is "+data); 9.
           }
10.       }
11.       public static void main(String args[])
12.      {
13.       TestOuter1.Inner obj=new TestOuter1.Inner();
14.       obj.msg();
15.       }
16.      }
```

Output:

```
data is 30
```

Java static nested class example with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
1.  class TestOuter2{
2.    static int data=30;
3.  static class Inner
4.  {
5.     static void msg()
6.      {
7.       System.out.println("data is "+data); 8.
         }
9.    }
10. public static void main(String args[])
11. {
12.   TestOuter2.Inner.msg();//no need to create the instance of static nested class
13. }
14. }
```

Output:

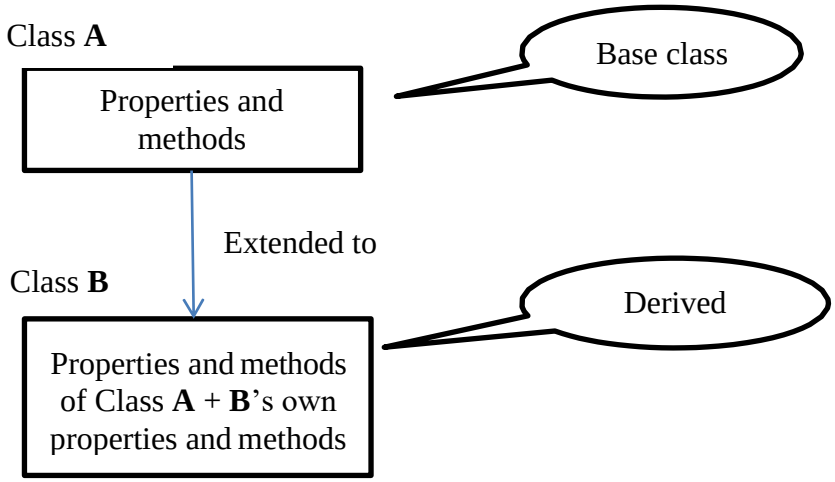
```
data is 30
```

4.Explain inheritance with example(13)

Definition:

Inheritance is a process of deriving a new class from existing class, also called as "extending a class".

- ✓ The class whose property is being inherited by another class is called **"base class"** (or) **"parent class"** (or) **"super class"**.
- ✓ The class that inherits a particular property or a set of properties from the base class is called **"derived class"** (or) **"child class"** (or) **"sub class"**.



- ✓ Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

➢ **ADVANTAGES OF INHERITANCE:**

- **Reusability of Code:**
  - ✓ code reusability (Code reusability means that we can add extra features to an existing class without modifying it).
- **Effort and Time Saving:**
  - ✓ The advantage of reusability saves the programmer time and effort. Since the main code written can be reused in various situations as needed.
- **Increased Reliability:**
  - ✓ The program with inheritance becomes more understandable and easily maintainable as the sub classes are created from the existing reliably working classes.

➢ **"extends" KEYWORD:**

- ✓ Inheriting a class means creating a new class as an extension of another class.
- ✓ The **extends** keyword is used to inherit a class from existing class.
- ✓ The general form of a **class** declaration that inherits a superclass is shown here:
- ✓ **Syntax:**

```
[access_specifier] class subclass_name extends superclass_name
{
        // body of class
}
```

Characteristics of Class Inheritance:

1. A class cannot be inherited from more than one base class.
2. Sub class can access only the non-private members of the super class.
3. Private data members of a super class are local only to that class.
4. Protected features in Java are visible to all subclasses as well as all other classes in the same package.

✓ **Example:**

```
class Vehicle
{
   String brand;
   String color;
}
class Car extends Vehicle
{
   int totalDoor;
}

class Bike extends Vehicle
{
}
```

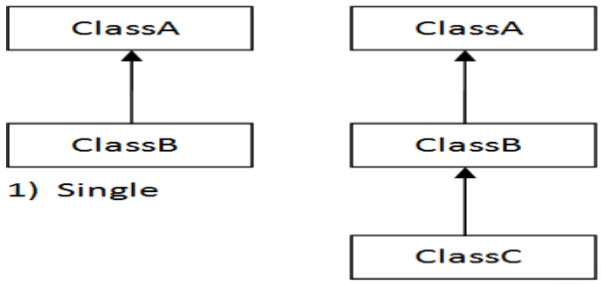
In the above example, Vehicle is the **super class** or base class that holds the common property of Car and Bike. Car and Bike is the **sub class** or derived class that inherits the property of class Vehicle **extends** is the keyword used to inherit a class.

➢ TYPES OF INHERITACE:

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance

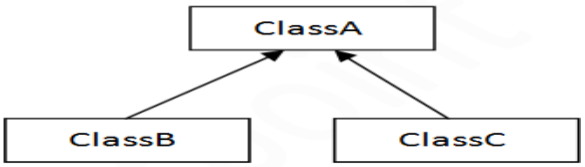
Note: The following inheritance types are not directly supported in Java.

4. Hierarchical Inheritance
5. Hybrid Inheritance

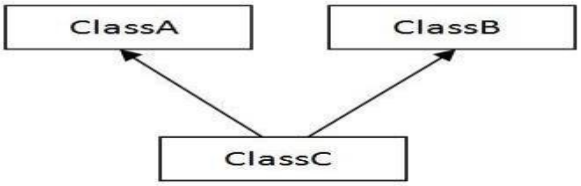


1) Single

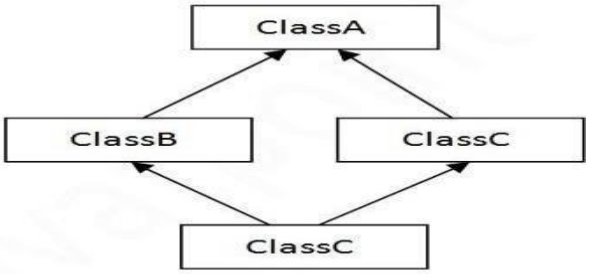
2) Multilevel



3) Hierarchical

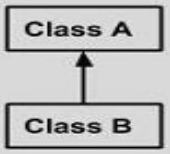


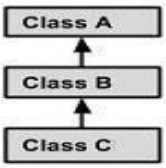
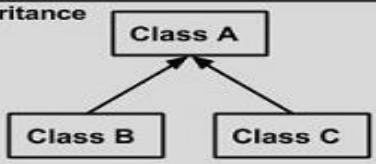
4) Multiple

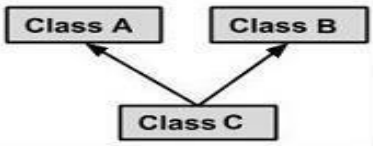


5) Hybrid

.

Single Inheritance 	public class A {
	.....
	}
	public class B **extends** A {
	.....
	}

Multi Level Inheritance 	public class A { .....} public class B **extends** A {.....} public class C **extends** B {..... }
Hierarchical Inheritance 	public class A { .....} public class B **extends** A {.....} public class C **extends** A {..... }

Multiple Inheritance 	public class A { .....}
	public class B {.....}
	public class C **extends** A,B {
	.....
	} // Java does not support mutiple Inheritance

1. SINGLE INHERITANCE

The process of creating only one subclass from only one super class is known as **Single Inheritance.**

- ✓ Only two classes are involved in this inheritance.
- ✓ The subclass can access all the members of super class.

Example: Animal → Dog

```
1. class Animal
2. {
3.     void eat()
4.     {
5.         System.out.println("eating...");
6.     }
7. }
8. class Dog extends Animal
9. {
10. void bark()
11. {
12.       System.out.println("barking...");
13.

} 14.}
15.class TestInheritance
16.{
17.    public static void main(String args[])
18.    {
19.        Dog d=new Dog();
20.        d.bark();
21.        d.eat();
22.

} 23.}
```

Output:

```
$java TestInheritance
barking...
eating...
```

2. MULTILEVEL INHERITANCE:

- ✓ The process of creating a new sub class from an already inherited sub class is known as **Multilevel Inheritance**.
- ✓ Multiple classes are involved in inheritance, but one class extends only one.
- ✓ The lowermost subclass can make use of all its super classes' members.
- ✓ Multilevel inheritance is an indirect way of implementing multiple inheritance.
- ✓ **Example: Animal ➷ Dog ➷ BabyDog**

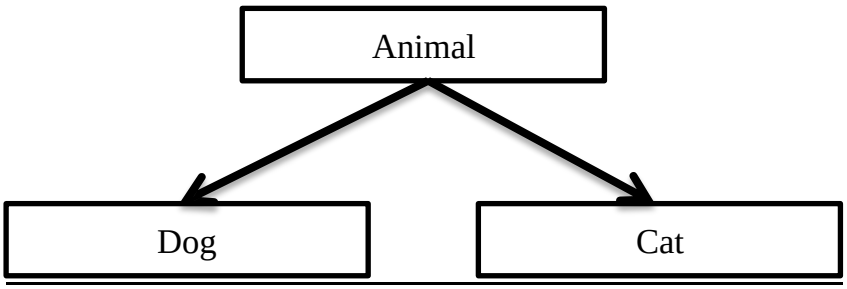
```
1.  class Animal
2.  {
3.    void eat()
4.    {
5.        System.out.println("eating...");
6.    }
7.  }
8.  class Dog extends Animal
9.  {
10.     void bark()
11.     {
12.       System.out.println("barking...");
13.     }
14.   }
15. class BabyDog extends Dog
16. {
17.    void weep()
18.    {
19.       System.out.println("weeping...");
20.    }
21. }
22. class TestInheritance2
23. {
24.   public static void main(String args[]) {
25.       BabyDog d=new BabyDog();
26.       d.weep();
27.       d.bark();
28.       d.eat();
29.   }
30. }
```

Output:

```
$java TestInheritance2
weeping...
barking...
eating..
```

3. HIERARCHICAL INHERITANCE

✓ The process of creating more than one sub classes from one super class is called **Hierarchical Inheritance.**



✓ **Example:**

```
1.  class Animal
2.  {
3.  void eat()
4.  {
5.  System.out.println("eating...");
6.  }
7.  }
8.  class Dog extends Animal
9.  {
10. void bark()
11. {
12. System.out.println("barking...");
13. }
14. }
15. class Cat extends Animal
16. {
17. void meow()
18. {
19. System.out.println("meowing...");
20. }
```

```
21.     }
22.     class TestInheritance3
23.     {
24.     public static void main(String args[])
25.     {
26.     Cat c=new Cat();
27.     c.meow();
28.     c.eat();
29.     //c.bark();//C.T.Error
30.     }
31.     }
```

Output:

```
meowing...
eating...
```

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritances is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, therewill be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A  {
    void msg()
   {
       System.out.println("Hello");
   }
}
class B  {
   void msg()
   {
        System.out.println("Welcome");
   }
}
class C extends A,B // this is multiple inheritance which is ERROR
  {
        Public Static void main(String args[])
        {
           C obj=new C();
           obj.msg();//Now which msg() method would be i
nvoked?
        }
}
```

Multiple Inheritance using Interface

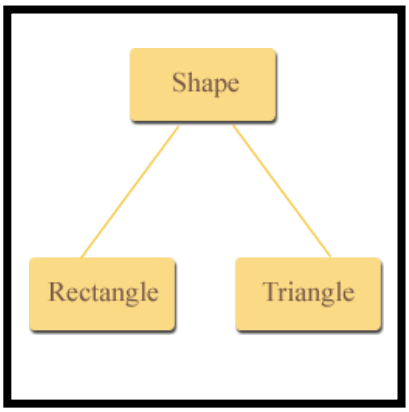
```
interface Printable {
    void print();
}

interface Showable {
    void show();
}
 class A implements Printable, Showable {
       public void print() {
           System.out.println("Hello");
       }
      public void show() {
          System.out.println("Welcome");
      }
     public static void main(String args[]) {
          A obj = new A();
          obj.print();
          obj.show();
     }
}
```

5.Explain the private members of java(13)

The private members of a class cannot be directly accessed outside the class. Following table describes the difference

Modifier	Class	Package	subclass	World
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No

Following program illustrates how the methods of a subclass can directly access a protected member of the superclass.



Consider two kinds of shapes: **rectangles and triangles**. These two shapes have certain common properties height and a width (or base).

This could be represented in the world of classes with a **class Shapes** from which we would derive the two other ones : Rectangle and Triangle

Program : (Shape.java)

```
public class Shape
{
    protected double height; // To hold height. protected
    double width; //To hold width or base
```

```
    public void setValues(double height, double width)
   {
         this.height = height; this.width
         = width;
   }
}
```

Program : (Rectangle.java)

```
public class Rectangle extends Shape
{
     public double getArea()
    {
       return height * width; //accessing protected members
    }
}
```

Program : (Triangle.java)

```
public class Triangle extends Shape
{
            public double getArea()
           {
                 return height * width / 2; //accessing protected members
           }
}
```

Program : (TestProgram.java)

```
public class TestProgram
{
      public static void main(String[] args)
     {
          //Create object of Rectangle.
           Rectangle rectangle = new Rectangle();

         //Create object of Triangle. Triangle
         triangle = new Triangle();

         //Set values in rectangle object
        rectangle.setValues(5,4);
```

```
        //Set values in trianlge object triangle.setValues(5,10);

       // Display the area of rectangle. System.out.println("Area
        of rectangle : " + rectangle.getArea());

     // Display the area of triangle. System.out.println("Area of
      triangle : " + triangle.getArea());
    }
}
```

Output :

Area of rectangle : 20.0

Area of triangle : 25.0

6.Explain the super keyword with example(13)

- ✓ Super is a special keyword that directs the compiler to invoke the superclass members. It is used to refer to the parent class of the class in which the keyword is used.
- ✓ **super keyword is used for the following three purposes**:
  1. To invoke superclass constructor.
  2. To invoke superclass members variables.
  3. To invoke superclass methods.

1. Invoking a superclass constructor:

- ✓ **super** as a standalone statement(ie. super()) represents a call to a constructor of the superclass.
- ✓ A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

```
super(); or
super(parameter-list);
```

- ✓ Here, parameter-list specifies any parameters needed by the constructor in the superclass.
- ✓ **super( )** must always be the first statement executed inside a subclassconstructor.
- ✓ The compiler implicitly calls the base class's no-parameter constructor or default constructor.
- ✓ If the superclass has parameterized constructor and the subclass constructor does not call superclass constructor explicitly, then the Java compiler reports an error.

2. Invoking a superclass members (variables and methods):

(i) Accessing the instance member variables of the superclass:

Syntax:

```
super.membervariable;
```

(ii) Accessing the methos of the superclass:

Syntax:

```
super.methodName();
```

This call is particularly necessary while calling a method of the super class that is overridden in the subclass.

✓ If a parent class contains a finalize() method, it must be called explicitly by the derived class's finalize() method.

```
super.finalize();
```

Example:

```
class A       // super class
{
  int i;
  A(String str)      //superclass constructor
  {
    System.out.println(" Welcome to "+str);
  }
  void show()        //superclass method
  {
    System.out.println(" Thank You!");
  }
}
class B extends A
{
  int i;     // hides the superclass variable 'i'.
  B(int a, int b)     // subclass constructor
  {
    super("Java Programming");      // invoking superclass constructor
    super.i=a;      //accessing superclass member variable
    i=b;
  }
  // Mehtod overriding
  @Override
  void show()
  {
    System.out.println(" i in superclass : "+super.i);
    System.out.println(" i in subclass : "+i);
    super.show();       // invoking superclass method
```

```
    }
  }
  public class UseSuper {
    public static void main(String[] args) {
      B objB=new B(1,2);      // subclass object construction
      objB.show();        // call to subclass method show()
    }
  }
```

Output:

```
Welcome to Java Programming
        i in superclass : 1 i
        in subclass : 2
        Thank You!
```

7.Explain method overriding with example(13)

The process of a subclass redefining a method contained in the superclass (with the same method signature) is called Method Overriding.

- ✓ When a method in a subclass has the same name and type signature as a method in its superclass, then the method in subclass is said to override a method in the superclass.

Example:

```
class Bank
{
int getRateOfInterest()// super class method
{
return 0;
}
}
class Axis extends Bank// subclass of bank
{
int getRateOfInterest()// overriding the superclass method
{
return 6;
}
}
class ICICI extends Bank// subclass of Bank
```

```
{
        int getRateOfInterest()// overriding the superclass method
{
return 15;
}
}
// Mainclass
class BankTest
{
public static void main(String[] a)
{
Axis a=new Axis();
ICICI i=new ICICI();
// following method call invokes the overridden method of subclass AXIS
System.out.println("AXIS: Rate of Interest = "+a.getRateOfInterest());

// following method call invokes the overridden method of subclass ICICI
System.out.println("ICICI: Rate of Interest = "+i.getRateOfInterest());
}
}
```

Output:

Z:\> java BankTest

AXIS: Rate of Interest = 6

ICICI: Rate of Interest = 15

- **RULES FOR METHOD OVERRIDING:**

  - ✓ The method signature must be same for all overridden methods.
  - ✓ Instance methods can be overridden only if they are inherited by the subclass.
  - ✓ A method declared final cannot be overridden.
  - ✓ A method declared static cannot be overridden but can be re-declared.
  - ✓ If a method cannot be inherited, then it cannot be overridden.
  - ✓ Constructors cannot be overridden.

- **ADVANTAGE OF JAVA METHOD OVERRIDING**

  - ✓ Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
  - ✓ Method Overriding is used for Runtime Polymorphism

8.Explain dynamic method dispatch with example(13)

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Example that illustrate 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       // dynamic method dispatch
r.callme();// calls A’s version of callme()

r = b;// r refers to an B object r.callme();//
calls B’s version of callme()
```

```
r = c;// r refers to an C object r.callme();//
calls C's version of callme()
}
}
```

The output from the program is shown here:

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

- **DIFFERENCE BETWEEN METHOD OVERLOADING AND METHOD OVERRIDING IN JAVA:**

	Method Overloading	Method Overriding
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to "add" or "extend" more to method's behavior.	Method Overriding is to "Change" existing behavior of method.

9.Explain Abstraction of java with example(13)

Abstraction is a process of hiding the implementation details and showing only the essential features to the user.

- ✓ For example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- ✓ Abstraction lets you focus on what the object does instead of how it does it.

- **Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

- **Abstract Classes:**

✓ **Syntax to declare the abstract class:**

```
abstract class <class_name>
{
    Member variables; Concrete
    methods { } Abstract
    methods();
}
```

✓ Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.

Properties of abstract class:

- abstract keyword is used to make a class abstract.
- Abstract class can't be instantiated.
- If a class has abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- Abstract classes can have both concrete methods and abstract methods.
- The subclass of abstract class must implement all the abstract methods unless the subclass is also an abstract class.
- A constructor of an abstract class can be defined and can be invoked by the subclasses.
- We can run abstract class like any other class if it has main() method.

Example:

```
abstract class GraphicObject {
  int x, y;
  ...
  void moveTo(int newX, int newY) {
    ...
  }
  abstract void draw(); abstract
  void resize();
}
```

- **Abstract Methods:**

A method that is declared as abstract and does not have implementation is known as **abstract method.** It acts as placeholder methods that are implemented in the subclasses.

✓ **Syntax to declare a abstract method:**

```
abstract class classname
{

abstract return_type <method_name>(parameter_list);//no braces{}
        // no implementation required
                ……..
}
```

✓Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Properties of abstract methods:

- The abstract keyword is also used to declare a method as abstract.
- An abstract method consists of a method signature, but no method body.
- If a class includes abstract methods, the class itself must be declared abstract.
- Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces as follows:

```
public abstract class Employee { private
  String name;
  private String address; private
  int number;
  public abstract double computePay();
  //Remainder of class definition
}
```

- Any child class must either override the abstract method or declare itself abstract.

10..Write a Java program to create an abstract class named Shape that contains 2 integers and an empty method named PrintArea(). Provide 3 classes named Rectangle, Triangle and Circle such that each one of the classes extends the class Shape. Each one of the classes contain only the method PrintArea() that prints the area of the given shape.(13)

```
abstract class shape
{
    int x, y;
    abstract void printArea();
}
```

```
class Rectangle extends shape
{
void printArea()
{
      System.out.println("Area of Rectangle is " + x * y);
}
}
class Triangle extends shape
{
void printArea()
{
System.out.println("Area of Triangle is " + (x * y) / 2);
}
}
class Circle extends shape
{
void printArea()
{
        System.out.println("Area of Circle is " + (22 * x * x) / 7);
}
}
class abs
{
public static void main(String[] args)
{
Rectangle r = new Rectangle();
r.x = 10;
r.y = 20;
r.printArea();

System.out.println("---------------------------------------------- ");

Triangle t = new Triangle();
t.x = 30;
t.y = 35;
t.printArea();

System.out.println("---------------------------------------------- ");
```

```
Circle c = new Circle();
c.x = 2;
c.printArea();

System.out.println("---------------------------------------------- ");
}
}
```

Output:

```
D:\>javac abs.java
D:\>java abs
Area of Rectangle is 200
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
Area of Triangle is 525
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
area of Circle is 12
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
```


11.Explain java packages with example(13)

Definition:

A Package can be defined as a collection of classes, interfaces, enumerations and annotations, providing access protection and name space management.

- ✓ Package can be categorized in two form:
  1. Built-in package
  2. user-defined package.

Packages	Description
Java.lang	It is a default package which contain primitive data type, displaying result on console screen, obtaining garbage collector etc.
java.io	It used for developing file handling applications, such as, opening the file in read or write mode, reading or writing the data, etc.
java.awt	This package is used for developing GUI (Graphic User Interface) components such as buttons, check boxes, scroll boxes, etc.
Java. applet	This package is used for developing browser oriented applications.
java.net	This package is used for developing client server applications.
java.util	Contains utility classes which implement data structures like Hash Table, Dictionary, etc.
java.sql	This package is used for retrieving the data from data base and performing various operations on data base.

Table: List of Built-in Packages

Advantage of Package:

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- Package provides access protection.
- Package removes naming collision.
- To bundle classes and interface
- The classes of one package are isolated from the classes of another package
- Provides reusability of code
- We can create our own package or extend already available package

: CREATING USER DEFINED PACKAGES:

Java package created by user to categorize their project's classes and interface are known as user-defined packages.

- ✓ When creating a package, you should choose a name for the package.
- ✓ Put a **package** statement with that name at the top of every source file that contains the classes and interfaces.
- ✓ The **package** statement should be the first line in the source file.
- ✓ There can be only one package statement in each source file

- ✓ **Syntax:**

```
package package_name.[sub_package_name]; public
class classname
{ ……..
  ……..
}
```

- ✓ Steps involved in creating user-defined package:
  1. Create a directory which has the same name as the package.
  2. Include package statement along with the package name as the first statement in the program.
  3. Write class declarations.
  4. Save the file in this directory as "name of class.java".
  5. Compile this file using java compiler.

- ✓ Example:

```
package pack; public
class class1 {
public static void greet()
{ System.out.println("Hello"); }
}
```

To create the above package,

1. Create a directory called pack.
2. Open a new file and enter the code given above.
3. Save the file as class1.java in the directory.
4. A package called pack has now been created which contains one class class1.

: ACCESSING A PACKAGE (using "import" keyword):

- The import keyword is used to make the classes and interface of another package accessible to the current package.

Syntax:

```
import package1[.package2][.package3].classname or *;
```

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

✓ **Using packagename.***

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

✓ **Using packagename.classname**

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

✓ **Using fully qualified name**

- If you use fully qualified name then only declared class of this package will be accessible.
- Example :

greeting.java (create a folder named "pack" in **F:** and save **)**

```
package pack;
public class greeting{
public static void greet()
{ System.out.println("Hello! Good Morning!"); }
}
```

FactorialClass.java (create a folder named "Factorial" inside **F:\pack** and save**)**

```
package Factorial;
public class FactorialClass
{
```

```
public int fact(int a)
{
if(a==1)
return 1;
else
return a*fact(a-1);
}
}
```

ImportClass.java (save the file in F:\ **)**

```
import java.lang.*; // using import package.*
import pack.Factorial.FactorialClass; // using import package.subpackage.class;
import java.util.Scanner;
public class ImportClass
{
public static void main(String[] arg)
{
int n;
Scanner in=new Scanner(System.in);
System.out.println("Enter a Number: "); n=in.nextInt();
pack.greeting p1=new pack.greeting(); // using fully qualified name
p1.greet();
FactorialClass fobj=new FactorialClass();
System.out.println("Factorial of "+n+" = "+fobj.fact(n));
System.out.println("Power("+n+",2) = "+Math.pow(n,2));
}
}
```

Output:

```
F:\>java ImportClass
Enter a Number:
5
Hello! Good Morning!
Factorial of 5 = 120
Power(5,2) = 25.0
```

UNIT- 3 EXCEPTION HANDLING AND MULTITHREADING

1.Explain in detail about exception handling(13)

- Information about the error including its type

Definition:

An Exception is an event that occurs during program execution which disrupts the normal flow of a program. It is an object which is thrown at runtime.

Occurrence of any kind of exception in java applications may result in an abrupt termination of the JVM or simply the JVM crashes.

- The state of the program when the error occurred
- Optionally, other custom information

All exceptions and errors extend from a common java.lang.Throwable parent class.

The **Throwable** class is further divided into two classes:

1. **Exceptions** and
2. **Errors.**

Exceptions: Exceptions represents errors in the Java application program, written by the user. Because the error is in the program, exceptions are expected to be handled, either

- Try to recover it if possible
- Minimally, enact a safe and informative shutdown.

Sometimes it also happens that the exception could not be caught and the program may get terminated. Eg. **ArithmeticException**

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Errors: Errors represent internal errors of the Java run-time system which could not be handled easily. Eg. **OutOfMemoryError.**

DIFFERENCE BETWEEN EXCEPTION AND ERROR:

S.No.	Exception	Error
1.	Exceptions can be recovered	Errors cannot be recovered
2.	Exceptions are of type **java.lang.Exception**	Errors are of type **java.lang.Error**
3.	Exceptions can be classified into two types: a) Checked Exceptions b) Unchecked Exceptions	There is no such classification for errors. Errors are always unchecked.
4.	In case of Checked Exceptions, compiler will have knowledge of checked exceptions and force to keep try…catch block. Unchecked Exceptions are not known to compiler because they occur at run time.	In case of Errors, compiler won't have knowledge of errors. Because they happen at run time.
5.	Exceptions are mainly caused by the application itself.	Errors are mostly caused by the environment in which application is running.

6.	Examples: **Checked Exceptions:** SQLException, IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundsException, NullPointerException	Examples: Java.lang.StackOverFlowError, java.lang.OutOfMemoryError

What is exception handling?

Exception Handling is a mechanism to handle runtime errors, such as ClassNotFoundException, IOException, SQLException, RemoteException etc. by taking the necessary actions, so that normal flow of the application can be maintained.

Advantage of using Exceptions:

- Maintains the normal flow of execution of the application.
- Exceptions separate error handling code from regular code.
  - Benefit: Cleaner algorithms, less clutter
- Meaningful Error reporting.
- Exceptions standardize error handling.

JAVA EXCEPTION HANDLING KEYWORDS

Exception handling in java is managed using the following five keywords:

S.No.	Keyword	Description
1	**try**	A block of code that is to be monitored for exception.
2	**catch**	The catch block handles the specific type of exception along with the try block. For each corresponding try block there exists the catch block.
3	**finally**	It specifies the code that must be executed even though exception may or may not occur.
4	**throw**	This keyword is used to explicitly throw specific exception from the program code.
5	**throws**	It specifies the exceptions that can be thrown by a particular method.

➢ try Block:

✓ The java code that might throw an exception is enclosed in try block. It must be used within the method and must be followed by either catch or finally block.

- ✓ If an exception is generated within the try block, the remaining statements in the try block are not executed.

➢ catch Block:

- ✓ Exceptions thrown during execution of the try block can be caught and handled in a catch block.
- ✓ On exit from a catch block, normal execution continues and the finally block is executed.

➢ finally Block:

A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed.

- ✓ Generally finally block is used for freeing resources, cleaning up, closing connections etc.
- ✓ Even though there is any exception in the try block, the statements assured by **finally** block are sure to execute.
- ✓ **Rule:**
  - **For each try block there can be zero or more catch blocks, but only one finally block.**
  - **The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).**

The try-catch-finally structure(Syntax):

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2) {
   // Handle ExceptionType2 exceptions
}
// ...
finally {
   // Code always executed after the
   // try and any catch block
}
```

Rules for try, catch and finally Blocks:

1) Statements that might generate an exception are placed in a try block.
2) Not all statements in the try block will execute; the execution is interrupted if an exception occurs
3) For each try block there can be zero or more catch blocks, but only one finally block.
4) The try block is followed by
   i. one or more catch blocks
   ii. or, if a try block has no catch block, then it must have the finally block
5) A try block must be followed by either at least one catch block or one finally block.
6) A catch block specifies the type of exception it can catch. It contains the code known as exception handler
7) The catch blocks and finally block must always appear in conjunction with a try block.
8) The order of exception handlers in the catch block must be from the most specific exception

Program without Exception handling: (Default exception handler):

```
class Simple
{
 public static void main(String args[])
{
    int data=50/0;

    System.out.println("rest of the code...");
}
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero
As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed.

12.Explain the multiple catch of java with example(13)

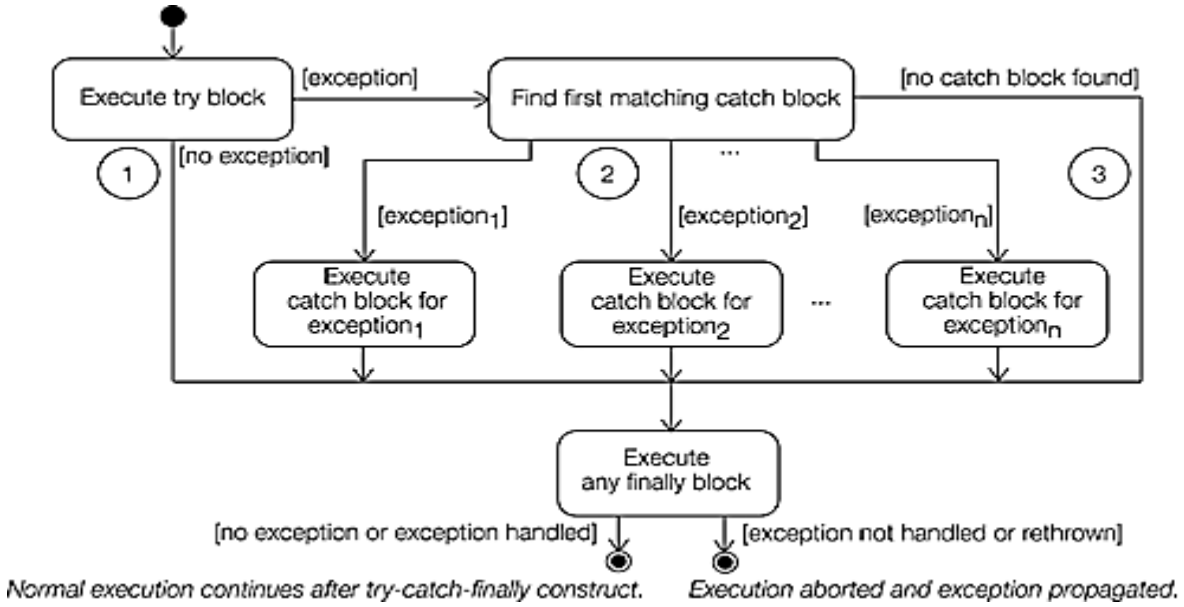
Multiple catch is used to handle many different kind of exceptions that may be generated while running the program. i.e more than one catch clause in a single try block can be used**.**

Rules:

- *At a time only one Exception can occur and at a time only one catch block is executed.*
- *All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception.*

Syntax:

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2) {
   // Handle ExceptionType2 exceptions
}
```



Example:

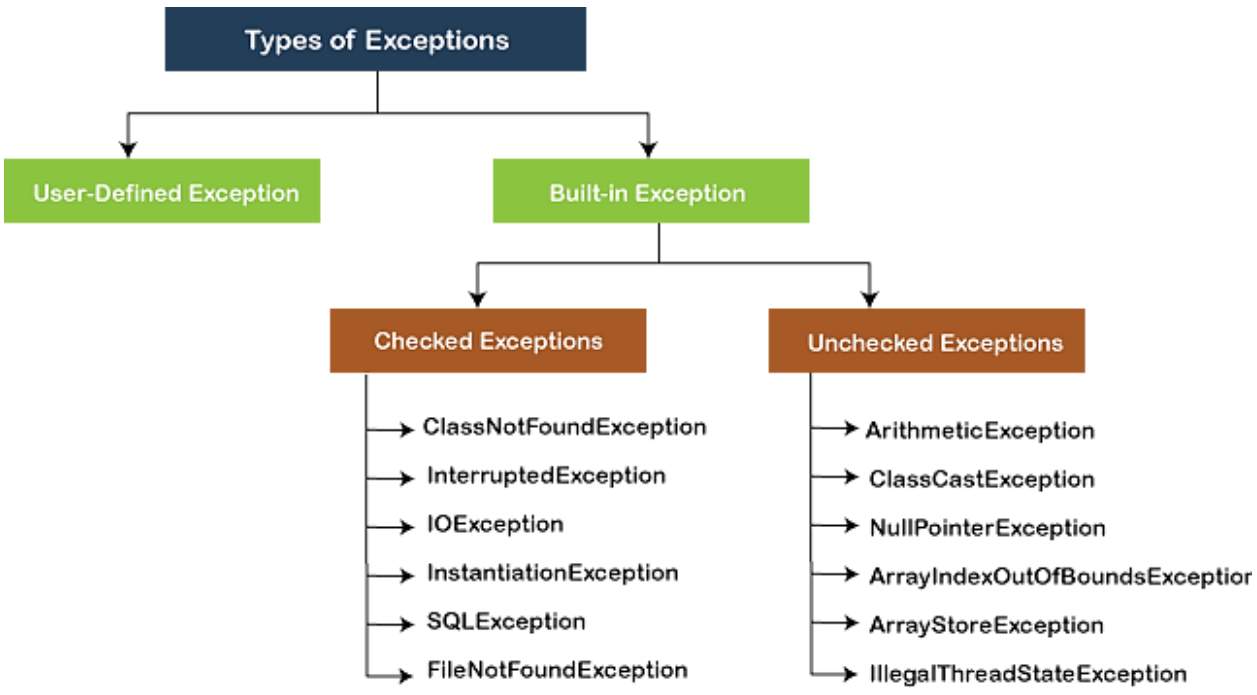
```
public class MultipleCatchBlock2 {

public static void main(String[] args) {

  try
{
    int a[]= {1,5,10,15,16};
    System.out.println("a[1] = "+a[1]);
    System.out.println("a[2]/a[3] = "+a[2]/a[3]);
    System.out.println("a[5] = "+a[5]);
  }
  catch(ArithmeticException e)
 {
```

```
        System.out.println("Arithmetic Exception occurs");
    }
     catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("ArrayIndexOutOfBounds Exception occurs");
    }
     catch(Exception e)
    {
        System.out.println("Parent Exception occurs");
    }
        System.out.println("rest of the code");
}
}
```

Output:

a[1] = 5
a[2]/a[3] = 0
ArrayIndexOutOfBounds Exception occurs
rest of the code

3.Explain the types of exception(13)



: Built-in Exceptions

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

S. No.	Exception	Description
1.	**ArithmeticException**	Thrown when a problem in arithmetic operation is noticed by the JVM.
2.	**ArrayIndexOutOfBoundsException**	Thrown when you access an array with an illegal index.
3.	**ClassNotFoundException**	Thrown when you try to access a class which is not defined
4.	**FileNotFoundException**	Thrown when you try to access a non-existing file.
5.	**IOException**	Thrown when the input-output operation has failed or interrupted.
6.	**InterruptedException**	Thrown when a thread is interrupted when it is processing, waiting or sleeping
7.	**IllegalAccessException**	Thrown when access to a class is denied
8.	**NoSuchFieldException**	Thrown when you try to access any field or variable in a class that does not exist
9.	**NoSuchMethodException**	Thrown when you try to access a non-existing method.
10.	**NullPointerException**	Thrown when you refer the members of a null object
11.	**NumberFormatException**	Thrown when a method is unable to convert a string into a numeric format
12.	**StringIndexOutOfBoundsException**	Thrown when you access a String array with an illegal index.

A. Checked Exceptions:

- ✓ Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- ✓ Checked Exceptions forces programmers to deal with the exception that may be thrown.
- ✓ The compiler ensures whether the programmer handles the exception using try.. catch () block or not. The programmer should have to handle the exception; otherwise, compilation will fail and error will be thrown.

Example:

1. ClassNotFoundException
2. CloneNotSupportedException
3. IllegalAccessException,
4. MalformedURLException.
5. NoSuchFileException
6. NoSuchMethodException
7. IOException

Example Program: (Checked Exception)

`FileNotFoundException` **is a checked exception in Java. Anytime, we want to read a file from filesystem, Java forces us to handle error situation where file may not be present in place.**

Without try-catch

```
import java.io.*;

public class CheckedExceptionExample {
public static void main(String[] args)
{
    FileReader file = new FileReader("src/somefile.txt");
}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Unhandled exception type FileNotFoundException
```

To make program able to compile, you must handle this error situation in `try-catch` block. **Below given code will compile absolutely fine.**

With try-catch

```
import java.io.*;
public class CheckedExceptionExample {
public static void main(String[] args) {
try {
@SuppressWarnings("resource")
FileReader file = new FileReader("src/somefile.java");
System.out.println(file.toString());
}
catch(FileNotFoundException e){
System.out.println("Sorry...Requested resource not availabe...");
} }
}
```

Output:

```
Sorry...Requested resource not availabe...
```

B. Unchecked Exceptions(RunTimeException):

- ✓ The **unchecked** exceptions are just opposite to the **checked** exceptions.
- ✓ Unchecked exceptions are not checked at compile-time rather they are checked at runtime.
- ✓ The compiler doesn't force the programmers to either catch the exception or declare it in a throws clause.
- ✓ In fact, the programmers may not even know that the exception could be thrown.

Example:

1. ArrayIndexOutOfBoundsException
2. ArithmeticException
3. NullPointerException.

Example: Unchecked Exception

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
class Main {
      public static void main(String args[]) {
            int x = 0;
            int y = 10;
            int z = y/x;
      }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:5)
```

Example 1: NullPointer Exception

```
//Java program to demonstrate NullPointerException
class NullPointer_Demo
{
   public static void main(String args[])
   {
     try {
         String a = null; //null value
         System.out.println(a.charAt(0));
      } catch(NullPointerException e) {
         System.out.println("NullPointerException..");
      }
   }
 }
```

Output:

```
NullPointerException..
```

Example 2: NumberFormat Exception

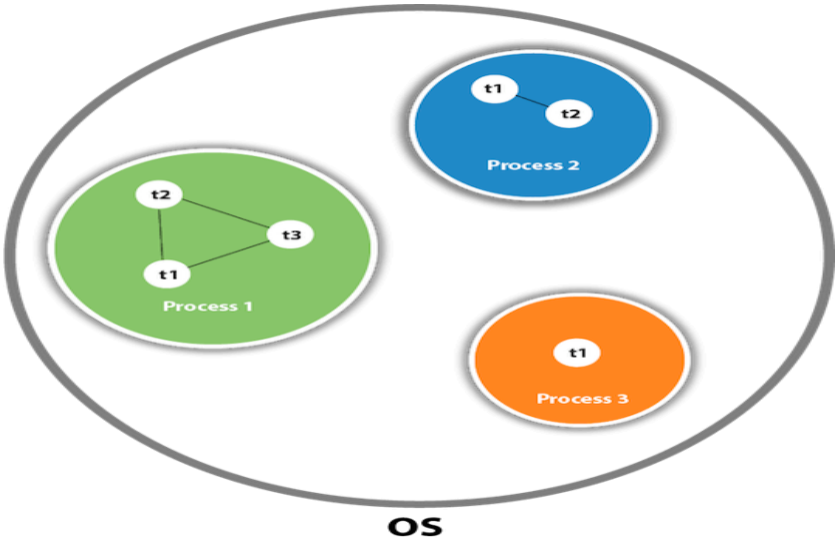
```
// Java program to demonstrate NumberFormatException
class NumberFormat_Demo
{
   public static void main(String args[])
   {
     try {
       // "akki" is not a number
       int num = Integer.parseInt ("akki") ;
       System.out.println(num);
     } catch(NumberFormatException e) {
         System.out.println("Number format exception");
       }
    }
}
```

Output:

```
Number format exception
```

3.Explain multithreading with example(13)

- ✓ Threads are independent.
- ✓ If there occurs exception in one thread, it doesn't affect other threads.
- ✓ It uses a shared memory area.



- ✓ As shown in the above figure, a thread is executed inside the process.
- ✓ There is context-switching between the threads.
- ✓ There can be multiple processes inside the OS, and one process can have multiple threads.

DIFFERENCE BETWEEN THREAD AND PROCESS:

S.NO	PROCESS	THREAD
1)	Process is a heavy weight program	Thread is a light weight process
2)	Each process has a complete set of its own variables	Threads share the same data
3)	Processes must use IPC (Inter-Process Communication) to communicate with sibling processes	Threads can directly communicate with each other with the help of shared variables
4)	Cost of communication between	Cost of communication between

	processes is high.	threads is low.
5)	Process switching uses interface in operating system.	Thread switching does not require calling an operating system.
6)	Processes are independent of one another	Threads are dependent of one another
7)	Each process has its own memory and resources	All threads of a particular process shares the common memory and resources
8)	Creating & destroying processes takes more overhead	Takes less overhead to create and destroy individual threads

: MULTITHREADING

A program can be divided into a number of small processes. Each small process can be addressed as a single thread.

Definition: Multithreading

Multithreading is a technique of executing more than one thread, performing different tasks, simultaneously.

Multithreading enables programs to have more than one execution paths which executes concurrently. Each such execution path is a thread. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

Advantages of Threads / Multithreading:

1. Threads are light weight compared to processes.
2. Threads share the same address space and therefore can share both data and code.
3. Context switching between threads is usually less expensive that between processes.
4. Cost of thread communication is low than inter-process communication.
5. Threads allow different tasks to be performed concurrently.
6. Reduces the computation time.
7. Through multithreading, efficient utilization of system resources can be achieved.

4. .Explain multitasking with example(13)

Definition: Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to maximize the utilization of CPU.

Multitasking can be achieved in two ways:

1) **Process-based Multitasking (Multiprocessing):-**
   - **It is a feature of executing 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 or visiting a web site.

2) **Thread-based Multitasking (Multithreading):-**
   - **It is a feature that a single program can perform two or more tasks simultaneously.**
   - For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

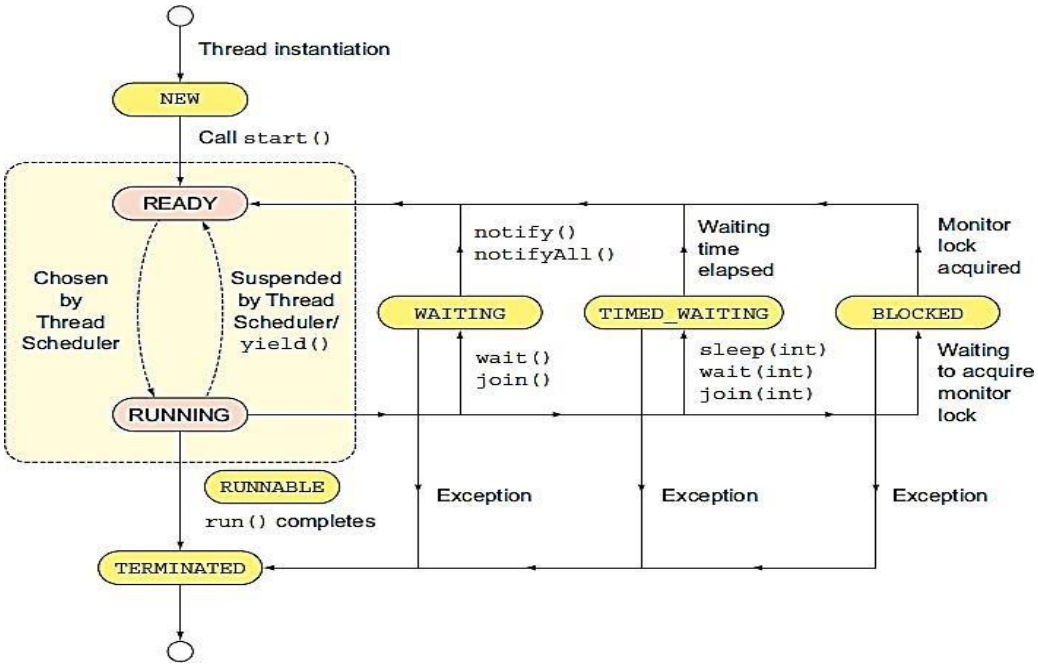
Differences between multi-threading and multitasking

Characteristics	Multithreading	Multitasking
Meaning	A process is divided into several different sub-processes called as threads, which has its own path of execution. This concept is called as multithreading.	The execution of more than one task simultaneously is called as multitasking.
Number of CPU	Can be one or more than one	One
Number of process being executed	Various components of the same process are being executed at a time.	One by one job is being executed at a time.
Number of users	Usually one.	More than one.
Memory Space	Threads are lighter weight. They share the same address space	Processes are heavyweight tasks that require their own separate address spaces.
Communication between units	Interthread communication is inexpensive	Interprocess communication is expensive and limited.
Context Switching	Context switching from one thread to the next is lower in cost.	Context switching from one process to another is also costly.

5.Explain the different states of thread(13)

Different states, a thread (or applet/servlet) travels from its object creation to object removal (garbage collection) is known as life cycle of thread.

1. New State
2. Runnable State
3. Running State
4. Waiting/Timed Waiting/Blocked state
5. Terminated State/ dead state



1. New State:

A new thread begins its life cycle in the new state.

the thread by calling **start()** method, which places the thread in the **runnable** state.

- ✓ A new thread is also referred to as a born thread.
- ✓ When the thread is in this state, only **start()** and **stop()** methods can be called. Calling any other methods causes an **IllegalThreadStateException**.
- ✓ Sample Code: **Thread myThread=new Thread();**

2. Runnable State:

After a newly born thread is started, the thread becomes runnable or running by calling the **run()** method.

- ✓ A thread in this state is considered to be executing its task.
- ✓ Sample code: **myThread.start();**
- ✓ The **start()** method creates the system resources necessary to run the thread,

3. Running state:

- ✓ **Thread scheduler** selects thread to go from runnable to running state. In running state Thread starts executing by entering **run()** method.

- ✓ When threads are in running state, **yield()** method can make thread to go in Runnable state.

4. Waiting/Timed Waiting/Blocked State :

❖ Waiting State:

A runnable thread can be moved to a waiting state by calling the **wait**() method.

- ✓ A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- ✓ A call to **notify()** and **notifyAll()** may bring the thread from waiting state to runnable state.

❖ Timed Waiting:

A runnable thread can enter the timed waiting state for a specified interval of time by calling the **sleep()** method.

- ✓ After the interval gets over, the thread in waiting state enters into the runnable state.
- ✓ Sample Code:

```
try {
Thread.sleep(3*60*1000);// thread sleeps for 3 minutes
}
catch(InterruptedException ex) { }
```

❖ Blocked State:

When a particular thread issues an I/O request, then operating system moves the thread to blocked state until the I/O operations gets completed.

- ✓ This can be achieved by calling **suspend()** method.
- ✓ After the I/O completion, the thread is sent back to the runnable state.

5. Terminated State:

A runnable thread enters the terminated state when,

(i) It completes its task (when the run() method has finished)

```
public void run() { }
```

(ii) Terminates (when the stop() is invoked) – **myThread.stop();**

A terminated thread cannot run again.

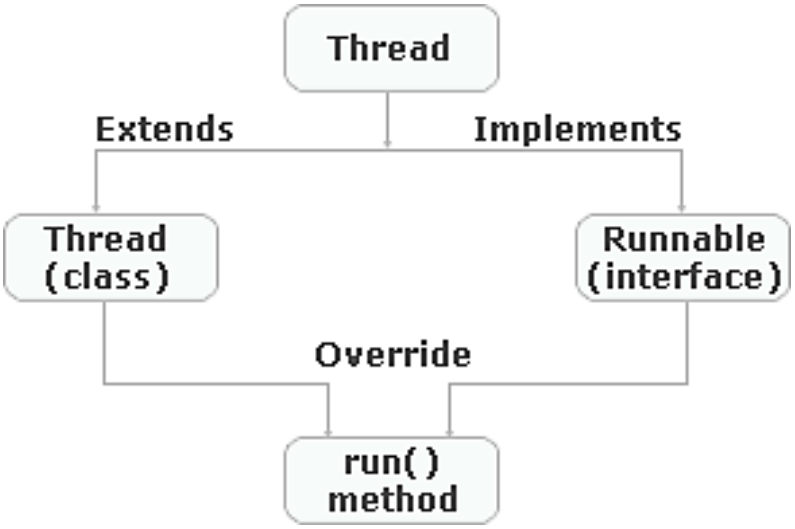
New : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it. **Runnable :** After invocation of start() method on new thread, the thread becomes runnable. **Running :** A thread is in running state if the thread scheduler has selected it. **Waiting :** A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive. **Terminated :** A thread enter the terminated state when it complete its task.

6.Explain how to create threads in java with xample

We can create threads by instantiating an object of type **Thread.**

Java defines two ways to create threads:

1. By implementing **Runnable** interface(**java.lang.Runnable)**
2. By extending the **Thread** class (**java.lang.Thread)**



1. Creating threads by implementing Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed as a thread.Implementing thread program using Runnable is preferable than implementing it by extending Thread class because of the following two reasons:
    1. If a class extends a Thread class, then it cannot extend any other class.
    2. If a class Thread is extended, then all its functionalities get inherited. This is an expensive operation.
- The Runnable interface has only one method that must be overridden by the class

which implements this interface:

public void run()// run() contains the logic of the thread

{

// implementation code

}

- **Steps for thread creation:**

1. Create a class that implements **Runnable** interface. An object of this class is **Runnable** object.

public class MyThread implements Runnable

{

}

2. Override the **run()** method to define the code executed by the thread.
3. Create an object of type Thread by passing a Runnable object as argument.

Thread t=new Thread(Runnable threadobj, String threadName);

4. Invoke the **start()** method on the instance of the Thread class.

t.start();

- **Example:**

```
class MyThread implements Runnable
{

public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
```

```
    }
      }
}
}
public class RunnableDemo {
public static void main(String[] args)
{
MyThread obj=new MyThread();
MyThread obj1=new MyThread(); Thread
t=new Thread(obj,"Thread-1"); t.start();
Thread t1=new Thread(obj1,"Thread-2"); t1.start();
}
}
```

Output:

```
Thread-0  #  Printing  0
Thread-1  #  Printing  0
Thread-1  #  Printing  1
Thread-0  #  Printing  1
Thread-1  #  Printing  2
Thread-0 # Printing 2
```

2. Creating threads by extending Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

All the above constructors creates a new thread.

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.

3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public boolean isAlive():** tests if the thread is alive.
12. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
13. **public void suspend():** is used to suspend the thread(depricated).
14. **public void resume():** is used to resume the suspended thread(depricated).
15. **public void stop():** is used to stop the thread(depricated).
16. **public boolean isDaemon():** tests if the thread is a daemon thread.
17. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
18. **public void interrupt():** interrupts the thread.
19. **public boolean isInterrupted():** tests if the thread has been interrupted.
20. **public static boolean interrupted():** tests if the current thread has been interrupted.

- **Steps for thread creation:**

  1. Create a class that extends **java.lang.Thread** class.

  ```
  public class MyThread extends Thread
  {
  ---
  }
  ```

  2. Override the **run()** method in the sub class to define the code executed by the thread.

  3. Create an object of this sub class.

  MyThread t=new MyThread(String threadName);

  4. Invoke the **start()** method on the instance of the subclass to make the thread for running.

  start();

- **Example:**

```
class SampleThread extends Thread
{
public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
}
}
}
}

public class ThreadDemo {
public static void main(String[] args) {
SampleThread obj=new SampleThread();
obj.start();
SampleThread obj1=new SampleThread();
obj1.start();
}
}
```

Output:

```
Thread-0 # Printing 0
Thread-1 # Printing 0
Thread-1 # Printing 1
Thread-0 # Printing 1
Thread-0 # Printing 2
Thread-1 # Printing 2
```

7.Explain in detail about thread priority(13)

- ✓ Thread priority determines how a thread should be treated with respect to others.
- ✓ Priorities are represented by a number between 1 and 10.
  1 – Minimum Priority5 – Normal Priority 10 – Maximum Priority
- ✓ Thread scheduler will use priorities while allocating processor.
- ✓ The thread which is having highest priority will get the chance first.
- ✓ Higher priority threads get more CPU time than lower priority threads.
- ✓ A higher priority thread can also preempt a lower priority thread.
- ✓ **3 constants defined in Thread class:**
  1. public static int MIN_PRIORITY
  2. public static int NORM_PRIORITY
  3. public static int MAX_PRIORITY
- ✓ Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.
- ✓ To set a thread's priority, use the **setPriority()** method.
- ✓ To obtain the current priority of a thread, use **getPriority()** method.

- ✓ **Example:**

```
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getNam e());
  System.out.println("running thread priority is:"+
                                        Thread.currentThread().getPriority());

 }

 public static void main(String args[]){
  TestMultiPriority1 m1=newTestMultiPriority1();
  TestMultiPriority1 m2=newTestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY); m1.start();
  m2.start();
 }
}
```

Output:

```
running thread name is:Thread-0
running thread priority is:10 running
thread name is:Thread-1 running
```

thread priority is:1

8.Explain in detail about thread synchronization(13)

Thread Synchronization

Thread synchronization is the concurrent execution of two or more threads that share critical resources.

✓ Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

✓ Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

✓ Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data.
This can be done by two ways in java:

1. by synchronized method
2. by synchronized block

✓ Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor.

1. Java synchronized method

✓ If you declare any method as synchronized, it is known as synchronized method.

✓ Synchronized method is used to lock an object for any shared resource.

Syntax to use synchronized method:

```
Access_modifier synchronized return_type
method_name(parameters)
{ …….. }
```

Example of java synchronized method:

```
class Table{
synchronized void printTable(int n)//synchronized method
{
for(int i=1;i<=5;i++) {
    System.out.println(n*i);
    try{    Thread.sleep(400);       }
        catch(Exception e) { System.out.println(e); }
    }
 }
}

class MyThread1 extends Thread {
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}

class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){ t.printTable(100);
}
}
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();     //only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
} }
```

Output:

```
 5
 10
 15
 20
```

25
100
200
300
400
500

2. Synchronized block in java

- ✓ Synchronized block can be used to perform synchronization on any specific resource of the method.
- ✓ Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- ✓ If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

1. **synchronized (object reference expression) {**
2. **//code block**
3. **}**

Example of synchronized block

```
class Table{
 void printTable(int n)
{
  synchronized(this) //synchronized block
  {
    for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{        Thread.sleep(400);          }catch(Exception e){System.out.println(e);}
    }
  }
 }//end of the method
}

class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
                this.t=t;
        }
```

```
        public void run(){
                t.printTable(5);

        }
}

class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(100);
        }
}

public class TestSynchronizedBlock1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj); MyThread2
t2=new MyThread2(obj);

t1.start();
t2.start();
}
}
```

Output:

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

Difference between synchronized method and synchronized block:

Synchronized method	Synchronized block
1. Lock is acquired on whole method. 2. Less preferred. 3. Performance will be less as compared to synchronized block.	1. Lock is acquired on critical block of code only. 2. Preferred. 3. Performance will be better as compared to synchronized method.

9.Explain in detail about Wrappers(13)

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

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

Use of Wrapper classes

- ✓ **Change the value in Method:** Java supports only call by value.
- ✓ **Serialization:** to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- ✓ **Synchronization:** Java synchronization works with objects in Multithreading.
- ✓ **java.util package:** The java.util package provides the utility classes to deal with objects.
- ✓ **Collection Framework:** Java collection framework works with objects only.

Example:

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa public
class WrapperExample3{ public static
void main(String args[]){ byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a'; boolean
b2=true;

//Autoboxing: Converting primitives into objects Byte
byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l; Float
floatobj=f; Double
doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;

//Printing objects

System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);
```

```
//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj; int
intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj; boolean
boolvalue=boolobj;

//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}
}
```

Output

```
---Printing object values--- Byte
object: 10
Short object: 20
Integer object: 30
Long object: 40
```

Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a boolean
value: true

9.Explain in detail about autoboxing(13)

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Example:

```
public class WrapperExample1{ public
static void main(String args[]){
//Converting int into Integer int
a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output

20 20 20

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

Example:

```
//Unboxing example of Integer to int
public class WrapperExample2
{
public static void main(String args[])
{
//Converting Integer to int Integer
a=new Integer(3);
int i=a.intValue();              //converting Integer to int explicitly
int j=a;                         //unboxing, now compiler will write a.intValue() internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output

3 3 3

10.Write a java program for finding factorial(using recursion) prints the stack trace of a recursive factorial function.

```
import java.util.Scanner; public

class StackTraceTest
{
   public static int factorial(int n)
   {
      System.out.println(" Factorial ("+n+"):"); Throwable
      t=new Throwable(); StackTraceElement[]
      frames=t.getStackTrace(); for(StackTraceElement
      f:frames)
      {
         System.out.println(f);
      }
      int r;
      if(n<=1)
         r=1;
      else
         r=n*factorial(n-1);
      System.out.println("return "+r);
```

```
        return r;
    }

    public static void main(String[] args)
    {
        Scanner in=new Scanner(System.in);
        System.out.println("Enter n: ");
        int n=in.nextInt();
        factorial(n);
    }
}
```

Output:

Enter n:

3

Factorial (3):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.main(StackTraceTest.java:30)
Factorial (2):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30)
Factorial (1):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30) **return**
1

return 2

return 6

Unit IV

1.Explai in detail about Generic Programming(13)

. Generic Programming

- Generic programming means to write code that can be reused for objects of many different types.

Generic Array Lists

- Array is fixed size at compile time.
- It is a problem for the programmers, because programmers wants to change the array at run time.
- In Java, the situation is much better when compared with other languages.
- You can set the size of an array at runtime.

```
int actualSize = . . .;
Employee[] staff = new Employee[actualSize];
```

- This code does not completely solve the problem of dynamically modifying arrays at runtime.
- Once you set the array size, you cannot change it easily.
- Instead, theeasiest way in Java to deal with this common situation is to use another Java class
- The ArrayList class is similar to an array, but it automatically adjusts its capacityas you add and remove elements, without your needing to write any code.
- Here we declare and construct an array list that holds Employee objects:

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

- You use the add method to add new elements to an array list.

```
staff.add(new Employee("Harry Hacker", . . .));
```

- The array list manages an internal array of object references.
- If you know the size then you can do it in two ways

```
staff.ensureCapacity(100);
```

- You can also pass an initial capacity to the ArrayList constructor:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```

- The size method returns the actual number of elements in the array list.

```
staff.size();
```

Example

```
import java.util.ArrayList;
class Employee{
   private String empName;
   private float salary;
   public Employee(String n,float s){
      empName=n;
      salary=s;
   }
   public String toString(){
      return("Name of the class is " +this.getClass()+" [empName : "+empName+" salary : "+salary+"]");
   }
```

```
}
public class DemoArrayList {
  public static void main(String s[]){
     Employee e;
     ArrayList staff = new ArrayList(10);
      staff.add(new Employee("Carl Cracker", 75000));
      staff.add(new Employee("Harry Hacker", 50000));
      staff.add(new Employee("Tony Tester", 40000));
     for (int i=0;i<staff.size();i++){
         e=(Employee) staff.get(i);
         System.out.println(e.toString()+"\n");
     }
  }
}
```

2. **Explai in detail about Generic class(13)**

- A generic class is a class with one or more type variables.
- Consider the example given below; we use a simple Pair class as an example.
- This class allows us to focus on generics.

```
public class Pair<T>
{
        public Pair() {
                first = null; second = null; }
        public Pair(T first, T second) {
                this.first = first; this.second = second; }
        public T getFirst() {
                return first; }
        public T getSecond() {
                return second; }
        public void setFirst(T newValue) {
                first = newValue; }
        public void setSecond(T newValue) {
                 second = newValue; }
        private T first;
        private T second;
}
```

- The Pair class introduces a type variable T, enclosed in angle brackets <>, after the class name.
- A generic class can have more than one type variable.
- For example, we could have defined the Pair class with separate types for the first and second field:

```
c class Pair<T, U> { . . . }
```

- The type variables are used throughout the class definition to specify method return types and the types of fields and local variables.
- For example:

```
private T first; // uses type variable
```

- You instantiate the generic type by substituting types for the type variables, such as

Pair<String>

- You can think of the result as an ordinary class with constructors

  Pair<String>()

  Pair<String>(String, String)

- and methods

  String getFirst()
  String getSecond()
  void setFirst(String)
  void setSecond(String)

Example

```
class DemoGeneric{
   public static void main(String s[]){
      Pair <String> obj1= new Pair("HELLO","WELCOME");
      System.out.println(obj1.getFirst());
      System.out.println(obj1.getSecond());
      Pair <Integer> obj2=new Pair(10,20);
      System.out.println(obj2.getFirst());
      System.out.println(obj2.getSecond());
   }
}
class Pair<T>
{
   public Pair() {
      first = null; second = null;
   }
   public Pair(T first, T second) {
      this.first = first;
      this.second = second;
   }
   public T getFirst() {
      return first;
   }
   public T getSecond() {
      return second;
   }
   public void setFirst(T newValue)
   {
      first = newValue;
   }
   public void setSecond(T newValue) {
      second = newValue;
   }
   private T first;
   private T second;
}
```

The output of the above program would be

```
HELLO
WELCOME
10
20
```

3. Explai in detail about Generic method(13)

- define a single method with type parameters.

```
class ArrayAlg
{
        public static <T> T getMiddle(T[] a)
        {
        return a[a.length / 2];
        }
}
```

- This method is defined inside an ordinary class, not inside a generic class.
- However, it is a generic method, as you can see from the angle brackets and the type variable.
- Note that the type variables are inserted after the modifiers (public static, in our case) and before the return type.
- You can define generic methods both inside ordinary classes and inside generic classes.
- When you call a generic method, you can place the actual types, enclosed in angle brackets, before the method name:

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.<String>getMiddle(names);
```

- In this case you can omit the <String> type parameter from the method call.
- The compiler has enough information to infer the method that you want.
- It matches the type of names (that is, String[]) against the generic type T[] and deduces that T must be String.
- That is, you can simply call

```
String middle = ArrayAlg.getMiddle(names);
```

- In almost all cases, type inference for generic methods works smoothly.
- Occasionally, the compiler gets it wrong, and you'll need to decode an error report.
- Consider this example:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

- The error message is: "found: java.lang.Number&java.lang.Comparable<? extends java.lang.Number&
- java.lang.Comparable<?>>, required: double".
- the compiler autoboxed the parameters into a Double and two Integer objects, and then it tried to find a common supertype of these classes.
- It actually found two: Number and the Comparable interface, which is itself a generic type. In this case, the remedy is to write all parameters as double values.

Example

```
class DemoGeneric1{
   public static void main(String s[]){
      Method <Integer,String>obj1 =new Method(10,"Hello");
      System.out.println(obj1.getFirst());
      System.out.println(obj1.getSecond());
   }
}
class Method <T,U>
{
   public Method(T first, U second) {
      this.first = first;
      this.second = second;
```

```
    }
    public T getFirst(){
      return(first);
    }
     public U getSecond(){
      return(second);
    }
    private T first;
    private U second;
}
```

Output of the above program would be

10
Hello

4.Explain in detail about Bounds for Type Variables(13)

Bounds for Type Variables

- Sometimes, a class or a method needs to place restrictions on type variables.

```
class ArrayAlg
{
        public static <T> T min(T[] a) // almost correct
        {
                if (a == null || a.length == 0)
                        return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
                if (smallest.compareTo(a[i]) > 0)
                        smallest = a[i];
        return smallest;
        }
```

- The variable smallest has type T, which means that it could be an object of an arbitrary class.
- How do we know that the class to which T belongs has a compareTo method?
- The solution is to restrict T to a class that implements the Comparable interface—a standard interface with a single method, compareTo.
- You achieve this by giving a bound for the type variable T:

```
public static <T extends Comparable> T min(T[] a) . . .
```

- the Comparable interface is itself a generic type.
- For now, we will ignore that complexityand the warnings that the compiler generates.
- Now, the generic min method can only be called with arrays of classes that implement theComparable interface, such as String, Date, and so on.
- Calling min with a Rectangle array is acompile-time error because the Rectangle class does not implement Comparable.
- You may wonder why you use the extends keyword rather than the implements keyword inthis situation—after all, Comparable is an interface.

- The notation<T extends BoundingType>expresses that T should be a subtype of the bounding type.
- Both T and the bounding typecan be either a class or an interface.
- The extends keyword was chosen because it is a reasonableapproximation of the subtype concept, and the Java designers did not want toadd a new keyword (such as sub) to the language.
- A type variable or wildcard can have multiple bounds.
- For example:

  T extends Comparable & Serializable
- The bounding types are separated by ampersands (&) because commas are used to separatetype variables.
- As with Java inheritance, you can have as many interface supertypes as you like, but atmost one of the bounds can be a class.
- If you have a class as a bound, it must be the firstone in the bounds list.
- Themethod computes the minimum and maximum of a generic array, returning a Pair<T>.

public int compareTo(T o)

- Method of comparable interface.
- Comparable interface itself a generic interface.
- Compares this object with the specified object for order.
- Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Program

```
import java.util.*;
public class PairTest2
{
   public static void main(String[] args)
   {
      GregorianCalendar[] birthdays ={
         new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
         new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
         new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
         new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
      };
      Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
      System.out.println("min = " + mm.getFirst().getTime());
      System.out.println("max = " + mm.getSecond().getTime());
   }
}
```

5.Explain in detail about Reflection and Generics

Reflection and Generics

- The Class class is now generic.

- For example, String.class is actually an object (in fact, thesole object) of the class Class<String>.
- The type parameter is useful because it allows the methods of Class<T> to be more specificabout their return types.
- The following methods of Class<T> take advantage of thetype parameter:

  T newInstance()

  T cast(Object obj)

  T[] getEnumConstants()

  Class<? super T> getSuperclass()

  Constructor<T> getConstructor(Class... parameterTypes)

  Constructor<T> getDeclaredConstructor(Class... parameterTypes)
- The newInstance method returns an instance of the class, obtained from the default constructor.
- Its return type can now be declared to be T, the same type as the class that isbeing described by Class<T>. That saves a cast.
- The cast method returns the given object
- The getEnumConstants method returns null if this class is not an enum
- Finally, the getConstructor and getDeclaredConstructor methods return a Constructor<T> object.

Example

```
import java.lang.reflect.*;
import java.util.*;
public class Foo {
   ArrayList<Integer> genericList;

   public static void main(String[] args) throws Exception {
      // get the basic information
      Field field = Foo.class.getDeclaredField("genericList");
      ParameterizedType ptype = (ParameterizedType) field.getGenericType();
      Class rclas = (Class) ptype.getRawType();
      System.out.println("rawType is class " + rclas.getName());

      // list the type variables of the base class
      TypeVariable[] tvars = rclas.getTypeParameters();
      for (int i = 0; i < tvars.length; i++) {
         TypeVariable tvar = tvars[i];
         System.out.print(" Type variable " + tvar.getName() + " with upper bounds [");
         Type[] btypes = tvar.getBounds();
         for (int j = 0; j < btypes.length; j++) {
            if (j > 0) {
               System.out.print("");
            }
            System.out.print(btypes[j]);
         }
         System.out.println("]");
      }

      // list the actual type arguments
      Type[] targs = ptype.getActualTypeArguments();
      System.out.print("Actual type arguments are\n (");
```

```
        for (int j = 0; j < targs.length; j++) {
            if (j > 0) {
                System.out.print(" ");
            }
            Class tclas = (Class) targs[j];
            System.out.print(tclas.getName());
        }
        System.out.print(")");
    }
}
```

The output of the above program would be

```
rawType is class java.util.ArrayList
 Type variable E with upper bounds [class java.lang.Object]
Actual type arguments are
 (java.lang.Integer)
```

Restrictions and Limitations

- ❖ Type Parameters Cannot Be Instantiated with Primitive Types
- ❖ Runtime Type Inquiry Only Works with Raw Types
- ❖ You Cannot Throw or Catch Instances of a Generic Class
- ❖ Arrays of Parameterized Types Are Not Legal
- ❖ You Cannot Instantiate Type Variables
- ❖ Type Variables Are Not Valid in Static Contexts of Generic Classes
- ❖ Beware of Clashes After Erasure

Unit v

1. Explain in detail about Event handling(13)

Changing the state of an object is known as an event.

For example, click on button, dragging mouse etc.

The java.awt.event package provides many event classes and Listener interfaces for event handling.

Event handling has three main components,

- **Events** : An event is a change in state of an object.
- **Events Source** : Event source is an object that generates an event.
    - **Listeners** : A listener is an object that listens to the event
    - . Events are supported by a number of Java packages, like **java.util**, **java.awt** and **java.awt.event**.

Important Event Classes and Interface

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener

ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
focusEvent	generated when component gains or loses keyboard focus	FocusListener

Steps to handle events:

- Implement appropriate interface in the class.
- Register the component with the listener.

How to implement Listener

1. Declare an event handler class and specify that the class either implements an ActionListener(any listener) interface or extends a class that implements an ActionListener interface. For example:

```
public class MyClass implements ActionListener
{
// Set of Code
}
```

2. Register an instance of the event handler class as a listener on one or more components. For example:

```
someComponent.addActionListener(instanceOfMyClass);
```

3. Include code that implements the methods in listener interface. For example:

```
public void actionPerformed(ActionEvent e) {
    //code that reacts to the action
}
```

```
public class Mouse implements MouseListener {
```

```
        TextArea s;
public Mouse()
{
        Frame d=new
        Frame("kkkk"); s=new
        TextArea(""); d.add(s);
        s.addMouseListener(this);
        d.setSize(190, 190);
    d.show();
    }
    public void mousePressed(MouseEvent e) {
      System.out.println("MousePressed");
      int a=e.getX();
        int b=e.getY();
        System.out.println("X="+a+"Y="+b);
    }
    public void mouseReleased(MouseEvent e) {
        System.out.println("MouseReleased");
        }
    public void mouseEntered(MouseEvent e) {
        System.out.println("MouseEntered");
    }
    public void mouseExited(MouseEvent e) {
        System.out.println("MouseExited");
    }
    public void mouseClicked(MouseEvent e) {
        System.out.println("MouseClicked");
       }
    public static void main(String arg[])
    {
```

```
        Mouse a=new Mouse();
        }
}
```

Mouse Motion Listener

```
package Listener;
import java.awt.event.MouseEvent;
import
java.awt.event.MouseMotionListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
public class MouseMotionEventDemo extends JPanel implements
MouseMotionListener { MouseMotionEventDemo()
{
            extArea a=new JTextArea();
            a.addMouseMotionListener(thi
            s); JFrame b=new JFrame();
            b.add(a);
            b.setVisible(true);
}
public void mouseMoved(MouseEvent e) {
System.out.println("Mouse is Moving");
}
    public void mouseDragged(MouseEvent e) {
            System.out.println("MouseDragged");
}
public static void main(String arg[])
{
```

```
        MouseMotionEventDemo a=new MouseMotionEventDemo();
}
}
```

2.Write a program for WindowStateListener and ActionStateListener(13)

WindowStateListener

```
package Listener;
import java.awt.event.MouseEvent;
import
java.awt.event.MouseMotionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowStateListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
public class window2 extends JPanel implements
WindowStateListener { window2()
{
        JFrame b=new JFrame();
        b.addWindowStateListener(this);
        b.setVisible(true);
}
public static void main(String arg[])
```

```
{
        window2 b=new window2();
}
                    public void
    windowStateChanged(WindowEvent e) {
      // TODO Auto-generated method stub
       System.out.println("State Changed");
}}
```

ACTION LISTENER

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
  public class A extends JFrame implements
       ActionListener { Scientific() {
          JPanel buttonpanel = new JPanel();
           JButton b1 = new JButton("Hai");
           buttonpanel.add(b1);
           b1.addActionListener(this);
       }
       public void actionPerformed(ActionEvent e)
           { System.out.println(“Hai button”);
        }
        public static void main(String
             args[]) { A f = new A();
           f.setTitle("ActionListener");
           f.setSize(500,500);
           f.setVisible(true);
       }}
```

3.Write aprogram for window adapter and mouse adapter(13)

Java WindowAdapter Example

```
import java.awt.*;
import
java.awt.event.*;
public class
  AdapterExample{
  Frame f;
  AdapterExample(){
    f=new Frame("Window Adapter");
    f.addWindowListener(new
    WindowAdapter(){
      public void
        windowClosing(WindowEvent e) {
        f.dispose();
      }
    });
     f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
```

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

Java MouseAdapter Example

```
import java.awt.*;
import java.awt.event.*;
  public class MouseAdapterExample extends
    MouseAdapter{ Frame f;
    MouseAdapterExample(){
      f=new Frame("Mouse
      Adapter");
      f.addMouseListener(this);
      f.setSize(300,300);
      f.setLayout(null);
      f.setVisible(true);
    }
    public void mouseClicked(MouseEvent
      e) { Graphics g=f.getGraphics();
      g.setColor(Color.BLUE);
      g.fillOval(e.getX(),e.getY(),30,30);
    }
     public static void main(String[] args) {
    new MouseAdapterExample();
} }
```

4.Explain RadioButtons with example(13)

- The JRadioButton class is used to create a radio button.
- It is used to choose one option from multiple options.
- It is widely used in exam systems or quiz. It should be added in ButtonGroup to select one radio button only.

```
import javax.swing.*;
import java.awt.event.*;
class RadioButtonExample extends JFrame implements
ActionListener{ JRadioButton rb1,rb2;
JButton b;
RadioButtonExample(){
rb1=new
JRadioButton("Male");
rb1.setBounds(100,50,100,30);
rb2=new JRadioButton("Female");
rb2.setBounds(100,100,100,30);
ButtonGroup bg=new ButtonGroup();
bg.add(rb1);bg.add(rb2);
b=new JButton("click");
b.setBounds(100,150,80,30);
b.addActionListener(this);
add(rb1);add(rb2);add(b);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
```

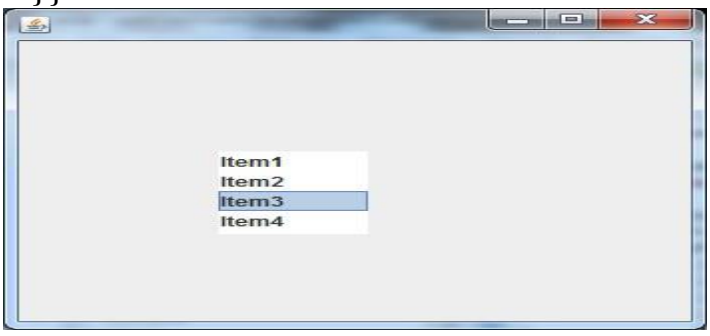
```
if(rb1.isSelected()){
JOptionPane.showMessageDialog(this,"You are
Male.");
}
if(rb2.isSelected()){
JOptionPane.showMessageDialog(this,"You are
Female.");
}       }
public static void main(String args[]){
new RadioButtonExample();
}}
```

5.Explain List with example(13)

- The object of JList class represents a list of text items.
- It inherits JComponent class

```
import javax.swing.*;
public class ListExample
{
        ListExample(){
          JFrame f= new JFrame();
          DefaultListModel<String> l1 = new
           DefaultListModel<>(); l1.addElement("Item1");
           l1.addElement("Item2");
           l1.addElement("Item3");
           l1.addElement("Item4");
           JList<String> list = new
           JList<>(l1);
           list.setBounds(100,100, 75,75);
           f.add(list);
```

```
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String args[])
    {
    new ListExample();
    }}
```



6.Explain DialogBoxes with example(13)

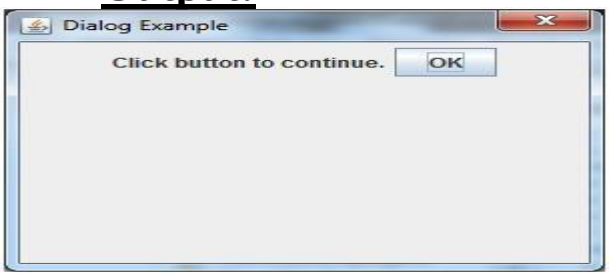
- The JDialog control represents a top level window with a border and a title used to take some form of input from the user.
- It inherits the Dialog class.Unlike JFrame,
- it doesn't have maximize and minimize buttons.

Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
    public class DialogExample
    { private static JDialog d;
    DialogExample() {
        JFrame f= new JFrame();
```

```
    d = new JDialog(f , "Dialog Example", true);
    d.setLayout( new FlowLayout() );
    JButton b = new JButton ("OK");
    b.addActionListener ( new ActionListener()
    {
       public void actionPerformed( ActionEvent e )
       {
          DialogExample.d.setVisible(false); }}};
    d.add( new JLabel ("Click button to
    continue.")); d.add(b);
    d.setSize(300,300
    );
    d.setVisible(true)
    ;
}
public static void main(String args[])
{
    new DialogExample(); }}
```

Output:



7.Explain Windows with example(13)

The class JWindow is a container that can be displayed but does not have the title bar

JMenu Bar class is used to display menubar on the window or frame. It may have several menus. The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.

The object of JMenuItem class adds a simple labeled menu item. The items used in a menu must belong to the JMenuItem or any of its subclass.

```
import javax.swing.*;
class MenuExample
{
    JMenu menu, submenu;
    JMenuItem i1, i2, i3, i4,
    i5; MenuExample(){
    JFrame f= new JFrame("Menu and MenuItem
    Example"); JMenuBar mb=new JMenuBar();
    menu=new JMenu("Menu");
    submenu=new JMenu("Sub
    Menu"); i1=new
    JMenuItem("Item 1");
    i2=new JMenuItem("Item 2");
    i3=new JMenuItem("Item 3");
    i4=new JMenuItem("Item 4");
    i5=new JMenuItem("Item 5");
    menu.add(i1); menu.add(i2);
    menu.add(i3); submenu.add(i4);
    submenu.add(i5);
```

```
        menu.add(submenu);
        mb.add(menu);
        f.setJMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
    new MenuExample();
    }}
```