Savitribai Phule Pune University
Semester IV (Computer Engineering)

Principles of Programming Languages

Unit III : Java as Object Oriented Programming Language-Overview

Q.1 Explain why Java is Secured, Robust, Portable, and Dynamic? Which of the concepts in Java ensures these?

SPPU : Dec. 17, May 18, May 19, 6 Marks

Ans. : Java Features

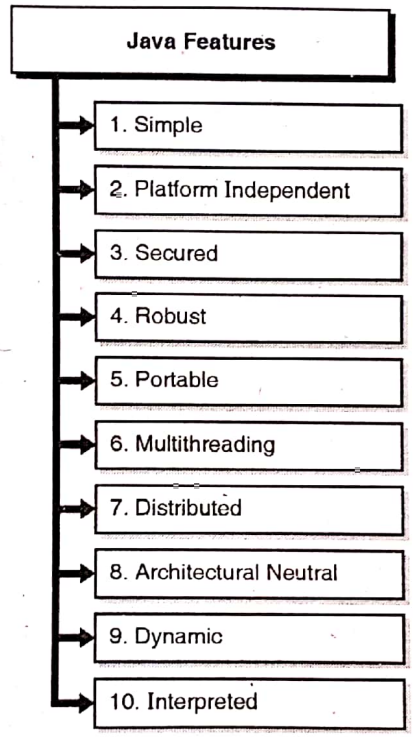

Fig. 3.1 : Java Features

1. **Simple**

   Syntax of language is very simple to use as compared to other programming languages.

2. **Platform Independent**

   - It is operating system and hardware independent. It is possible due to java bytecode. Bytecode is universally accepted code by any JVM implementation.
   - Though JVM is dependent, bytecode is platform independent and because of which Java can be considered as platform independent language.

3. **Secured**

   - It is most secured language because it does not provide the pointers. Security is added at different layers.
   - It is added during the programming in terms of methods and private variables. JVM verifies the bytecode means another layer of security is added.

4. **Robust**
   - It means strong. It uses very efficient memory management. It does not provide pointers so avoids security problems.
   - Automatic garbage collection is supported by java.

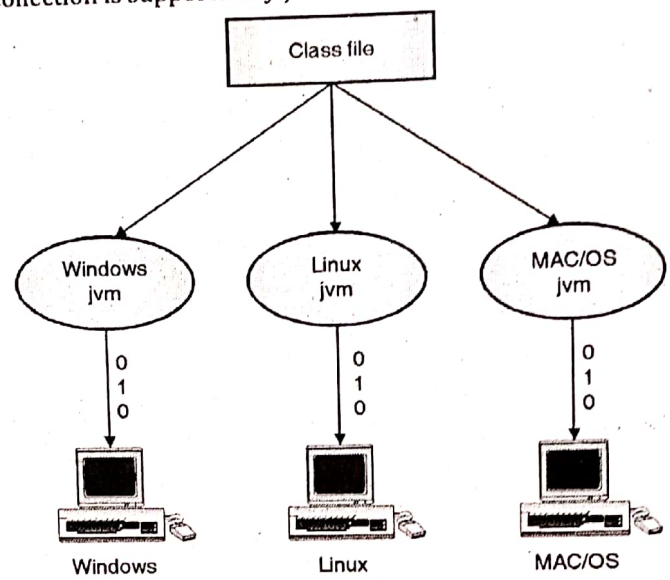

Fig. 3.2 : Platform independent

5. **Portable :** Java application can be installed on any device without any change.
6. **Multithreading**
   - To utilize the processor power, multithreading is supported by Java.
   - Separate library is provided to multithread the program.
7. **Distributed**
   - Distributed application development is possible using java. RMI and EJB can be used for it.
   - File access through internet is possible.
8. **Architecture Neutral :** All programs can run on any platform and on any operating system.
9. **Dynamic :** Memory allocation to object of class is dynamic.
10. **Interpreted**
    - Java byte code is translated on the fly to native machine instructions and is not stored anywhere.
    - The development process is more rapid and analytical since the linking is an incremental and light weight process.

Q. 2 Explain Java Virtual Machine(JVM) Architecture in details. Explain each component briefly. **(6 Marks)**

Ans. : Java Virtual Machine

- Java Virtual Machine or simply JVM is the engine that drives the Java code. Mostly in other Programming Languages, compiler produce a compiled code for a particular system but Java compiler produce Bytecode for a

- When we compile any Java program, then bytecode is generated. In java, bytecode is the source code that can be used to run on any platform. Bytecode is an intermediary language between Java source and the system. In this way, bytecode which gets interpreted on a different machine and hence it makes it Platform or Operating system independent.
- Java is called platform or operating system independent because of Java Virtual Machine. As different computers with the different operating system have their different JVM, when we submit a .class file to any operating system, JVM interprets the bytecode into machine level language.
- JVM is the essential component of Java architecture, and it is the part of the JRE (Java Runtime Environment). In particular, a program of JVM is written in C Programming Language, and JVM is Operating System dependent. JVM is responsible for allocating the necessary memory needed by the Java program.

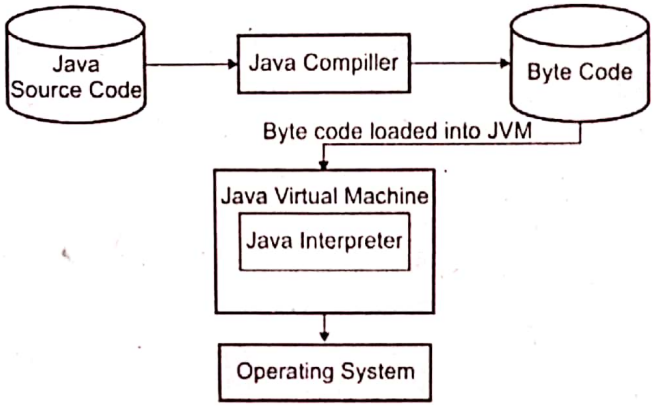

Fig. 3.3 : A JVM

- JVM is responsible for de-allocating memory spaces.

JVM Architecture

- Let's understand the Architecture of JVM. It contains :
  - Class loader
  - Memory area
  - Execution engine

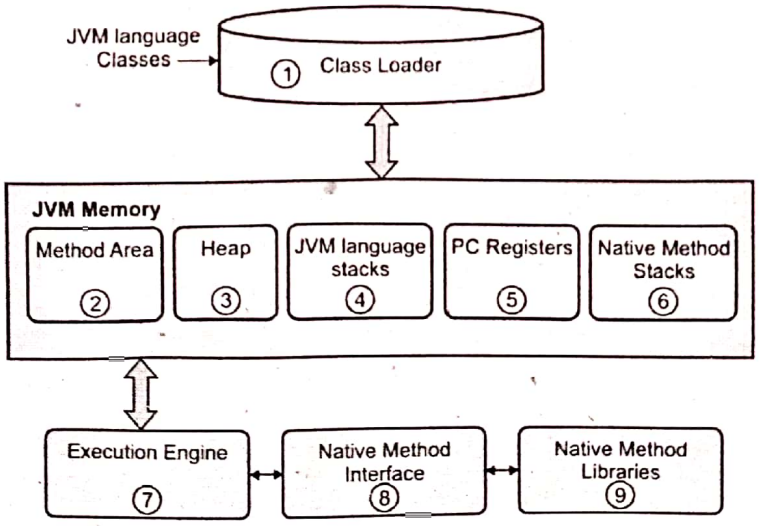

Fig. 3.4 : JVM architecture

1. **Class Loader**

   The class loader is used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

2. **Method Area**

   JVM Method Area stores class structures like the code for methods, metadata, and the constant runtime pool.

3. **Heap**

   All the java objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.

4. **JVM language Stacks**

   Java language stacks store local variables and its partial results. Each thread has its own local JVM stack, created simultaneously as the thread is created. Whenever a method is invoked, a new frame is created and it is deleted when method invocation process is complete.

5. **PC Registers**

   PC register store the address of the currently executing JVM instruction. In Java, each thread has its separate PC register.

6. **Native Method Stacks**

   Native method stacks hold the instruction of native code that depends on the native library. It is written in another language instead of Java.

7. **Execution Engine**

   It is a type of software used to test software, hardware, or complete systems. The test execution engine never carries any information about the any tested product.

8. **Native Method Interface**

   It is a programming framework that allows Java code which is running in a JVM to call by libraries and native applications.

9. **Native Method Libraries**

   Native Libraries are the collection of the Native Libraries (C, C++) which are needed by the Execution Engine.

Q. 3 What are the primitive data types? List the primitive data types in Java and their respective storage capacity.

Ans. : Primitive Types **SPPU : May 17, 6 Marks**

Primitive data types are also called simple data types. There are eight primitive data types in java : **byte, short, int, long, char, float, double and Boolean**. These are classified into following categories :

1. **Integers :** This group includes **byte, short, int** and **long,** which are for whole-valued signed numbers.

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

2. Floating-point numbers

This group includes **float** and **double**, which represent numbers with fractional precision.

Name	Size (Bits)	Range
float	32	1.4e – 045 to 3.4e + 038
double	64	4.9e – 324 to 1.8e + 308

3. Characters :

This group includes **char**, which represents symbols in a character set, like letters and numbers.

4. Boolean

- This group includes Boolean, which is a special type for representing **true** and **false** values.
- You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

Q. 4 What are Strings in Java? Explain following operations of class Strings in Java with example.

1. To find length of the string
2. To compare two strings
3. Extraction of a character from string .

SPPU : May 19, 6 Marks

Ans. :

String Handling : String Class Methods

- In java everything is class. String is also class. String class provides many methods to manipulate the string. String is object which represents characters sequence. Java implements string as object and not as array.
- String class has many methods like compare string, length of the string, getting character, etc. String object can be constructed using many different methods. After creating string object, characters within string cannot be changed. This property of string is called immutable. Which means string is immutable objects.
- If programmer wanted to modify the string during run time then StringBuffer can be used. String class is part of java.lang. Whenever we display some message using System.out.print() function, string object is created internally.
- System.out.print("String example");
- In above statement string object is created for "String example". String is created using many methods. Simple method is given as follows :
- String str="This is example string";
- String is displayed using System.out.print() functions. Two strings can be concatenated using + operator.
- Program 3.5 gives simple string example.

String Constructor

String provides almost thirteen constructors which are as shown in Fig. 3.5.

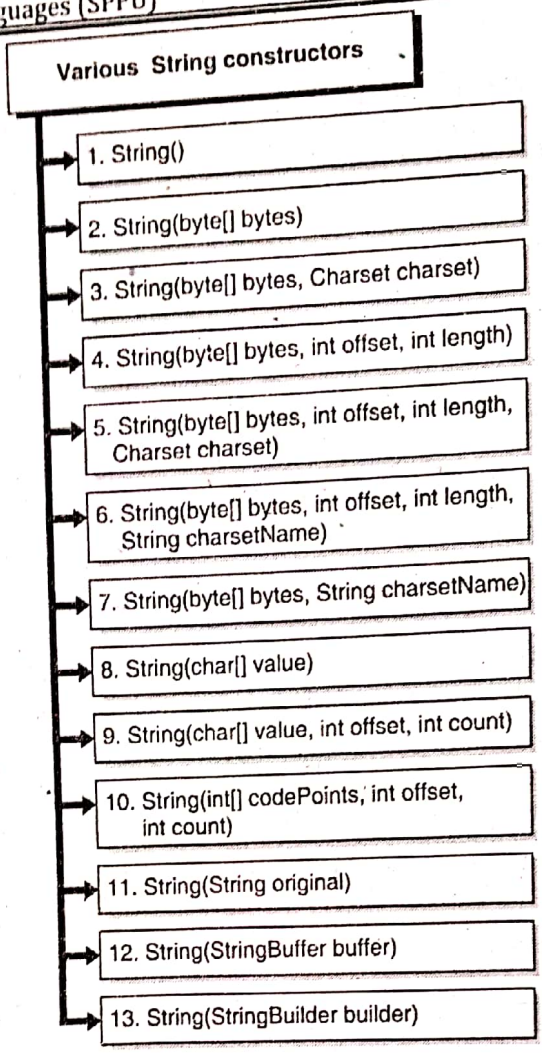

Fig. 3.5 : String constructor

1. **String()**

   It initializes created String object to empty characters.

2. **String(byte[] bytes)**

   This constructs a new String by decoding the specified array of bytes.

3. **String(byte[] bytes, charset charset)**

   This constructs a new String by decoding the specified array of bytes.

4. **String(byte[] bytes, int offset, int length)**

   This constructs a new String by decoding the specified subarray of bytes.

5. **String(byte[] bytes, int offset, int length, charset charset)**

   This constructs a new String by decoding the specified subarray of bytes.

6. **String(byte[] bytes, int offset, int length, String charsetName)**

   This constructs a new String by decoding the specified subarray of bytes.

7. **String(byte[] bytes, String charsetName)**

   This constructs a new String by decoding the specified array of bytes.

8. **String(char[] value)**

   This allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

9. **String(char[] value, int offset, int count)**

   This allocates a new String that contains characters from a subarray of the character array argument.

10. **String(int[] codePoints, int offset, int count)**

    This allocates a new String that contains characters from a subarray of the Unicode code point array argument.

11. **String(String original)**

    This initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

12. **String(String Buffer buffer)**

    This allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

13. **String(String Builder builder)**

    This allocates a new string that contains the sequence of characters currently contained in the string builder argument.

String Operations and Character Extraction

- Different operations are possible with string.
- One of the famous operations is concatenation.
- Two strings can be concatenated.
- Concatenation of string, integer, characters, double, float and bytes are also possible. It is possible with + operator. Here + operator acts as concatenation and not addition.
- It means in string a + b is concatenation of number and not addition.
- If addition is needed we have to write (a + b).

simple program to string operations

```
public class StringOperations
{
    public static void main(String[] args)
    {
        System.out.println("Operations on string in Java");
        String str ="Original string";
        String str1 ="another string";
        System.out.println("Length of string is: " +str.length());
        System.out.println("Length of abcd is: "+"abcd".length());
        System.out.println("Fifth position character in  "+str+" is: "+str.charAt(4));
        char buf[]= newchar[5];
        str.getChars(2, 7, buf, 0);
        System.out.println("Range of characters : ");
        System.out.println(buf);
```

```
        char arr[] = newchar[25];
        System.out.print("String using character array ");
        System.out.println(str.toCharArray());
    }
}
```

Output :

```
Operations on string in Java
Length of string is: 15
Length of abcd is: 4
Fifth position character in Original string is: i
Range of characters :
igina
String using character array
Original string
```

String Comparison

- Comparison of two string is possible with equals() and equalsIgnoreCase() method.
- Case sensitive comparison is possible with equals() method whereas equalsIgnoreCase() compares two string in case insensitive manner. But there is difference between equals() method and == operators.
- Equals() method compares two strings character by character.
- == operator compares two object references to check whether they refers to same instance of string.
- String str="hello" will create separate object. String str2=str will pointing to same object but different reference variables.

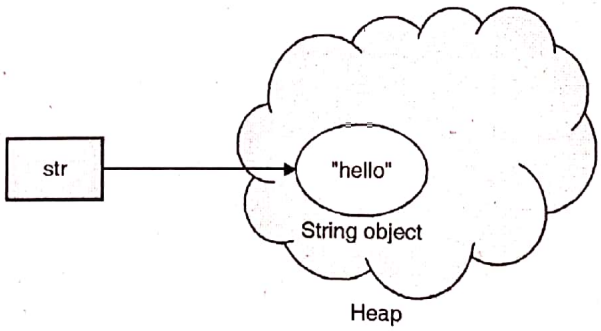

Fig. 3.7.2

- String str="Hello"; and String str2=new String(str); will create two separate objects.

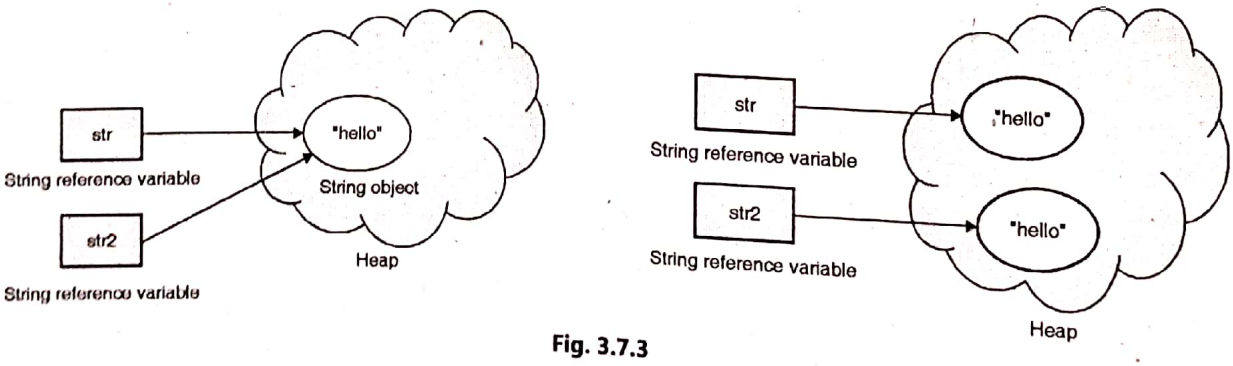

Fig. 3.7.3

simple program to compare string.

```
public class StringComparision
{
    public static void main(String[] args)
    {
      System.out.println("String comparison in Java");
      String str="abc";
      String str1="abc";
      System.out.println(str+" & "+ str1+" are equal? "+ str.equals(str1));
      String s1="abc", s2="ABC";
      System.out.println(s1+" & "+ s2+" are equal with case sensitive? "+ s1.equals(s2));
      String s3="xyz", s4="XYZ";
      System.out.println(s3+" & "+ s4+" are equal if case insensitive? "+ s3.equalsIgnoreCase(s4));
      System.out.println("Use of == operator");
      String str3="abc";
      String str4=new String("abc");
      System.out.println(str3+ " & " + str4+ " are different? : "+(str3==str4) );
    }
}
```

Output :

```
String comparison in Java
abc&abc are equal? true
abc& ABC are equal with case sensitive? false
xyz& XYZ are equal if case insensitive? true
Use of == operator
abc&abc are different? : false
```

Q. 5 What is a Constructor ? **SPPU : May 18, Dec. 18, Dec. 19, 2 Marks**

Ans. : Constructors

- A constructor can be used to set initial values for object attributes. It is a block of code that initializes the newly created object. At the time of calling constructor, memory for the object is allocated in the memory.
- A constructor resembles an instance method in java but it's not a method as it doesn't have a return type even void also.
- Constructor will be invoked by the JVM automatically, whenever you create an object.
- Constructor has same name as the class and looks like this in a java code.

```
public class MyClass
{
    //This is the constructor
    MyClass()
    {
    }
    ..
}
```

Here you can note that the constructor name matches with the class name and it doesn't have a return type.

Q. 6 What are its different types of constructor ? Demonstrate with suitable example the different types of constructors used in Java. **SPPU : May 18, Dec. 18, Dec. 19, 4 Marks**

Ans. :

Types of Java Constructors

There are two types of constructors in Java :

1. **Default constructor :** A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates default constructor with no arguments for the class. Arguments are the value that we pass in place of parameters. If we write a constructor with arguments or no-arguments then the compiler does not create a default constructor. Default constructor provides the default values to the object like 0, null, etc. depending on the type.
2. **Parameterized constructor :** A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor. There are no "return value" statements in constructor, but constructor returns current class instance. We can write 'return' inside a constructor.

Program 1

```
class Project
{
    int value1;
    int value2;
    Project ()
    {
    value1 = 10;
    value2 = 20;
      System.out.println("Inside Constructor");
    }
    public void display()
    {
      System.out.println("Value1 === "+value1);
      System.out.println("Value2 === "+value2);
    }
    public static void main(String args[])
    {
    Project p1 = new Project ();
      p1.display();
    }
}
```

Program 2

```
class Example1
{
        int variable1;
        int variable2;
        Example1()                          // no-argument constructor
        {
         variable1 = 30;
        variable2 = 40;
           System.out.println(" No-argument Constructor");
        }
        public void show()
        {
           System.out.println("variable1 = "+ variable1);
           System.out.println("variable2 = "+ variable2);
        }
        public static void main(String args[])
        {
        Example1 obj = new Example1();
           obj.show();
        }
}
```

Output :

```
No-argument Constructor
variable1 = 30
variable2 = 40
```

Program 3

```
class Example2
{
        int variable1;
        int variable2;
        Example2()                          // no-argument constructor
        {
           variable1 = 1;
           variable2 = 2;
           System.out.println(" First base Constructor");
        }
        Example2(int a)                     // argument constructor
```

```
    {
    variable1 = a;
      System.out.println("Second base Constructor");
    }
    public void show()
    {
      System.out.println("variable1 === "+ variable1);
      System.out.println("variable2 === "+ variable2);
    }
    public static void main(String args[])
    {
      Example2derived d1 = new Example2derived ();
          d1.display();
    }
}
class Example2derived extends Example2
{
    int variable3;
    int variable4;
    Example2derived()
    {
      //super(5);
      variable3 = 3;
      variable = 4;
      System.out.println("The Constructor of Derived");
    }
    public void show()
    {
      System.out.println("Variable1 === "+variable1);
      System.out.println("Variable2 === "+variable2);
      System.out.println("Variable1 === "+variable3);
      System.out.println("Variable2 === "+variable4);
    }
}
```

Output :

First base Constructor

The Constructor of Derived

Variable1 === 1

Variable2 === 2

Variable1 === 3

Variable2 === 4

Q. 7 What do you mean by method overloading ? Write a program which adds two integers and three integers by using overloaded methods for adding two and three integers respectively. **SPPU : May 17, 7 Marks**

Ans. :

Overloading Methods

- Method Overloading is a feature that allows a class to have multiple methods with same name by changing parameters. If we have to perform only one operation, has same name of the methods increases the readability of the program.
- For example you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int, int) for two parameters, and b(int, int, int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.
- Different ways to overload the method. There are three ways to overload the method in java.

1. Number of Parameters

For example :

add(int, int)

add(int, int, int)

2. Data type of Parameters

For example :

add(int, int)

add(int, float)

3. Sequence of Data type of Parameters

For example :

add(int, float)

add(float, int)

Invalid case of method overloading

If two methods have same name, same parameters and have different return type, then this is not a valid method overloading. This will throw compilation error. Following example will give a compilation error.

int add(int, int)

float add(int, int)

Advantages of Method Overloading

- Method overloading increases the readability of the program.
- Overloading in Java is the ability to create multiple methods of the same name, but with different parameters.
- The main advantage of this is cleanliness of code.
- Method overloading increases the readability of the program.
- Overloaded methods give programmers the flexibility to call a similar method for different types of data.
- Overloading is also used on constructors to create new objects given different amounts of data.
- You must define a return type for each overloaded method. Methods can have different return types.

Different Ways to Overload Method in Java

When you overloading methods, you must check the following rules :

1. Method name must be same.
2. Return type can be anything.
3. Parameters must be changed in one of the following three ways :
   a) Type of parameters b) Number of parameters
   c) Order of parameters

Here is a simple example that illustrates method overloading :

Program to demonstrate method overloading.

```
// Demonstrate method overloading.
class Arithmetic
{
    void test()
    {
      System.out.println ("No parameters");
    }
    // Overload test for one integer parameter.
    void test(int x)
    {
      System.out.println ("x: " + x);
    }
    // Overload test for two integer parameters.
    void test(int x, int y)
    {
      System.out.println ("x and y: " + x + " " + y);
    }
    // overload test for a double parameter
    double test (double x)
    {
      System.out.println ("double x: " + x);
      return x*x;
    }
}
class Overload
{
    public static void main (String args[])
    {
      Arithmetic a = new Arithmetic ();
      double result;                          // call all versions of test()
      a.test();
```

```
        a.test (20);
        a.test (20, 30);
        result = a.test (150.30);
        System.out.println ("Result of a.test (150.30): " + result);
    }
}
```

Output :

```
No parameters
a: 20
a and b: 20 30
double a: 150.30
Result of a.test(150.30): 22590.09
```

- As you can see, test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter by passing argument value 20, the third takes two integer parameters with argument value 20 and 30, and the fourth takes one double parameter by taking argument value 150.30.
- The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

Q. 8 Give example of static declaration. What are restrictions on methods which are declared static ?

SPPU : May 17, 7 Marks

Ans. : Static Variables

- **Static** is a non-access specifier in Java. The static keyword in Java is mainly used for memory management. The keyword static indicates that the particular member belongs to a type itself, rather than to an instance of that type.
- This means that only one instance of that static member is created which is shared across all instances of the class. We can apply java static keyword with variables, methods, blocks and nested class.
  a) Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.
  b) It is a variable which belongs to the class and not to object(instance).
  c) A static variable can be accessed directly by the class name and doesn't need any object.
  d) A single copy to be shared by all instances of the class.
- Static members are common for all the instances (objects) of the class but non-static members are separate for each instance of class.
- If you need to do the computation in order to set your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. Take a look at the below Java program to understand the usage of Static Block.

Program to demonstrate the use of static blocks

```
public class BlockEx
{
    // static variable
    static int i = 5;
```

```
    static int k;
    // static block
    static
    {
      System.out.println ("Static block initialized.");
      k = i * 10;
    }
    public static void main(String[] args)
    {
      System.out.println ("Inside main method");
      System.out.println ("Value of i: "+i);
      System.out.println ("Value of k: "+k);
    }
}
```

When you execute the above program, static block gets initialized and displays the values of the initialized variables.

Output :

Static block initialized

Inside main method

Value of i: 5

Value of k: 50

There are a few restrictions imposed on a static method :

- The static method cannot use non-static data member or invoke non-static method directly.
- The this and super cannot be used in static context.
- The static method can access only static type instance variable.
- There is no need to create an object of the class to invoke the static method.
- A static method cannot be overridden in a subclass.

Q. 9 State the use of final keyword in Java. **SPPU : May 19, 2 Marks**

Ans. :

- Keyword **final** is a non-access specifier in Java. You can declare final as a variable. So, it stops to modify its contents. Whenever you declare final variable you must need to initialize it. It is applicable only to a variable, a method or a class. It can be initialized in the constructor only.
- The blank final variable can be static also which will be initialized in the static block only. If the final variable is a reference, it means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from final array or words.

For example :

```
// a final variable
final int PH_INDICATOR = 5;
// a blank final variable
final int PH_INDICATOR;
// a final static variable PH
static final double PH = 5.568;
// a blank final static variable
static final double PH;
```

Ways to Initialize a final variable :

Compiler will throw a compile-time error if we have not initialized final variable. It can initialized by an initializer or an assignment statement only once. There are three ways to initialize a final variable :

1. You can initialize a final variable when it is declared. A final variable is not initialized while declaration is called blank final variable. Below are the two ways to initialize a blank final variable.
2. A blank final static variable initialized inside static block.
3. A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.

❑❑❑

Unit IV : Inheritance, Packages and Exception Handling using Java

Q. 1 What is Inheritance ?

SPPU . May 18, May 19, Dec. 19, 2 Marks

Ans. :

Inheritances

- Inheritance is one of the keystones of object-oriented programming because it allows the creation of hierarchical classifications. The process by which one class access the data members (properties) and methods (functionalities) of another class is called inheritance.
- The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Q. 2 List and explain the different types of Inheritance. Demonstrate how Java supports Multiple Inheritance.

SPPU . Dec. 17, May 19, Dec. 19, 6 Marks

Ans. :

1. Single Inheritance

Single Inheritance refers to a child and parent class relationship where a class extends the another class.

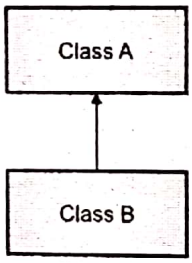

Fig. 4.1 : Single inheritance

2. Multilevel Inheritance

It refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.

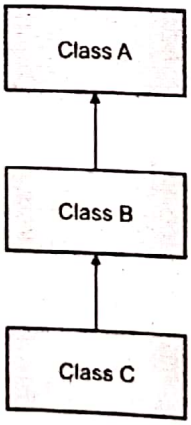

Fig. 4.2 : Multilevel inheritance

3. Hierarchical Inheritance

It refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.

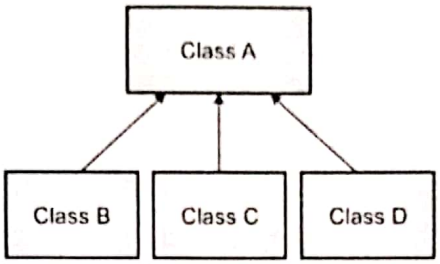

Fig. 4.3 : Hierarchical inheritance

4. Multiple Inheritance

It refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example : class C extends both classes A and B. Java does not support multiple inheritance. We have illustrated in interfaces section.

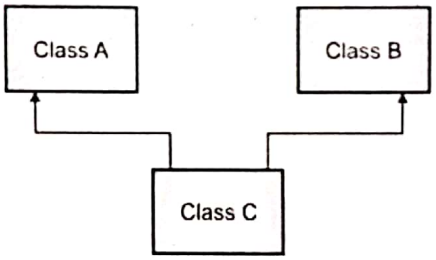

Fig. 4.4 : Multiple inheritance

5. Hybrid inheritance

Combination of more than one types of inheritance in a single program. For example : class A & B extends class C and another class D extends class A then this is a hybrid inheritance example because it is a combination of single and hierarchical inheritance.

Program

```
import java.io.*;
class GFG
{
    public static void main(String[] args)
    {
        dog d1 = new dog();
        d1.bark();
        d1.run();
        cat c1 = new cat();
        c1.meww();
    }
}

class Animal
```

```
{
        public void run()
        {
          String name;
          String colour;
          System.out.println("animal is running");
        }
}
class dog extends Animal
{
        public void bark()
        {
          System.out.println("wooh!wooh! dog is barking");
        }
        public void run()
        {
          System.out.println("dog is running");
        }
}
class cat extends Animal
{
        public void meww()
        {
          System.out.println("meww! meww!");
        }
}
```

Output :

```
wooh!wooh! dog is barking
dog is running
meww! meww!
```

Q. 3 What are advantages of using inheritance?

SPPU : May 18, 4 Marks

Ans. : Advantages of Inheritance

Not limited in Java, but in General Inheritance in Object Oriented Programming has lot of advantages.

Advantages :

- It helps in code reuse.
- It helps to avoid redundancy of code in subclass by inheriting from super-class.
- Saves time and effort as the main code need not be written again.
- Inheritance provides a clear model structure which is easy to understand without much complexity.
- Using inheritance, classes become grouped together in a hierarchical tree structure Code are easy to manage and divided into parent and child classes.

- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance can also make application code more flexible to change because classes that inherit from a common superclass can be used interchangeably. If the return type of a method is superclass.
- **Reusability** - Reusability enhanced reliability. The base class code will be already tested and debugged. facility to use public methods of base class without rewriting the same.
- **Extensibility** - extending the base class logic as per business logic of the derived class.
- **Data hiding** - base class can decide to keep some data private so that it cannot be altered by the derived class
- **Overriding** -With inheritance, we will be able to override the methods of the base class so that meaningful implementation of the base class method can be designed in the derived class.
- One of the key benefits of inheritance is to minimize the amount of duplicate code in an application by sharing common code amongst several subclasses. Where equivalent code exists in two related classes, the hierarchy can usually be refactored to move the common code up to a mutual superclass. This also tends to result in a better organization of code and smaller, simpler compilation units.

Q. 4 What is method overriding in Java? What is advantage of using overriding ? Demonstrate method overriding with example. **SPPU : May 17, May 19, Dec. 19, 6 Marks**

Ans. : Method Overriding

- Declaring a method in subclass which is already present in superclass is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- In this case the method in parent class is called overridden method and the method in child class is called overriding method.

Method Overriding Example

- Let's take a one example to understand this. We have two classes: A derived class Monkey and a base class Animal. The Monkey class extends Animal class. Both the classes have a common method void jump(). Monkey class is giving its own implementation to the jump() method or in other words it is overriding the jump() method.
- The purpose of Method Overriding is clear here. Derived class wants to give its own implementation so that when it calls this method, it prints Monkey is jumping instead of Animal is jumping.

Program

```
class Animal
{
        //Overridden method
        public void jump()
        {
           System.out.println("Animal is jumping");
        }
}
class Monkey extends Animal
{
        //Overriding method
```

```
    public void eat()
    {
      System.out.println("Monkey is jumping");
    }
    public static void main( String args [])
    {
      Monkey obj = new Monkey();
  /*This will call the child class version of eat() */
      obj.jump();
    }
}
```

Output :

Monkey is jumping

Advantages of method overriding

- The main advantage of method overriding is that the class can give its own specific implementation to an inherited method without even modifying the parent class code.
- This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Q. 5 What is Polymorphism ? Which type of polymorphism is method overriding ? **SPPU : Dec. 19, 5 Marks**

Ans. : Polymorphism

- It refers to ability to take more than one form in the programming.
- Which behaviour to use totally depends upon type of data used in the operation.
- If we consider the simple case of addition of two number. If we call the functions with two number then function should perform the addition.
- But if we pass two different strings then instead of addition, concatenation of string operation should be performed automatically.
- Polymorphism can be compile time polymorphism and run time polymorphism.
- Compile time polymorphism is possible with function and operator overloading whereas run time polymorphism is possible with virtual functions.
- Polymorphism example is shown in Fig. 4.5.

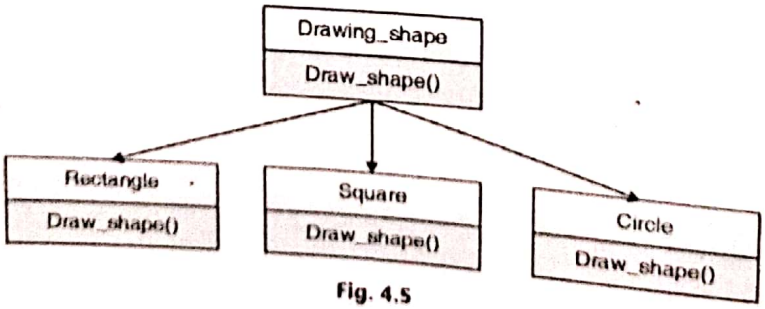

Fig. 4.5

- As shown in the Fig. 4.5, which Draw_shape() to call is totally depend upon the number of parameters to functions.
- If we are passing radius and centre coordinates then circle type of Draw_shape() should get called. If we passing just one side then Draw_shape() for square should get called and so on. Another example is shown in below Fig. 4.6

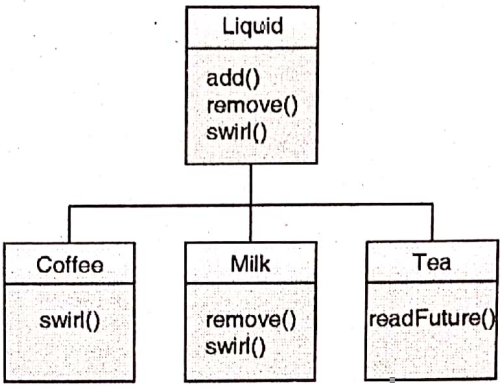

Fig. 4.6

Q. 6 Explain the concept of dynamic dispatch while overriding method in inheritance. Give example and advantages of doing so. **SPPU : Dec. 17, 5 Marks**

Ans. : Dynamic Method Dispatch

- Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method (parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch.
- Let's see an example to understand this.

Program

```
class A
{
    //Overridden method
    public void display()
    {
      System.out.println("display() method of parent class");
    }
}
class Demo extends ABC
{
    //Overriding method
    public void display()
    {
      System.out.println("display() method of Child class");
    }
    public void newMethod()
```

```
    {
      System.out.println("new method of child class");
    }
    public static void main( String args[])
    {
      /* When Parent class reference refers to the parent class object then in this case overridden method (the method of
      parent class)is called.*/
      ABC obj = new ABC();
      obj.disp();
      /* When parent class reference refers to the child class object then the overriding method (method of child class) is
      called. This is called dynamic method dispatch and runtime polymorphism*/
      A obj2 = new Show();
      obj2.display();
    }
}
```

Output :

```
display() method of parent class
display() method of Childclass
```

- In the above example the call to the display() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).

> **Note :** In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object obj2 is calling the disp().

- However, if you try to call the newMethod() method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following messages :
  - Exception in thread "main"java.lang.Error:.
  - Unresolved compilation problem.
  - The method xyz() is undefined for the type ABC.

Advantages of Dynamic Method Dispatch

- Dynamic method dispatch allows Java to support overriding of methods which is central for run-time polymorphism.
- It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- It also allows subclasses to add its specific methods subclasses to define the specific implementation of some of those methods.
- The main advantage of method overriding is that the class can give its own specific implementation to an inherited method without even modifying the parent class code.
- This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Q. 7 state the difference with examples : Static and Dynamic Dispatch. **SPPU : May 18, 3 Marks**

Ans. : Difference between Static and Dynamic Dispatch

- Dispatch is the act of sending something somewhere. In computer science, this term is used
- To indicate the same concept in different contexts, like to dispatch a call to a function, dispatch an event to a listener, dispatch an interrupt to a handler or dispatch a process to the CPU.
- There are two forms of dispatch, **static** and **dynamic**. The former means that a call to a method is resolved at **compile time** and the latter means that is resolved at **run time.**

Table 4.1 : Difference between static and dynamic dispatch

Sr. No.	Static Dispatch	Dynamic Dispatch
1.	Static dispatch (or early binding) happens when I know at compile time which function body will be executed when I call a method	Dynamic dispatch (or run-time dispatch or virtual method call or late binding) happens when I defer that decision to run time. This runtime dispatch requires either an indirect call through a function pointer, or a name-based method lookup.
2.	Some languages allow us to use a special method call syntax to force **static dispatch**. This is useful when we want to avoid open recursion, e.g. to make our base class more robust against changes in subclasses.	Some languages allow function call syntax for methods. In such a case, the programmer selects the fully qualified name of the method, so no **dynamic dispatch** is necessary.

Q. 8 State the use of Abstract class in Java. **SPPU : Dec. 17, May 19, 2 Marks**

Ans. : Abstract classes

- An abstract class is one that is not used to build objects, but only as a basis for creation subclasses. An abstract class exists only to express the common things of all its subclasses.
- A class that is not abstract is said to be concrete. You can create objects belonging to a concrete class, but not to an abstract class.
- A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.
- A "abstract" keyword is known as abstract class which is declared using class. It can have abstract function (function without body) as well as concrete function (regular methods with body).
- A normal class(non-abstract class) cannot have abstract methods. In this subsection you will learn what is a abstract class?, why you use it and what are the rules that you must remember while working with it in Java.
- An abstract class can not be instantiated, which means you are not allowed to create an object of it. Why?

Why require an abstract class?

- Let's say we have a class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say "Roar" in this method and Dog class will say "Woof".
- So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

- Since the Animal class has an abstract method, you must need to declare this class abstract.
- Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Q. 9 Explain the following Java concepts and state the difference with examples : Interface and Abstract class. **SPPU : May 18, 3 Marks**

Ans. :

Sr. No.	Interface Class	Abstract class
1.	In Java, interface is a reserved word with an additional, technical meaning. An "interface" in this sense consists of a set of instance method interfaces, without any associated implementations. It is similar to class. It is a collection of abstract methods.	A class that is declared using "**abstract**" keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (regular methods with body). A normal class (non-abstract class) cannot have abstract methods.
2.	**interface** keyword an interface can never be instantiated as the methods are unable to perform any action on invoking.	**abstract** keyword is used to create an abstract class in java. If a class have abstract methods, then the class should also be abstract using abstract keyword, else it will not compile.

3.	//Declaration using interface keyword
	Interface <interface_name>
	{ // declare constant fields
	// declare methods that abstract
	// by default.
	}
		//Declaration using abstract keyword
		Abstract <abstract_name>
		{ //This is abstract method
		//This is concrete method with body
		{ //Does something }
		}

4.	**Example :**
	Interface MyInterface
	{
	/* compiler will treat them as : public abstract void myMethod(); */
	public void myMethod();
	}
	class Show implements MyInterface
	{
	/* This class must have to implement the abstract myMethod otherwise it will get compilation error */
	public void myMethod()
	{
	System.out.println("implementation of myMethod");
	}
	public static void main(String arg[])
	{
	MyInterface obj=newShow();
	obj.myMethod();
	}
	}
	Output : implementation of myMethod
		Example :
		//abstract parent class
		abstract class Animal
		{
		//abstract method
		public abstract void sound();
		}
		//Cat class extends Animal class
		public class Cat extends Animal
		{
		public void sound()
		{
		System.out.println("Maw");
		}
		public static void main(String args[])
		{
		Animal obj=newCat();
		obj.sound();
		}
		}
		Output :
		Maw

Q. 10 What is mean by Packages in Java? Explain with suitable example. **SPPU : May 19, 3 Marks**

Ans. : Defining a Package

- A **Package** can be defined as container that grouping of related types (classes, interfaces, annotations and enumerations) providing access protection and namespace management. Like all subroutines in Java, the routines in the standard API are grouped into classes.
- To provide larger-scale organization, classes in Java can be grouped into packages. One of the sub-packages of java, for example: is called "awt". Since awt is contained within java, its full name is actually java.awt.
- This package contains classes that represent GUI components such as buttons and menus in the AWT, the older of the two Java GUI toolboxes, which is no longer widely used.
- However, java.awt also contains a number of classes that form the foundation for all GUI programming, such as the Graphics class which provides routines for drawing on the screen, the Color class which represents colors and the Font class which represents the fonts that are used to display characters on the screen. Since these classes are contained in the package java.awt, their full names are actually java.awt.Graphics, java.awt.Color and java.awt.Font.
- Similarly, javax contains a sub-package named javax.swing, which includes such classes as javax.swing.JButton, javax.swing.JMenu, and javax.swing.JFrame. The GUI classes in javax.swing, together with the foundational classes in java.awt, are all part of the API that makes it possible to program graphical user interfaces in Java.
- The java package includes several other sub-packages, such as java.io, which provides facilities for input/output, java.net, which deals with network communication, and java.util, which provides a variety of "utility" classes. The most basic package is called java.lang. This package contains fundamental classes such as String, Math, Integer, and Double.
- It might be helpful to look at a graphical representation of the levels of nesting in the java package, its sub-packages, the classes in those sub-packages, and the subroutines in those classes. This is not a complete picture, since it shows only a very few of the many items in each element.

Finding Packages and Classpath

- A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
- The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
- To use a class or a package from the library, you need to use the import keyword.

Syntax

```
import package.name.Class;
// Import a single class
import package.name.*;
// Import the whole package
```

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

In the example above, java.util is a package, while Scanner is a class of the java.util package.

Consider the following scene of a movie :

- "Humphrey Bogart has just killed the German Commander in front of the Chief of Police. The Chief then calls his office and informs them that the Commander has been murdered, and that they should round up the usual suspects".
- For Java, the CLASSPATH variable is a list of the usual suspects. When Java wants to find a class file, it looks in all the directories that are listed in the CLASSPATH*. In order to have Java look in new places, just add more paths to the CLASSPATH variable.

  (* There are also system paths that are searched that are not listed in the CLASSPATH.)

What is a Package ?

- The simplest definition of a package is a folder that contains java class files. However, packages do more than that. They also indicate where a java class can be found. Essentially, packages allow for an extension to the CLASSPATH list, without adding new paths to it. If a class file is in a package named sandy, then Java will look in all the paths in the CLASSPATH with sandy appended to them.
- For example, suppose the CLASSPATH contains :

```
//      myData
//      myFiles
```

- Java will then look for the class file in,

```
//      myData/sandy
//      myFiles/sandy
```

- As a more complicated example, suppose the package is skf.cg124.sk, then Java will look for the class file in,

```
//      myData/ skf/cg124/sk
// myFiles/skf/cg124/sk
```

Q. 11 How is access protection provided for packages ? **SPPU : Dec. 18, 2 Marks**

Ans. : Access Protection

- You can see, many levels of protection to allow well control over the visibility of variables and functions within classes, subclasses, and packages by Java platform.
- Packages act as containers for classes and other subordinate packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members :
  - Subclasses in the same package.
  - Non-subclasses in the same package.
  - Subclasses in different package.
  - Classes that are neither in the same package nor subclasses.
- The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 4.2 sums up the interactions.
- While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere i.e. in same class, outside of class and end method of program.
- Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that childclass your class directly, then declare that element **protected**.

- Table 4.2 relates only to members of classes. A class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

Table 4.2 : Access Modifier

YES: ✓ NO: ×	Public	Private	Protected	No Modifier
Same class	✓	✓	✓	✓
Same package subclass	✓	×	✓	✓
Same package non-subclass	✓	×	✓	✓
Different package subclass	✓	×	✓	×
Different package non-subclass	✓	×	×	×

Q. 12 What is an Interface in Java? How is this different than a class? Give example of interface.

SPPU : May 17, Dec. 18, May 19 6 Marks

Ans. :

- **Definition :** In Java, interface is a reserved word with an additional, technical meaning. An **"interface"** in this sense consists of a set of instance method interfaces, without any associated implementations. It is similar to class. It is a collection of abstract methods.

Example :

```
access_specifier interface interface_name
{
    return_type method_name1(parameter_list);
    return_type method_name2(parameter_list);
    ----
    type final_variable_name1 = value;
    type final_variable_name2= value;
    -----
}
```

- After JDK 8, java permits interface to have default implementation for any method. But it is still infrequent in programming and original purpose of interface has remained same.
- Variables are not instance variable. They serve as constants and cannot be changes by any implementing class. They can be defined inside interface but they are implicitly final and static.
- All variables methods and are implicitly public in interface.

Implementation :

- A class can implement an interface by providing an implementation for each of the methods specified by the interface. Here is an example of a very simple Java interface :

```
public interface Shapes
{
    public void draw(Graphics g);
}
```

- This looks much like a class definition, except that the implementation of the draw() method is omitted. A class that implements the interface Shapes must provide an implementation for this method. So, the class can also include other methods and variables.

For example :

```
public class Circle implements Shapes
{
    public void draw(Graphics g)
    {
      . . . // do something, draw a Circle
    }
    . . . // other methods and variables
}
```

- Note that to implement an interface, a class must do more than simply provide an implementation for each method in the interface; it must also state that it implements the interface, using the reserved word implements as in this example : "public class Circle implements Shapes".
- Any class that implements the Shapes interface defines a draw() instance method. Any object created from such a class includes a draw() method.

Nested Interface

- Java permits to write nested interfaces. One interface can be member of another interface or class.
- A nested interface is also called member interface and can be declared as public private or protected. Note that the top level interface must be declared as public or default.
- If a nested interface is used outside the enclosing scope, it must be specified with name of outer class or interface with dot operator (.).
- It enable us to logically group classes that are only used in one place, write more readable and maintainable code and increase encapsulation.

Program 1

```
interface Callable
{
    void displayMsg(String msg);
    public interface CallableInt
    {
      public void display Num(int num);
    }
}
class Screen implements Callable.CallableInt
{
    public void display Num(int n)
    {
      System.out.println("Number is: " + n);
```

```
    }
}
public class InterfaceController
{
    public static void main(String args[])
    {
        Callable.CallableInt c;
        c = new Screen();
        c.displayNum(10);
    }
}
```

Output :

```
Number is : 10
```

- In the above example outer interface is Callable and inner interface is CallableInt.

Applying Interface :

- An application of interface can be understood by considering and extracts of an abstract java application where Animal interface will define the must have methods for implementing classes.
- Let us call the interface as MotionAnimal.

Example :

```
interface MotionAnimal
{
    public void run();
    public void walk();
    public void jump();
}
```

- Hence this interface define that any class which will implement Motion Animal must provide the implementation of these three methods. It real world translation, all animals should be able to run, walk and jump.
- Interface specifies how an animal object can be interacted and what the capabilities of animal object. It also serves an objective. By just sighted the interface, one can get a clear idea of capabilities of animal object and the ways to interact with it.

Program 2

```
interface printable
{
    void print();
}
class A6 implements printable
{
    public void print()
    {
```

```
        System.out.println("Welcome");
    }
    public static void main(String args[])
    {
        A6 obj = new A6();
        obj.print();
    }
}
```

Output :

```
Welcome
```

In the above example, the Printable interface has only one method, and its implementation is provided in the A6 class.

Q. 13 State two major differences in class and interface. "Interface gives multiple inheritance facility just as in C++" justify.

SPPU : Dec. 17, Dec.18, 7 Marks

Ans. :

Class and Interface both are used to create new reference types. A class is a collection of fields and functions that operate on fields. An interface has fully abstract methods i.e. methods with no body. An interface is syntactically similar to the class but there is a major difference between class and interface that is a class can be instantiated, but an interface can never be instantiated.

Some difference between a class and interface is given as in Table 4.6.2.

Table 4.3 : Difference between a class and interface

Comparison	Class	Interface
Basic	A class is instantiated to create objects.	An interface can never be instantiated as the methods are unable to perform any action on invoking.
Keyword	Class	Interface
Access Modifiers	The members of a class can be private, public or protected.	The members of an interface are always public.
Methods	The methods of a class are defined to perform a specific action.	The methods in an interface are purely abstract.
Implement/ Extend	A class can implement any number of interface and can extend only one class.	An interface can extend multiple interfaces but can not implement any interface.
Constructor	A class can have constructors to initialize the variables.	An interface can never have a constructor as there is hardly any variable to initialize.

- Yes, Interface gives multiple inheritance facility just as in C++ because when one class extends more than one classes then this is called **multiple inheritance**. For example: Class Z extends class X and Y then this type of inheritance is known as multiple inheritance. Java doesn't allow multiple inheritance.

- As we have explained in the inheritance, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.
- Yes, we can implement more than one interfaces in our program because that does not cause any ambiguity for example.

Program

```
interface X
{
        public void myFunc();
}
interface Y
{
        public void myFunc();
}
class MyClass implements X, Y
{
        public void myFunc()
        {
           System.out.println("More than one interfaces implements");
        }
        public static void main(String args[])
        {
           MyClassobj=newMyClass();
           obj.myFunc();
        }
}
```

Output :

More than one interfaces implements

Q. 14 Define the term exception and advantages of Exception Handling. **SPPU : Dec. 17, Dec. 19, 4 Marks**

Ans. : Exception :

- Exceptions are nothing but the runtime errors. It generally alters the flow of program execution. When exception occurs, program execution stops and terminates the program abnormally. Program execution does not continue. In this situation we get system generated error on screen.
- The statement after the instructions which causes exceptions does not get executed. It is not always acceptable to terminate program abnormally. In safety critical applications it not acceptable to terminate program abnormally.
- Exceptions occur due to so many reasons. Some of the reasons are as follows :
  - Invalid input
  - File not available

- Network error
- In sufficient memory
- Improper use of array

- Some of the real time exceptions are :
  - We are trying to divide the number by zero.
  - We are trying to open file which is not existing.
  - We are storing incompatible value in array.
  - RAM is not enough to load your program.
  - We are trying to access address which is pointing to null.
  - Improper type conversion.
  - SQL errors.

Advantages of Exception Handling

- The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application that is why we use exception handling.
- Let's take a scenario :
  - statement 1;
  - statement 2;
  - statement 3;
  - statement 4;
  - statement 5; //exception occurs
  - statement 6;
  - statement 7;
  - statement 8;
  - statement 9;
  - statement 10;
- Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.
- The separation of error-handling code from normal code unlike traditional programming languages, there is a clear-cut distinction between the normal code and the error-handling code. This separation results in less complex and more readable (normal) code.
- A logical grouping of error types Exceptions can be used to group together errors that are related. This will enable us to handle related exceptions using a single exception handler. When an exception is thrown, an object of one of the exception classes is passed as a parameter. Objects are instances of classes, and classes fall into an inheritance hierarchy in Java. This hierarchy can be used to logically group exceptions. Thus, an exception handler can catch exceptions of the class specified by its parameter, or can catch exceptions of any of its sub-classes.
- The ability to propagate errors up the call stack and another important advantage of exception handling in object oriented programming is the ability to propagate errors up the call stack.
- Exception handling allows contextual information to be captured at the point where the error occurs and to propagate it to a point where it can be effectively handled.

Q. 15 What is an exception in Java ? What do you mean by handling an exception ? Give example to show the use of try( ), Catch( ) methods.

SPPU : May 17, Dec. 18, May 19, Dec. 19, 6 Marks

Ans. :

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.
- Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.
- The general form of an exception-handling block is as follows :

```
try
{
    // block of code to monitor for errors
}
catch(ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
catch(ExceptionType2 exOb)
{
    // exception handler for ExceptionType2
}
// ...
finally
 {
 // block of code to be executed before try block ends
}
```

Here, Exception Type is the type of exception that has occurred.

Exception Types

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

automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

- The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment itself. Stack overflow is an example of such an error. These typeof Errors are typically created in response to catastrophic failures that cannot usually be handled by your program.
- The top-level exception hierarchy is shown in Fig. 4.7.

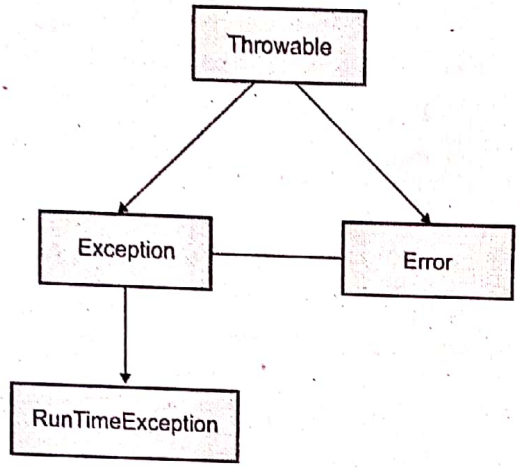

Fig. 4.7 : Top-level exception hierarchy

Uncaught Exceptions

- Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error.

```
class DivByZero
{
    public static void main(String args[])
    {
        int num1=30, num2=0;
        int output=num1/num2;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of DivByZero to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

Try and Catch Blocks

- Java provides exception handling mechanism in the form of **try** and **catch** block. Using **try-catch** block provides two advantages :

  First, exception will not cause program to end abruptly if successfully caught,

  Second, user can specify its own operation to recover from exception in catch block.
- Any code that can generate the exception is enclosed in **try** block. So we simply put inside **try** block, the entire code that we want to monitor.

- **catch** block immediately follows the **try** block and it contains the code that will only execute if exception is generated and caught successfully. So any recovery code, alternate code or message to user can be encapsulated in **catch**.
- **catch** specifies the exception type that we wish to catch. If the generated exception type is matched with the one specified in **catch**, exception is considered to be successfully caught.
- Let us consider a simple example demonstrating try and catch block operation as shown in the following example.

Program

```
class DivByZero
{
    public static void main(String args[])
    {
      int num1, num2;
      try
      {
      num1 =0;
      num2 =62/ num1;
            System.out.println(num2);
      }
      catch(ArithmeticException e)
      {
            System.out.println("You should not divide a number by zero");
      }
      catch(Exception e)
      {
            System.out.println("Exception occurred");
      }
            System.out.println("I'm out of try-catch block in Java.");
    }
}
```

Output :

```
You should not divide a number by zero
I'm out of try-catch block in Java.
```

- Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Put differently, catch is not "called," so execution never "returns" to the try block from a catch. Thus, the line "You should not divide a number by zero." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try / catch mechanism.

- A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement. A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described shortly). The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement.

Q. 16 How does Java manage Input / Output using Streams ? Differentiate Byte Stream, Character Stream and Predefined Stream ? **SPPU : Dec. 19, 6 Marks**

Ans. : Managing I/O

Java provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, I/O devise, other program etc.

Based on the data they handle there are two types of streams :

1. **Byte Streams** : These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc. **Byte Stream.**

   Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

2. **Character Streams** : These handle data in 16 bit Unicode. Using these you can read and write text data only.

   The Reader and Writer classes (abstract) are the super classes of all the character stream classes: classes that are used to read/write character streams.

   Whereas the InputStream and OutputStream classes (abstract) are the super classes of all the input/output stream classes: classes that are used to read/write a stream of bytes.

When to use Character Stream over Byte Stream ?

In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

When to use Byte Stream over Character Stream ?

Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.

- The main **difference between Byte Stream** and **Character Stream** in Java is that the **Byte Stream** helps to perform input and output operations of 8-bit **bytes** while the **Character Stream** helps to perform input and output operations of 16-bit Unicode. A **stream** is a **sequence** of data that is available over time
- File is important medium for permanent data storage. Along with file, I/O is also one of the important concept in java programming. Many time programs need input and need to store output. Java provides many classes for I/O system. Data source can be keyboard, file, network connections, array, etc.
- Data destination can be console output with monitors, output to file, output to network connections, output to array, etc. Sometimes data can be from memory also like array. Java provides many classes for manipulation of data from keyboard, file and network connections. Concepts of streams are provided to handle different types of data. Data storage medium can be different but it can be manipulated using streams only. In subsequent topic we will see in greater details concept of I/O streams.

Q. 17 Define the Streams. **SPPU : May 19, Dec. 19, 2 Marks**

Ans. : Streams

- Stream is nothing but sequence of data. It is nothing but flow of data for reading and writing purpose. Concept of index is not supported in stream. All input output is supported using stream concept.

- All the streams can be divided into input and output stream. Data input that is reading is possible with input stream whereas data storage that is writing is possible with output stream. Stream supports many kind of data including byte, characters, objects, etc.
- **OutputStream :** Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.
- **InputStream :** Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.
- Input / output stream is shown in Fig. 4.8

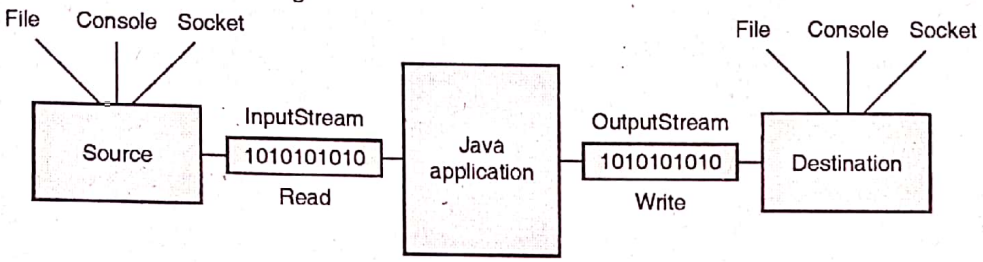

Fig. 4.8 : Input output streams

- Input and output streams are again divided into byte and character stream.
- It is decided by type of data flow from stream. If we pass byte data then it is byte stream. If we pass character data, then it character stream :
  - ByteStreams
  - CharacterStreams

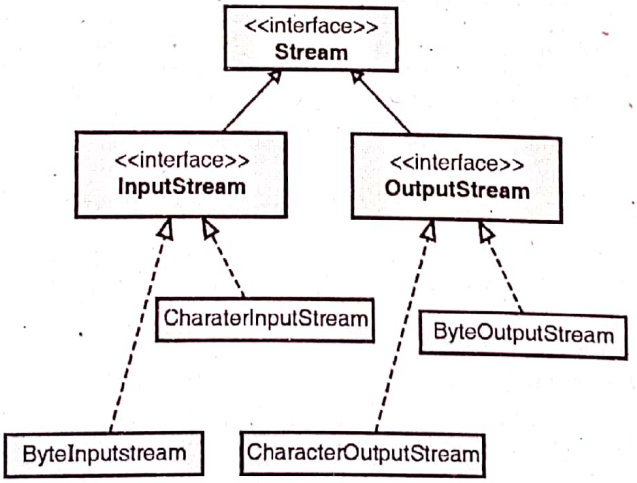

Fig. 4.9 : Stream types

Q.18 What is Byte Streams ? State any two examples of each Byte Stream classes for I/O in Java.

SPPU : Dec. 18, May 19, 3 Marks

Ans. : ByteStream

- It provides easy way to handle input and output of bytes. It is used while reading or writing binary data. Byte streams are implemented by two main classes for input and output purposes namely **InputStream** and **OutputStream.**
- They are called byte-oriented because they read or write 8-bit (byte). These classes are generally used to read and write non-character or binary data.

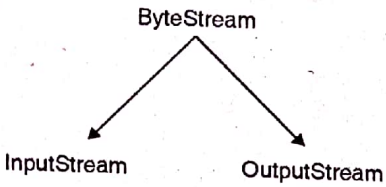

Fig. 4.10 : ByteStream

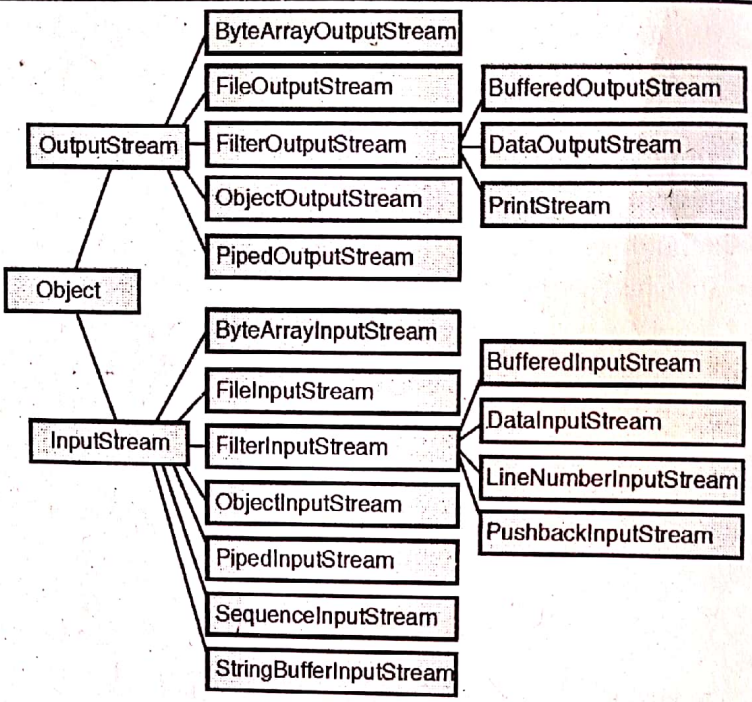

Fig. 4.11 : ByteStream classes

Table 4.4

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype.
DataOutputStream	An output stream that contain method for writing java standard data type.
FileInputStream	Input stream that reads from a file.
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and print in () method.

- In this section important byte class will be discussed with example.
- **FileInputStream** : It creates **Input Stream** to read bytes from a file. At the time of creating object, file name can be passed to constructors. It throws **FileNotFound Exception** exception. We need to catch that exception otherwise program will not get compiled.

Program 1

```
import java.io.*;
public class FileInputStreamExample
{
    public static void main(String[] args)
    {
        System.out.println("This is example of File input Stream");
```

```
        try
        {
                FileInputStream f = new FileInputStream("FileInput.txt");
                int i=0;
                System.out.println("Available data:"+f.available());
                while((i=f.read())!=-1)
                {
                    System.out.print((char)i);
                }
                System.out.println("Available
                data:"+f.available());
        }
        catch(Exception e)
        {
                System.out.println("Specified file not found");
        }
    }
}
```

Output :

```
This is example of File input Stream
Available data:44
This is example of file input stream in java Available data:0
```

FileOutputStream : It is used to write data in file. Data can be written in byte format only. It also throws **FileNotFoundException** which should be handled otherwise program will not get compiled.

Program 2

```
import java.io.*;
public class FileOutputStreamExample
{
    public static void main(String args[])
    {
      try
      {
            FileOutputStream fout = new FileOutputStream("example.txt");
            Strings="This string will be written in file example.txt" +"you can check out by opening file
            from current directory";
            Byteb[]=s.getBytes();                    //converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("Data Successfully written in file");
```

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

Output :

```
Data Successfully written in file
Contents of example.txt
```

This string will be written in file example.txt you can check out by opening file from current directory

ByteArrayInputStream : In this option byte array will be act as input to program instead of files. It means data source can be file, array, keyboard, network connections, etc. Constructors of class will take argument as byte array. We can get specified bytes from byte array also. Program 4.9.3 shows the construction of **ByteArrayInputStream.**

Program 3

```
import java.io.ByteArrayInputStream;
public class ByteArrayInputStream Example
{
    public static void main(String[] args)
    {
        System.out.println("This is example of byte array as input stream");
        String str="This is string for byte array";
        byteb[]=str.getBytes();
        //need to extract bytes from string
        ByteArrayInputStream ba=new ByteArrayInputStream(b);
        System.out.println("Successfully read data from array");
        int a=ba.read();
        while(a!=-1)
        {
                System.out.print((char)a);
                a=ba.read();
        }
    }
}
```

Output :

```
This is example of byte array as input stream
Successfully read data from array
This is string for byte array
```

ByteArrayOutputStream : It is used to write data in byte array. It means data destination is array in ByteArrayOutputStream.

Program 4

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
public class ByteArrayOutputStreamExample
{
    public static void main(String[] args)
    {
      System.out.println("This is example of byte array as output stream");
      ByteArrayOutputStream ba= new ByteArrayOutputStream();
      String str="This is string for writing in byte array";
      byte b[]=str.getBytes();
      //need to extract bytes from string
      try
      {
            ba.write(b);
      }
      catch(IOException e)
      {
            System.out.println("Error in writing data");
      }
      System.out.println("Reading data from output stream");
      byte c[]=ba.to ByteArray();
      for(int i=0;i<b.length;i++)
      {
            System.out.print((char)c[i]);
      }
    }
}
```

Output :

```
This is example of byte array as output stream
Reading data from output stream
This is string for writing in byte array
```

BufferedInputStream : It is used to read information from stream. It provides buffering to input stream for performance purposes. It attaches memory buffer to stream. Java's **BufferedInputStream** class allows you to "wrap" any **Input Stream** into a buffered stream and achieve this performance improvement.

Program 5

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
public class BufferedInputExample
{
    public static void main(String args [])
    {
      System.out.println("Buffered input stream example\n");
      try
      {
            FileInputStream fin = new FileInputStream("example.txt");
            BufferedInputStream bin = new BufferedInputStream(fin);
            int i;
            while((i=bin.read())!=-1)
            {
                System.out.print((char)i);
            }
            bin.close();
            fin.close();
      }
      catch(Exception e)
      {
            System.out.println(e);
      }
    }
}
```

Output :

```
Buffered input stream example
This string is for reading by buffered input stream example
```

Buffered Output Stream : Java **Buffered Output Stream** class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Program 6

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
class BufferedOutputExample
{
    public static void main(String args [])
    throws Exception
```

```
    {
        FileOutputStream fout = new FileOutputStream("example.txt");
        BufferedOutputStream bout = new BufferedOutputStream(fout);
        Strings="This string will be written in file by buffering";
        byte b[]=s.getBytes();
        bout.write(b);
        bout.flush();
        bout.close();
        fout.close();
        System.out.println("Successfully written in file.");
    }
}
```

Output :

```
Successfully written in file.
Contents of example.txt
This string will be written in file by buffering
```

Q. 19 What is Character Streams ? State any two examples of each Character Stream classes for I/O in Java.

SPPU : May 17, May 18, Dec. 18, May 19, 7 Marks

Ans. :

Character Stream

- It is one of the important addition in java. It is made as counterpart of byte oriented stream. All the classes are made by comparing with byte oriented stream. Super class of character oriented stream are Reader and Writer more like **InputStream** and **OutputStream** in byte stream.
- Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer. For input purpose Reader class is used whereas for output or writing purpose Writer class is used. Character streams are added specifically to work with character data(character, character array, string).
- The character stream classes differ from the byte streams classes in that they operate on buffered input and output and properly convert each character from the encoding scheme of the native operating system to the Unicode character set used by the Java platform.
- On the other hand, **InputStream** and **OutputStream**, and their subclasses operate on bytes and arrays of bytes. The byte-oriented streams can read characters, but they only correctly handle 7-bit ASCII characters.
- Character stream classes are added to replace byte stream classes.

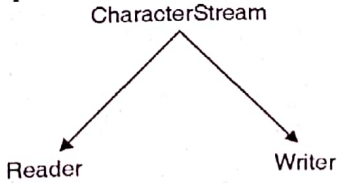

Fig. 4.12 : CharacterStream

- Some important Character Stream classes.

Table 4.5

Character Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character.
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input.
Writer	Abstract class that define character stream output.

- Although byte stream classes offer us a lot of efficiency and flexibility when dealing with bytes and binary data, they are not the best way to handle character input and output.
- Character streams operate directly on Unicode characters and so for storing and retrieval of Unicode text for internationalization character streams are a better alternative to byte streams.
- For this reason the character stream classes were introduced to the language in JDK 1.1 to offer a more efficient and easier way of dealing with character-based I/O.

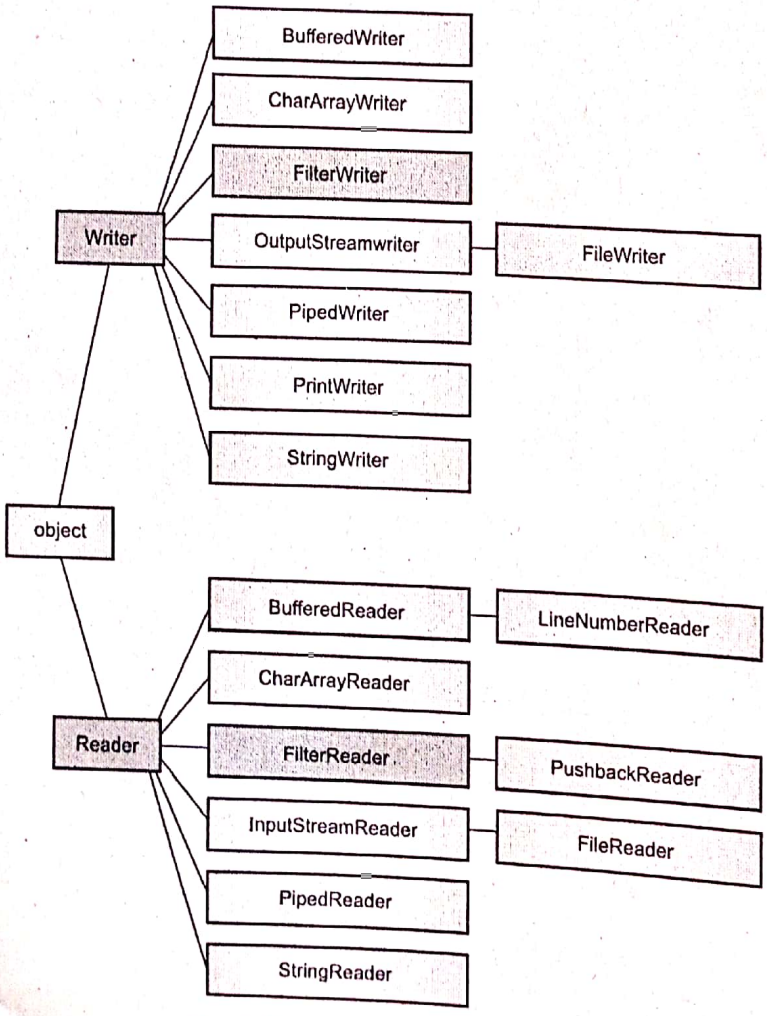

Fig. 4.13 : Character Stream classes

Reader

It is abstract class which defines character input stream. It handles UNICODE characters. It implements closeable and Readable interfaces. All methods in this class throws IO Exception. Different Reader classes are Buffered Reader, Char Array Reader, Filter Reader, Input Stream Reader, Piped Reader, String Reader, etc.

Writer

It is abstract class which defines character output stream. It implements Closeable, Flushable and Appendable interfaces. All methods throw IO Exception. It contains different classes like Buffered Writer, Char Array Writer, Filter Writer, Output Stream Writer, Piped Writer, Print Writer, String Writer, etc.

CharacterArrayReader() :

- The Java CharArrayReader class (java.io.CharArrayReader) enables you to read the contents of a char array as a character stream.
- The Java CharArrayReader is handy when you have data in a char array, but need to pass that data to some component which can only read from a Reader.

Example :

```
char[] chars = "123".toCharArray();

CharArrayReadercharArrayReader = new CharArrayReader(chars);

int data = charArrayReader.read();
while(data != -1)
{
    //do something with data
    data = charArrayReader.read();
}
charArrayReader.close();
```

CharacterArrayWriter() :

The CharArrayWriter class can be used to write common data to multiple files. This class inherits Writer class. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.

Example :

```
CharArrayWritercharArrayWriter = new CharArrayWriter();

charArrayWriter.write("CharArrayWriter");

char[] chars1 = charArrayWriter.toCharArray();

charArrayWriter.close();
```

Q. 20 What are predefined I/O classes ? **SPPU : Dec. 18, May 19, 3 Marks**

Ans. :

Predefined Streams

- There are some predefined streams already available for out java programs. They are available through a class called System. System class is defined in java. lang which is implicitly imported all our java programs.
- The System class contains three stream variables : in, out, err.
- The three stream variables are declared with System class as public, static and final. So need to create object of System class to access these three. Name of System class is enough.
  - **System.out** refers to standard output stream,
  - **System.in** refers to standard input stream and
  - **System.err** refers to standard error stream.
- Standard input stream is connected to keyboard by default. Standard output stream and standard error stream is connected to console by default.
- System.in is an object of Input Stream class. Similarly System.out and System.err are both objects of PrintStream class.
- These are byte streams, even though they are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

❑❑❑

Unit V : Multithreading in Java

Q. 1 Compare and contrast between thread based and process based multitasking. **(4 Marks)**

Ans. :

There are two distinct types of multitasking : Process based and thread-based.

Sr. No	Process-based multitasking	Thread-based multitasking
1.	A process is, in core, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.	The thread is the smallest unit of code capable of dispatch.
2.	A process is, at its core, a program that is being implemented. Thus, the feature that enables your computer to run two or more programs simultaneously is process-based multitasking.	This implies that two or more tasks may be carried out concurrently by a single program.
3.	Process-based multitasking, for instance, allows you to run the Java compiler at the same time as using a text editor.	A text editor, for example, can format text at the same time as it is being printed, as long as two different threads perform these two acts.
4.	Multitasking based on processes deals with the "big picture".	Multitasking based on thread manages the details.

Multithreading enables you to write very powerful programs that allow :

- Full use of the CPU, since it is possible to reserve idle time to a minimum.
- This is especially important because idle time is shared in the collaborative, networked environment in which Java operates.

Q. 2 Explain in detail Java thread model. **(6 Marks)**

Ans. :

Java Thread Model

- Java uses threads to enable the complete setting to be asynchronous. This benefit reduces inefficiency by preventing the waste of CPU cycles.
- The value of a multithreaded environment is best understood in contrast to its counterpart.
- Single-threaded systems use a method called an event loop with polling.
- In this model, one thread of management runs in Associate in attention infinite loop, polling one event queue to make a decision what to try and do next. Once this polling mechanism returns with, say, a proof that a network file scan (is prepared) to be read, then the event loop notices management to the suitable event handler.
- Until this event handler returns, nothing else will happen within the system. This wastes mainframe time. It also can lead to one a part of a program dominating the system and preventing the other events from being processed.
- The advantage of Java's multithreading is that the most loop/polling mechanism is removed.
- Threads exist in several states.

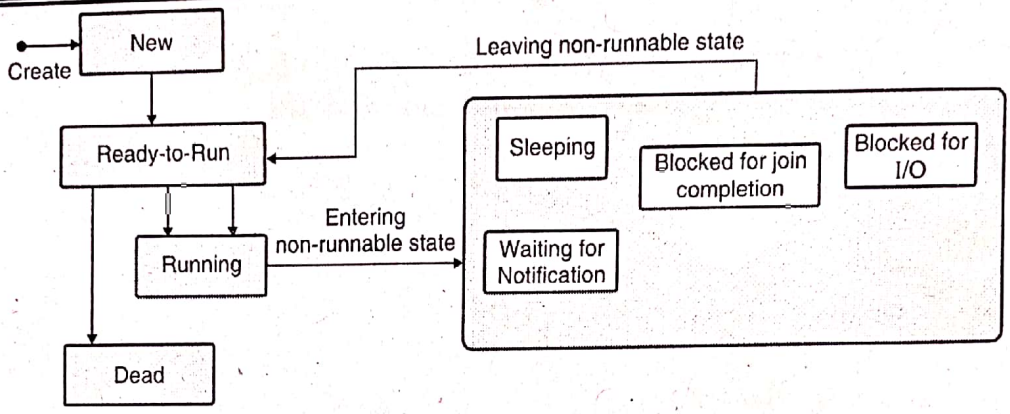

Fig. 5.1 : Thread States

- As shown in Fig. 5.1, the method has many states, equally a thread exists in many states. A thread may be within the following states :
  - A thread may be running.
  - It may be able to run as presently as it gets CPU time. A running thread may be suspended, that quickly suspends its activity.
  - A delayed thread will then be resumed, permitting it to choose up wherever it left off.
  - A thread may be blocked once looking ahead to a resource.
  - At any time, a thread may be terminated, that stops its execution instantly.
- Once terminated, a thread can't be resumed.

Types of Java Thread Model

Java thread model can be defined in the following three models :

a) Thread Priorities
b) Synchronization
c) Messaging

(a) Thread Priorities

Each thread has its own priority in Java. Thread priority is an absolute integer value. Thread priority decides only when a thread switches from one running thread to next, called *context switching*. Priority does increase the running time of the thread or gives faster execution.

(b) Synchronization

- Java supports an asynchronous multithreading, any number of thread can run simultaneously without disturbing other to access individual resources at different instant of time or shareable resources.
- But some time it may be possible that shareable resources are used by at least two threads or more than two threads, one has to write at the same time, or one has to write and other thread is in the middle of reading it.
- For such type of situations and circumstances Java implements synchronization model called *monitor*.
- As a thread enter in monitor, all other threads have to wait until that thread exits from the monitor.
- In such a way, a monitor protects the shareable resources used by it being manipulated by other waiting threads at the same instant of time synchronization.

(c) Messaging

- A program is a collection of more than one thread.
- Threads can communicate with each other. Java supports messaging between the threads with lost-cost.
- It provides methods to all objects for inter-thread communication.
- As a thread exits from synchronization state, it notifies all the waiting threads.

Q. 3 Explain Main Thread in Java. **(2 Marks)**

Ans. : The Main Thread

Java offers worked in help for multithreaded programming.

- A multi-strung program contains at least two sections that can run simultaneously.
- Each a piece of such a program is known as a thread and each thread characterizes a different way of execution.
- When a Java program fires up, one thread starts running right away. This is generally called the fundamental thread of our program, since the one is executed when our program starts.
- A thread can be made by applying the Runnable interface and abrogating the run() technique.
- The Main thread in Java is the one that starts executing when the program begins. All the youngster threads are produced from the Main thread. Likewise, it is the last thread to complete execution as different shut-down activities are performed by it.

Q. 4 Write a simple main thread program. **(4 Marks)**

Ans. :

```
public class Demo {
    public static void main(Thread args[])
    {
      Thread t = Thread.currentThread();
      System.out.println("Main thread: " + t);
      t.setName("current");
      System.out.println("Current thread: " + t);
      attempt {
            for (int i = 1; i<= 5; i++) {
            System.out.println(i);
            Thread.sleep(10);
      }
    }
    get (InterruptedException e)
    {
      System.out.println("Main thread is interrupted");
    }
    System.out.println("Exiting the Main thread");
    }
}
```

Output :

```
Primary thread: Thread[main,5,main]
Current thread: Thread[current,5,main]
1
2
3
4
5
Leaving the Main thread
```

JVM Thread :

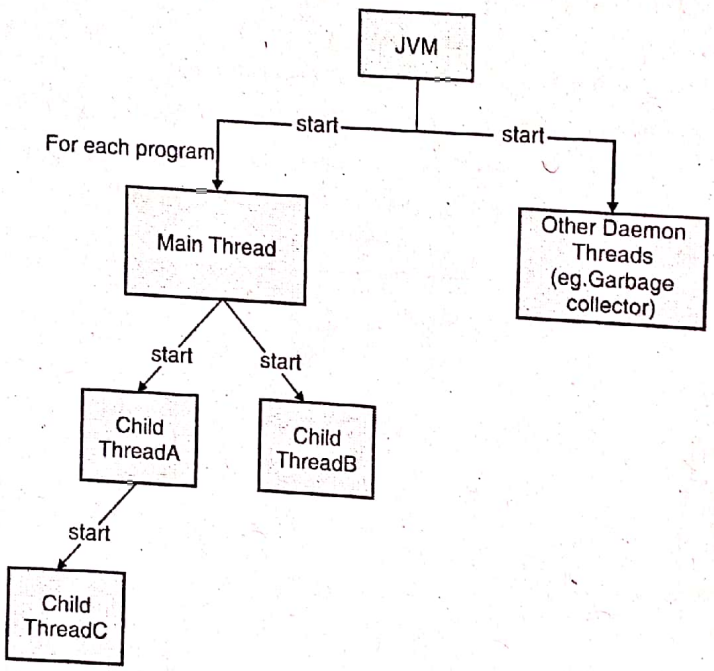

Fig. 5.2 : JVM Thread

- As shown in Fig 5.2, fundamental JVM thread is made consequently when our program is begun. To control it we should get a reference to it. This should be possible by calling the strategy current Thread( ) which is available in Thread class. This technique restores a reference to the thread on which it is called. The default need of Main thread is 5 and for all excess client threads need will be acquired from parent to child.

Q. 5 Write Program to create a Thread.

Ans. :

(4 Marks)

Creating a Thread

- As shown in Fig. 5.3, create a thread by starting up an object of type Thread. Java characterizes two manners by which this can be refined :
  - You can execute the Runnable interface.
  - You can broaden the Thread class, itself.

- The accompanying two segments take a gander at every strategy, thus.

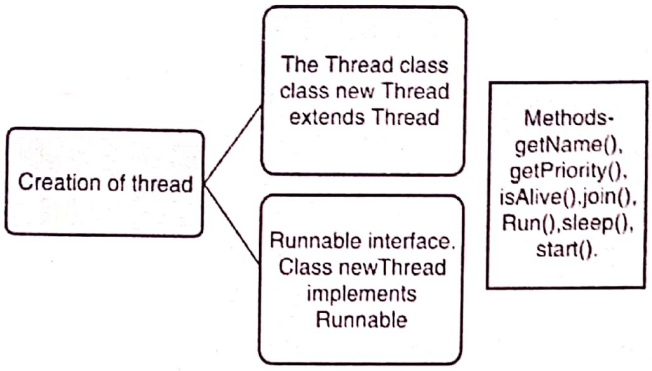

Fig. 5.3 : Create a Thread

Executing Runnable

- The casual method to make a thread is to make a class that executes the Runnable interface. Runnable modified works a unit of executable code. You can build a thread on any article that actualizes Runnable. To actualize Runnable, a class need just execute a solitary technique called run( ), which is announced this way :

```
public void run( )
```

- Inside run( ), you will characterize the code that comprises the new thread. Understand that run( ) can call different techniques, utilize different classes, and pronounce factors, much the same as the principle thread can. The solitary contrast is that run( ) sets up the section point for another, simultaneous thread of execution inside your program. This thread will end when run( ) returns.
- The subsequent method to make a thread is to make another class that expands Thread, and afterward to make an occasion of that class. The broadening class should supersede the run( ) strategy, which is the section point for the new thread. It should likewise call start( ) to start execution of the new thread. Here is the former program revised to expand Thread.

Program to create a subsequent thread by expanding thread.

```
// Create a subsequent thread by expanding Thread
class NewThread expands Thread
{
    NewThread()
    {
      //Create another, second thread
      super("Demo Thread");
      System.out.println("Child thread: " + this);
      start();//Start the thread
    }
      //This is the passage point for the subsequent thread.
      public void run()
      {
            attempt
            {
```

```
                for(int i = 5; i> 0; I--)
                {
                    System.out.println("Child Thread: " + I);
                    Thread.sleep(500);
                }
            }
            get (InterruptedException e)
            {
                System.out.println("Child interrupted.");
            }
                System.out.println("Exiting youngster thread.");
        }
    }
    class Extend Thread {
    public static void main(Thread args[]) {
    new NewThread();                    //make another thread
    attempt
    {
      for(int i = 5; i> 0; I--) {
      System.out.println("Main Thread: " + I);
      Thread.sleep(1000);
    }
    }
    get (InterruptedException e)
    {
      System.out.println("Main thread interrupted.");
    }
      System.out.println("Main thread exiting.");
}
```

Output :

```
Youngster thread : Thread[Demo Thread,5,main]
Principle Thread: 5
Youngster Thread: 5
Youngster Thread: 4
Principle Thread: 4
Youngster Thread: 3
Youngster Thread: 2
Principle Thread: 3
```

```
Youngster Thread: 1
Leaving youngster thread.
Principle Thread: 2
Principle Thread: 1
Principle thread leaving.
```

Q. 6 Write a program to create multiple threads. **(6 Marks)**

Ans. :

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs.

Program to create multiple threads.

```
// Create multiple threads
    class MyThread implements Runnable
    {
    String name;
    Thread t;
  MyThread (String thread)
    {
      name = threadname;
      t = new Thread(this, name);
     System.out.println("New thread: " + t);
     t.start();
    }
    public void run()
     {
    try
    {
      for(int i = 5; i > 0; i- -) {
       System.out.println(name + ": " + i);
       Thread.sleep(1000);
    }
    }
    catch (InterruptedException e)
    {
     System.out.println(name + "Interrupted");
    }
```

```
    System.out.println(name + " exiting.");
        }
}

        class MultiThread {
        public static void main(String args[])
        {
            new MyThread("Apple");
            new MyThread("Ball");
            new NewThread("Cat");
            try
            {
                    Thread.sleep(10000);
            }
            catch (InterruptedException e)
            {
            System.out.println("Main thread Interrupted");
            }
          System.out.println("Main thread exiting.");
          }
}
```

Output :

The output from this program is shown here :

```
New thread: Thread[Apple,5,main]
New thread: Thread[Ball,5,main]
New thread: Thread[Cat,5,main]
Apple: 5
Ball: 5
Cat: 5
Apple: 4
Ball: 4
Cat: 4
Apple: 3
Cat: 3
Ball: 3
Apple: 2
Cat: 2
```

Ball: 2

Apple: 1

Cat: 1

Ball 1

Apple exiting.

Ball exiting.

Cat exiting.

Main thread exiting.

- This strategy is characterized by Thread and its overall structure is appeared here :

  final boolean isAlive( )

- The isAlive( ) technique returns valid if the string whereupon it is called is as yet running. It returns false otherwise.
- While isAlive( ) is infrequently helpful, the technique that you will all the more generally use to trust that a string will complete is called join( ), appeared here :

  last void join( ) tosses InterruptedException

- This technique holds up until the string on which it is called ends. Its name comes from the idea of the calling string holding up until the predefined string goes along with it. Extra types of join( ) permit you to determine a most extreme measure of time that you need to trust that the predetermined string will end.
- Here is an improved form of the previous model that utilizations join( ) to guarantee that the fundamental string is the last to stop. It additionally shows the isAlive( ) strategy.

❑❑❑

Unit VI : Logical and Functional Programming

Q. 1 What are the characteristics of functional programming ? **(6 Marks)**

Ans. :

Functional Programming - Characteristics

The most prominent characteristics of functional programming are as follows :

- Functional programming languages are intended on the idea of mathematical functions that use provisional expressions and recursion to achieve computation.
- Functional programming languages do not provision flow Controls like loop declarations and conditional declarations like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages provision general notions such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Feature of Functional Paradigm

1. **Functional Programming is based on Lambda Calculus :**
   - Lambda calculus is system created by Alonzo Church to contemplate calculations with capacities.
   - It can be called as the littlest programming language of the world. It gives the meaning of what is calculable.
   - Anything that can be registered by lambda math is process able. It is identical to Turing machine in its capacity to process.
   - It gives a hypothetical system to portraying capacities and their assessment.
   - It shapes the premise of practically all current utilitarian programming languages.
2. **Pure functions :**
   - These functions have two main properties. First, they always produce the same output for same arguments irrespective of anything else. Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.
   - Later property is called immutability. The pure functions only result is the value it returns. They are deterministic.
   - Programs done using functional programming are easy to debug because pure functions have no side effect or hidden Input/Output.
   - Pure functions also make it easier to write parallel/concurrent applications.
3. **Recursion :**
   - There are no "for" or "while" loop in functional languages. Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves, until it reaches the base case.

Example of the recursive function :

```
fib(n)
if (n <= 1)
return 1;
else
return fib(n - 1) + fib(n - 2);
```

4. Referential transparency :

- In useful projects factors once characterized do not change their incentive all through the program.
- Functional programs don't have task statements. On the off chance that we need to supply some esteem, we characterize new factors all things being equal.
- This disposes of any odds of results on the grounds that any factor can be supplanted with its genuine incentive anytime of execution. Condition of any factor is steady at any moment.

Example :

```
x = x + 1        // This progressions the worth appointed to the variable x.
                 // So the expression isn't referentially straightforward
```

5. Functions are First-Class and can be Higher-Order

- First-class functions are preserved as first-class variable. The first class variables can be passed to functions as limitation, can be returned from functions or stored in data structures.
- Higher order functions are the functions that gross other functions as opinions and they can also return functions.

Example :

```
show_output(f)                  // function show_output is declared taking argument f
                                // which are another function
f();                            // calling passed function
print_gfg()                     // declaring another function
print("hello gfg");
show_output(print_gfg)          // passing function in another function
```

6. Variables are Immutable

- In functional programming, we can not adjust a variable after it's been prepared.
- We can create new variables - but we can't alter usual variables, and this actually benefits to preserve state throughout the runtime of a program.
- Once we generate a variable and set its value, we can have full self-assurance meaningful that the values of that variable will not ever alteration.

Q. 2 What are the advantages and disadvantages of functional programming ? **(4 Marks)**

Ans. :

Advantages of Functional Programming

1. Pure functions are more obvious on the grounds that they do not change any states and rely just upon the information given to them. Whatever yield they produce is the return esteem they give. Their capacity signature gives all the data about them for example their return type and their contentions.
2. The capacity of functional programming languages to regard capacities as qualities and pass them to capacities as boundaries make the code more lucid and effectively reasonable.
3. Testing and troubleshooting is simpler. Since unadulterated capacities take just contentions and produce yield, they don't create any progressions don't take information or produce some concealed yield. They utilize changeless qualities, so it gets simpler to check a few issues in projects composed utilizations unadulterated capacities.

4. It is utilized to actualize simultaneousness/parallelism on the grounds that unadulterated capacities don't change factors or some other information outside of it.
5. It embraces apathetic assessment which evades rehashed assessment on the grounds that the worth is assessed and put away just when it is required.

Disadvantages of Functional Programming

1. Sometimes writing pure functions can decrease the comprehensibility of code.
2. Writing programs in recursive style as opposed to utilizing circles can be bit scaring.
3. Writing pure functions are simple however consolidating them with rest of utilization and I/O activities is the troublesome assignment.
4. Immutable qualities and recursion can prompt diminishing in execution.

Programming Languages that support functional programming :

Haskell, JavaScript, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Racket.

Q. 3 Describe Lisp uses Prefix Notation.

Ans. : **(8 Marks)**

LISP Uses Prefix Notation

- You might have noted that LISP uses **prefix notation.**
- In the above program the + symbol works as the function name for the process of summation of the numbers.
- In prefix notation, operators are written before their operands. For example, the expression,

  a * (b + c) / d

  will be written as :

  (/ (* a (+ b c)) d)
- Let us take another example, let us write code for converting Fahrenheit temp of 60° F to the centigrade scale :
- The mathematical expression for this conversion will be :

  (60 * 9 / 5) + 32
- Create a source code file named main.lisp and type the following code in it.

  (write(+ (* (/ 9 5) 60) 32))
- When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

  140

Basic Building Blocks in LISP

- LISP programs are made up of three basic building blocks :
  - atom
  - list
  - string
- An **atom** is a number or string of contiguous characters. It includes numbers and special characters.
- A **list** is a sequence of atoms and/or other lists enclosed in parentheses.

- Following are examples of some valid lists :

```
( i am a list)
(a ( a b c) d e fgh)
(father arvind ( araddhana deshmukh))
(one two three four five six )
()
```

- A **string** is a group of characters enclosed in double quotation marks.
- Following are examples of some valid strings :

```
"I am a string"
"aba c d efg #$% ^ &!"
```

Adding Comments

The semicolon symbol (;) is used for indicating a comment line.

For Example :

```
(write-line "Araddhana") ;
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

```
Araddhana
```

Following are some of the important points :

- The basic numeric processes in LISP are +, -, * and /.
- LISP signifies a function call f(x) as (f x), for example cos(45) is written as cos 45.
- LISP terminologies are case-insensitive, cos 45 or COS 45 are same.
- LISP attempts to appraise all, including the arguments of a function. Only three types of elements are constants and always return their own value.
  - Numbers
  - The letter **t,** that stands for logical true.
  - The value **nil,** that stands for logical false, as well as an empty list.

LISP code takes the following steps :

- The reader translates the strings of characters to LISP objects or **s-expressions.**
- The evaluator describes syntax of Lisp **forms** that are constructed from s-expressions. This second level of assessment describes a syntax that controls which s-expressions are LISP forms.

  Now, a LISP forms could be.

  1. An Atom
  2. An empty or non-list
  3. Any list that has a symbol as its first element.
- The evaluator everything as a function that takes a legal LISP form as an argument and returns a value. This is the motive why we put the **LISP expression in parenthesis,** because we are conveyance the whole expression/form to the evaluator as arguments.

Naming Conventions in LISP

Name or images can comprise of quite a few alphanumeric characters other than whitespace, open and shutting brackets, twofold and single statements, oblique punctuation line, comma, colon, semicolon and vertical bar. To utilize these characters in a name, you need to utilize get away from character (\).

- A name can have digits yet not completely made of digits, since then it would be perused as a number. Essentially a name can have periods, yet can not be made totally of periods.
- Utilization of Single Quotation Mark.
- Stutter assesses everything including the capacity contentions and rundown individuals.
- On occasion, we need to take molecules or records in a real sense and don't need them assessed or treated as capacity calls.
- To do this, we need to go before the particle or the rundown with a solitary quote.
- The accompanying model exhibits this.
- Make a record named main.lisp and type the accompanying code into it.

```
(compose line "single statement utilized, it restrains assessment")
(compose '(* 2 3))
(compose line " ")
(compose line "single statement not utilized, so articulation assessed")
(compose (* 2 3))'
```

- At the point when you click the Execute catch, or type Ctrl+E, LISP executes it quickly and the outcome returned is :
- Single statement utilized, it represses assessment,

```
(* 2 3)
```

- Single statement not utilized, so articulation assessed,

```
6
```

The 'Hello World' Program

- Learning a new programming language doesn't really take off until you learn how to greet the entire world in that language, right!
- So, please create new source code file named main.lisp and type the following code in it.

```
(write-line "Hello World")
(write-line "I am Prof. Araddhana Deshmukh ! Learning LISP")
```

- When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

```
Hello World
I am I am Prof.AraddhanaDeshmukh! Learning LISP
```

- The essential numeric tasks in LISP are +, - , * and /.
- LISP addresses a capacity call f(x) as (f x), for instance cos(45) is composed as cos 45.
- LISP articulations are case-heartless, cos 45 or COS 45 are same.
- LISP attempts to assess everything, including the contentions of a capacity. Just three sorts of components are constants and consistently return their own worth,
  - Numbers
  - The letter t, that represents consistent valid.
  - The worth nil, that represents consistent bogus, just as an unfilled rundown.
- In LISP, every factor is addressed by an image. The variable's name is the name of the image and it is put away in the capacity cell of the image.

Q. 4 Write short note on :

1. Local variable
2. Global variable

(8 Marks)

Ans. :

Global Variables

Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.

Global variables are generally declared using the **defvar** construct.

Example 1

```
(defvar x 234)
(write x)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

```
234
```

Since there is no type declaration for variables in LISP, you directly specify a value for a symbol with the **setq** construct.

Example 2

```
(setq x 10)
```

- The above expression assigns the value 10 to the variable x. You can refer to the variable using the symbol itself as an expression.
- The **symbol-value** function allows you to extract the value stored at the symbol storage place.

Example 3

Create new source code file named main.lisp and type the following code in it.

```
(setq x 10)
(setq y 20)
(format t "x = ~2d y = ~2d ~%" x y)

(setq x 100)
(setq y 200)
(format t "x = ~2d y = ~2d" x y)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

```
x = 10 y = 20
x = 100 y = 200
```

Local Variables

- Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.
- Like the global variables, local variables can also be created using the **setq** construct.
- There are two other constructs - **let** and **prog** for creating local variables.
- The let construct has the following syntax :

```
(let ((var1 val1) (var2 val2).. (varn valn))<s-expressions>)
```

- Where var1, var2, ...varn are variable names and val1, val2, ... valn are the initial values assigned to the respective variables.
- When **let** is executed, each variable is assigned the respective value and lastly the *s-expression* is evaluated. The value of the last expression evaluated is returned.
- If you don't include an initial value for a variable, it is assigned to **nil.**

Example 1

Create new source code file named main.lisp and type the following code in it.

```
(let ((x 'a) (y 'b)(z 'c))
(format t "x = ~a y = ~a z = ~a" x y z))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

```
x = A y = B z = C
```

- The **prog** construct also has the list of local variables as its first argument, which is followed by the body of the **prog,** and any number of s-expressions.
- The **prog** function executes the list of s-expressions in sequence and returns nil unless it encounters a function call named **return.** Then the argument of the **return** function is evaluated and returned.

Example 2

Create new source code file named main.lisp and type the following code in it.

```
(prog ((x '(a b c))(y '(1 2 3))(z '(p q 10)))
(format t "x = ~a y = ~a z = ~a" x y z))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

```
x = (A B C) y = (1 2 3) z = (P Q 10)
```

Q. 5 Define functions in LISP.

Ans. : **(4 Marks)**

Defining Functions in LISP

The macro named defun is used for defining functions. The defun macro needs three arguments :

1. Name of the function
2. Parameters of the function
3. Body of the function

Syntax for defun is :

```
(defun name (parameter-list) "Optional documentation string." body)
```

Let us illustrate the concept with simple examples.

Example 1

Let's write a function named *averagenum* that will print the average of four numbers. We will send these numbers as parameters.

Create a new source code file named main.lisp and type the following code in it.

```
(defun averagenum (n1 n2 n3 n4)
  (/ ( + n1 n2 n3 n4) 4)
)
(write(averagenum 10 20 30 40))
```

When you execute the code, it returns the following result :

```
25
```

Example 2

Let's define and call a function that would calculate the area of a circle when the radius of the circle is given as an argument.

Create a new source code file named main.lisp and type the following code in it.

```
(defun area-circle(rad)
    "Calculates area of a circle with given radius"
    (terpri)
    (format t "Radius: ~5f" rad)
    (format t "~%Area: ~10f" (* 3.141592 rad rad))
)
(area-circle 10)
```

When you execute the code, it returns the following result :

```
Radius: 10.0
Area: 314.1592
```

Q. 6 List and explain predicates in LISP. **(8 Marks)**

Ans. : Predicates in LISP

Predicates are functions that test their arguments for some specific conditions and returns nil if the condition is false, or some non-nil value is the condition is true.

The Table 6.1 shows some of the most commonly used predicates :

Table 6.1 : Predicates in LISP

Sr. No.	Predicate & Description
1	atom It takes one argument and returns t if the argument is an atom or nil if otherwise.
2	equal It takes two arguments and returns t if they are structurally equal or nil otherwise.
3	eq It takes two arguments and returns t if they are same identical objects, sharing the same memory location or nil otherwise.
4	eql It takes two arguments and returns t if the arguments are eq, or if they are numbers of the same type with the same value, or if they are character objects that represent the same character, or nil otherwise.
5	evenp It takes one numeric argument and returns t if the argument is even number or nil if otherwise.
6	oddp It takes one numeric argument and returns t if the argument is odd number or nil if otherwise.
7	zerop It takes one numeric argument and returns t if the argument is zero or nil if otherwise.

Sr. No.	Predicate & Description
8	null It takes one argument and returns t if the argument evaluates to nil, otherwise it returns nil.
9	listp It takes one argument and returns t if the argument evaluates to a list otherwise it returns nil.
10	greaterp It takes one or more argument and returns t if either there is a single argument or the arguments are successively larger from left to right, or nil if otherwise.
11	lessp It takes one or more argument and returns t if either there is a single argument or the arguments are successively smaller from left to right, or nil if otherwise.
12	numberp It takes one argument and returns t if the argument is a number or nil if otherwise.
13	symbolp It takes one argument and returns t if the argument is a symbol otherwise it returns nil.
14	integerp It takes one argument and returns t if the argument is an integer otherwise it returns nil.
15	rationalp It takes one argument and returns t if the argument is rational number, either a ratio or a number, otherwise it returns nil.
16	floatp It takes one argument and returns t if the argument is a floating point number otherwise it returns nil.
17	realp It takes one argument and returns t if the argument is a real number otherwise it returns nil.
18	complexp It takes one argument and returns t if the argument is a complex number otherwise it returns nil.
19	characterp It takes one argument and returns t if the argument is a character otherwise it returns nil.
20	stringp It takes one argument and returns t if the argument is a string object otherwise it returns nil.
21	arrayp It takes one argument and returns t if the argument is an array object otherwise it returns nil.
22	packagep It takes one argument and returns t if the argument is a package otherwise it returns nil.

Example 1 : Create a new source code file named main.lisp and type the following code in it.

```
(write (atom 'abcd))
(terpri)
(write (equal 'a 'b))
(terpri)
(write (evenp 10))
```

```
(terpri)
(write (evenp 7 ))
(terpri)
(write (oddp 7 ))
(terpri)
(write (zerop 0.0000000001))
(terpri)
(write (eq 3 3.0 ))
(terpri)
(write (equal 3 3.0 ))
(terpri)
(write (null nil ))
```

When you execute the code, it returns the following result :

```
T
NIL
T
NIL
T
NIL
NIL
NIL
T
```

Example 2 : Create a new source code file named main.lisp and type the following code in it.

```
(defun factorial (num)
  (cond ((zeropnum) 1)
        (t ( * num (factorial (- num 1))))
  )
)
(setq n 6)
(format t "~% Factorial ~d is: ~d" n (factorial n))
```

When you execute the code, it returns the following result :

```
Factorial 6 is: 720
```

Q. 7 Explain Conditionals in LISP. **(4 Marks)**

Ans. : Conditionals in LISP

Conditionals use predicates to direct the course of evaluation.

(if test-expression then-expression else-expression)

- test-expression is evaluated.
- if the value is not nil, then-expression is evaluated and its value is the value of the **if**.

- otherwise, else-expression is evaluated and its value is the value of the **if.**

```
;;;
;;; count-elements takes a list as argument and
;;; returns the number of elements in the list
;;;
(defun count-elements (L)(if L (+ 1 (count-elements (rest L)) 0))
```

- (when test-expression then-expression)
- equivalent to(if test-expression then-expression NIL)
- (unless test-expression else-expression)
- equivalent to(if test-expression NIL else-expression)

The **cond** construct in LISP is most commonly used to permit branching.

Syntax for cond is :

```
(cond   (test1 action1)
        (test2 action2)
        ...
        (testn actionn))
```

- Each clause within the cond statement consists of a conditional test and an action to be performed.
- If the first test following cond, test1, is evaluated to be true, then the related action part, action1, is executed, its value is returned and the rest of the clauses are skipped over.
- If test1 evaluates to be nil, then control moves to the second clause without executing action1, and the same process is followed.
- If none of the test conditions are evaluated to be true, then the cond statement returns **nil.**

Example : Create a new source code file named main.lisp and type the following code in it.

```
(setq a 10)
(cond ((> a 20)
(format t "~% a is greater than 20"))
(t (format t "~% value of a is ~d " a)))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is :

```
value of a is 10
```

Please note that the **t** in the second clause ensures that the last action is performed if none other would.

Q. 8 Explain Recursion in Lisp detail.

Ans. : Recursion in LISP **(6 Marks)**

In pure **Lisp** there is no looping; **recursion** is used instead. A **recursive** function is defined in terms of :

1. One or more base cases.
2. Invocation of one or more simpler instances of itself.

Note that recursion is directly related to mathematical induction. An inductive proof has :

1. A basis clause
2. We can implement a function to compute factorials using recursion :

```
(defun factorial (N)
 "Compute the factorial of N."
 (if (= N 1)
     1
     (* N (factorial (- N 1)))))
```

The if form checks if N is one and returns one if that is the case, or else returns N * (N - 1)!. Several points are worth noting.

Multiple Recursions

Recall the definition of Fibonacci numbers :

Fib(n) = 1 for n = 0 or n = 1

Fib(n) = Fib(n–1) + Fib(n–2) for n > 1

This definition can be directly translated to the following LISP code :

```
(defun fibonacci (N)
 "Compute the N'th Fibonacci number."
 (if (or (zerop N) (= N 1))
     1
     (+ (fibonacci (- N 1)) (fibonacci (- N 2)))))
```

Lists had been the most important and the primary composite data structure in traditional LISP. Present day's Common LISP provides other data structures like, vector, hash table, classes or structures.

Lists are single linked lists. In LISP, lists are constructed as a chain of a simple record structure named **cons** linked together.

The cons Record Structure

- A **cons** is a record structure containing two components called the **car** and the **cdr.**
- Cons cells or cons are objects are pairs of values that are created using the function cons.
- The cons function takes two arguments and returns a new cons cell containing the two values. These values can be references to any kind of object.
- If the second value is not nil, or another cons cell, then the values are printed as a dotted pair enclosed by parentheses.
- The two values in a cons cell are called the car and the cdr. The car function is used to access the first value and the cdr function is used to access the second value.

Example : Create a new source code file named main.lisp and type the following code in it.

```
(write (cons 1 2))
(terpri)
(write (cons 'a 'b))
(terpri)
(write (cons 1 nil))
(terpri)
(write (cons 1 (cons 2 nil)))
(terpri)
```

```
(write (cons 1 (cons 2 (cons 3 nil))))
(terpri)
(write (cons 'a (cons 'b (cons 'c nil))))
(terpri)
(write ( car (cons 'a (cons 'b (cons 'c nil)))))
(terpri)
(write ( cdr (cons 'a (cons 'b (cons 'c nil)))))
```

When you execute the code, it returns the following result :

```
(1 . 2)
(A . B)
(1)
(1 2)
(1 2 3)
(A B C)
A
(B C)
```

The above example shows how the cons structures could be used to create a single linked list, e.g., the list (A B C) consists of three cons cells linked together by their cdrs.

Q. 9 Write short note on lists in Lisp.

Ans. : Lists in LISP **(8 Marks)**

- Although cons cells can be used to create lists, however, constructing a list out of nested **cons** function calls can't be the best solution. The **list** function is rather used for creating lists in LISP.
- The list function can take any number of arguments and as it is a function, it evaluates its arguments.
- The **first** and **rest** functions give the first element and the rest part of a list. The following examples demonstrate the concepts.

Example 1 : Create a new source code file named main.lisp and type the following code in it.

```
(write (list 1 2))
(terpri)
(write (list 'a 'b))
(terpri)
(write (list 1 nil))
(terpri)
(write (list 1 2 3))
(terpri)
(write (list 'a 'b 'c))
(terpri)
(write (list 3 4 'a (car '(b . c)) (* 4 -2)))
(terpri)
(write (list (list 'a 'b) (list 'c 'd 'e)))
```

When you execute the code, it returns the following result :

```
(1 2)
(A B)
(1 NIL)
(1 2 3)
(A B C)
(3 4 A B -8)
((A B) (C D E))
```

Example 2 : Create a new source code file named main.lisp and type the following code in it.

```
(defun my-library (title author rating availability)
  (list :title title :author author :rating rating :availabilty availability)
)
(write (getf (my-library "Hunger Game" "Collins" 9 t) :title))
```

When you execute the code, it returns the following result :

```
"Hunger Game"
```

List Manipulating Functions

The Table 6.13.1 provides some commonly used list manipulating functions.

Table 6.13.1 : List Manipulating Functions

Sr. No.	Function & Description
1	car It takes a list as argument and returns its first element.
2	cdr It takes a list as argument and returns a list without the first element
3	cons It takes two arguments, an element and a list and returns a list with the element inserted at the first place.
4	list It takes any number of arguments and returns a list with the arguments as member elements of the list.
5	append It merges two or more list into one.
6	last It takes a list and returns a list containing the last element.
7	member It takes two arguments of which the second must be a list, if the first argument is a member of the second argument and then it returns the remainder of the list beginning with the first argument.
8	reverse It takes a list and returns a list with the top elements in reverse order.

Please note that all sequence functions are applicable to lists.

Example 3 : Create a new source code file named main.lisp and type the following code in it.

```
(write (car '(a b c d e f)))
(terpri)
(write (cdr '(a b c d e f)))
(terpri)
(write (cons 'a '(b c)))
(terpri)
(write (list 'a '(b c) '(e f)))
(terpri)
(write (append '(b c) '(e f) '(p q) '() '(g)))
(terpri)
(write (last '(a b c d (e f))))
(terpri)
(write (reverse '(a b c d (e f))))
```

When you execute the code, it returns the following result :

```
A
(B C D E F)
(A B C)
(A (B C) (E F))
(B C E F P Q G)
((E F))
((E F) D C B A)
```

Concatenation of car and cdr Functions

- The **car** and **cdr** functions and their combination allows extracting any particular element/ member of a list.
- However, sequences of car and cdr functions could be abbreviated by concatenating the letter a for car and d for cdr within the letters c and r.
- For example we can write cadadr to abbreviate the sequence of function calls - car cdr carcdr.
- Thus, (cadadr '(a (c d) (e f g))) will return d.

Example 4 : Create a new source code file named main.lisp and type the following code in it.

```
(write (cadadr '(a (c d) (e f g))))
(terpri)
(write (caar (list (list 'a 'b) 'c)))
(terpri)
(write (cadr (list (list 1 2) (list 3 4))))
(terpri)
```

When you execute the code, it returns the following result :

```
D
A
(3 4)
```

□□□