CS 5 (Programming in Java) Unit II Long Answer Type

Q) Explain how to create and use package in Java program.

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Advantage of Java Package:-

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

2) Java package provides access protection.

3) Java package removes naming collision.

Types of Packages:-

Package in java can be categorized in two

1. built-in package
2. user-defined package

1.Built-in Packages

The Java API is a library of prewritten classes, that are free to use.

There are many built-in packages. Some of them are as follows :

Package Name	Functionality
java.lang	Basic language fundamentals
java.util	Utility classes & data structure classes
java.io	File handling operations
java.math	Mathematical functions
java.sql	JDBC to access databases
java.awt	AWT for native GUI components
javax.swing	Lightweight GUI components

Importing a Package or Class:

To use a class or a package from the library, we need to use the import keyword:

Syntax

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

Example:- To use the Scanner class, **which is used to get user input**, write the following code:

a) **import java.util.Scanner;**

   Here:
   → **java** is a top level package
   → **util** is a sub package
   → **Scanner** is a class which is in the sub package **util**.

b) **import java.util.*;**

Example

Using the Scanner class to get user input:

```
import      java.util.Scanner;

class MyClass
{
  public static void main(String[] args)
  {
    Scanner sc = new Scanner(System.in);
    int n;

    System.out.println("Enter n:");
    n = sc.nextInt( );
    System.out.println("n= " + n);

    //String userName = sc.nextLine();
```

```
        //System.out.println("Username is: " + userName);
    }
  }
```

2.User-defined Packages

- A user defined package is one which is developed by java programmers.
- Any class or interface is commonly used by many java programmers that class or interface must be placed in packages.

Creating a userdefined package:

Syntax:

package pack1[.pack2[.pack3……[.packn]…..]];

Here, **package** is a keyword for creating user defined packages, pack1 represents top level package and pack2 to packn represent sub packages.

Examples:

```
package   p1;
package   p1.p2;
```

RULE:

Whenever we create user defined package in a Java program, we must use **package** statement . It must be the first statement in the program.

<u>STEPS for developing a PACKAGE</u>:

1. Choose the appropriate package name.
2. Create a class or an interface with modifier must be **public**.
3. The modifier of Constructors of the class must be **public**.
4. The modifier of the methods of the class or interface must be **public**.
5. Give the file name as class name or interface name with extension **.java**

For example:

```
//Shapes.java

package  drawingTools;

public class Shapes {

  public void line()

  {

    System.out.println("Line is drawn");

  }

  public void circle()

  {

    System.out.println("Circle is drawn");

  }

}
```

Using package classes and interfaces in another java program :

In order to refer package classes and interfaces in JAVA we have two approaches, they are

1. Using import statement:
2. Using fully qualified name

import keyword is used to use classes and interfaces of a package.

Syntax -1:

import pack1 [.pack2 [………[.packn]]].*;

Examples:

```
import   p1.*;
import   p1.p2.*;
import   p1.p2.p3.*;
```

- With first example, we can access all the classes and interfaces of package p1 only, but not its sub packages p2 and p3 classes and interfaces.
- With second example, we can access the classes and interfaces of package p2 only, but not p1 and p3 classes and interfaces.
- With third example, we can access the classes and interfaces of package p3 only, but not p1 and p2 classes and interfaces.

Syntax-2:

import pack1 [.pack2 […………[.packn]]].classname/interfacename;

Example:

```
import p1.c1;
import p1.p2.c3;
```

With above examples, we can access c1 class of package p1 only, but not other classes and interfaces of p1 package, p2 package and p3 package.

Example :

```
import drawingTools.*;

class MyPicture {

  public static void main(String args[]) {

   Shapes so = new Shapes();
   so.circle();

   so.line();

  }

}
```

Q) Explain about wrapper classes in Java.

Object oriented programming is all about objects.

The eight primitive data types - byte, short, int, long, float, double, char and Boolean - are not objects,

To make Java a pure OOP language, we wrapper classes are introduced.

The **wrapper classes in Java** provide the mechanism *to convert primitive datatypes into objects and objects into primitive types*.

For eg., int to Integer and Integer to int, etc.

The eight classes of the *java.lang* package are known as wrapper classes.

Primitive Type	**Wrapper class**
Boolean	Boolean
Char	Character
Byte	Byte
Short	Short
Int	Integer

Long	Long
Float	Float
Double	Double

Boxing

The conversion of primitive data type into its corresponding wrapper class is known as boxing, for example, byte to Byte, int to Integer.

Example :

```
public void boxing( )

{

        int a = 12;

        Integer b = Integer.valueOf(a);  //  explicit  boxing

        Integer c = a;    //  autoboxing

}
```

Autoboxing

Since Java 5 . we do not need explicit conversion from primitive to object. The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing.

Example :

```
public void autoboxing( )

{

        int a = 12;

        Integer b = a;
```

}

Unboxing

The automatic conversion of object (wrapper type) into its corresponding primitive type is known as unboxing.

Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Example :

```
public  void  unboxing( )
{
        Integer  a = 5;
        int  b = a;    //  auto  unboxing
        int  c = a.intValue( );    //  explicit  unboxing
}
```

Q) Explain about String class and its methods in detail.

String is a sequence of characters.
E.g. “Hello” is a string of 5 characters.

In java, String is an immutable object which means it is constant and cannot be changed once it has been created.

Creating a String

There are two ways to create a String:

1. String literal
2. Using new keyword

1. String literal

Assign a String literal to a String instance:

```
String   str1 = "Welcome";
String   str2 = "Welcome";
```

The compiler creates a string object having the string literal (“Welcome”) and assigns it to the string reference variable(str1, str2).

If the object already exists in the memory, compiler does not create a new object rather it assigns the same old object to the new reference variable.

That means even though we have two string reference variables above(str1 and str2), compiler creates only one string object (having the value “Welcome”) and assigns the same to both the reference variables.

2. **Using New Keyword**

We can create strings using new operator like shown below to overcome the problem in the above style of string creation.

```
String str1 = new String("Welcome");
String str2 = new String("Welcome");
```

In this case compiler would create two different objects in memory having the same string.

String class methods

The java.lang.String class provides a lot of methods to work on string.

1. **length()** method returns length of the string.

```
String s="Sachin";
System.out.println(s.length());    //6
```

2. **toUpperCase()** method converts string into uppercase letters
3. **toLowerCase()** method converts string into lowercase letters.

```
String s="Sachin";
System.out.println(s.toUpperCase());       //SACHIN
System.out.println(s.toLowerCase());      //sachin
```

4. **trim()** method eliminates white spaces before and after string.

```
String s="  Sachin  ";
System.out.println(s);                     //  Sachin
System.out.println(s.trim());              //Sachin
```

5. **startsWith(prefix)** method tests whether string begins with prefix or not.

6. **endsWith(sufffix)** method tests whether string ends with sufffix or not.

```
String s="Sachin";
 System.out.println(s.startsWith("Sa"));        //true
 System.out.println(s.endsWith("n"));          //true
```

7. **charAt()** method returns a character at specified index.

```
String s="Sachin";
System.out.println(s.charAt(0));                  //S
System.out.println(s.charAt(3));                  //h
```

8. **valueOf()** method coverts given type such as int, double into string.

```
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);                 //  1010
```

9. **replace()** method replaces all occurrence of first parameter with second parameter.

```
String s1="Java is a OOPL. Java is platform.";
String s2=s1.replace("Java","Python");      //replaces all occurrences of "Java" to "Python"
System.out.println(s2);
```

Output:

Python is a OOPL. Python is platform.

10. **equals()** method compares two strings. If they are equal, the method returns true else false.

```
    String s1= new String("Hello");
    String s2= new String("Hello");
    System.out.println("s1 equals to s2:"+s1.equals(s2));
```

Output :-

```
s1 equals to s2 : true
```

Q) Explain about StringBuffer class.

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Constructors of StringBuffer class :

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class:

1. **append()** method concatenates the given argument with this string.

```
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");                    //now original string is changed
System.out.println(sb);               //prints Hello Java
```

2.replace() method replaces the given string from the specified beginIndex to endIndex-1.

```
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);                //prints HJavalo
```

3.reverse() method reverses the current string.

```
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);             //prints olleH
```

4.capacity() method returns the current capacity of the buffer.

The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());               //default 16
sb.append("Hello");
System.out.println(sb.capacity());             //16
sb.append("java is my favourite language");
System.out.println(sb.capacity());            //(16*2)+2=34 i.e (oldcapacity*2)+2
```

Q) Explain about exceptions and exception handling mechanism in detail.

Exception Handling in Java is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the program can be maintained.

Exception

An exception is an abnormal condition.
An Exception is an unwanted event that interrupts the normal flow of the program.

Exception Handling

When an exception occurs program execution is terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Example of system generated exception :

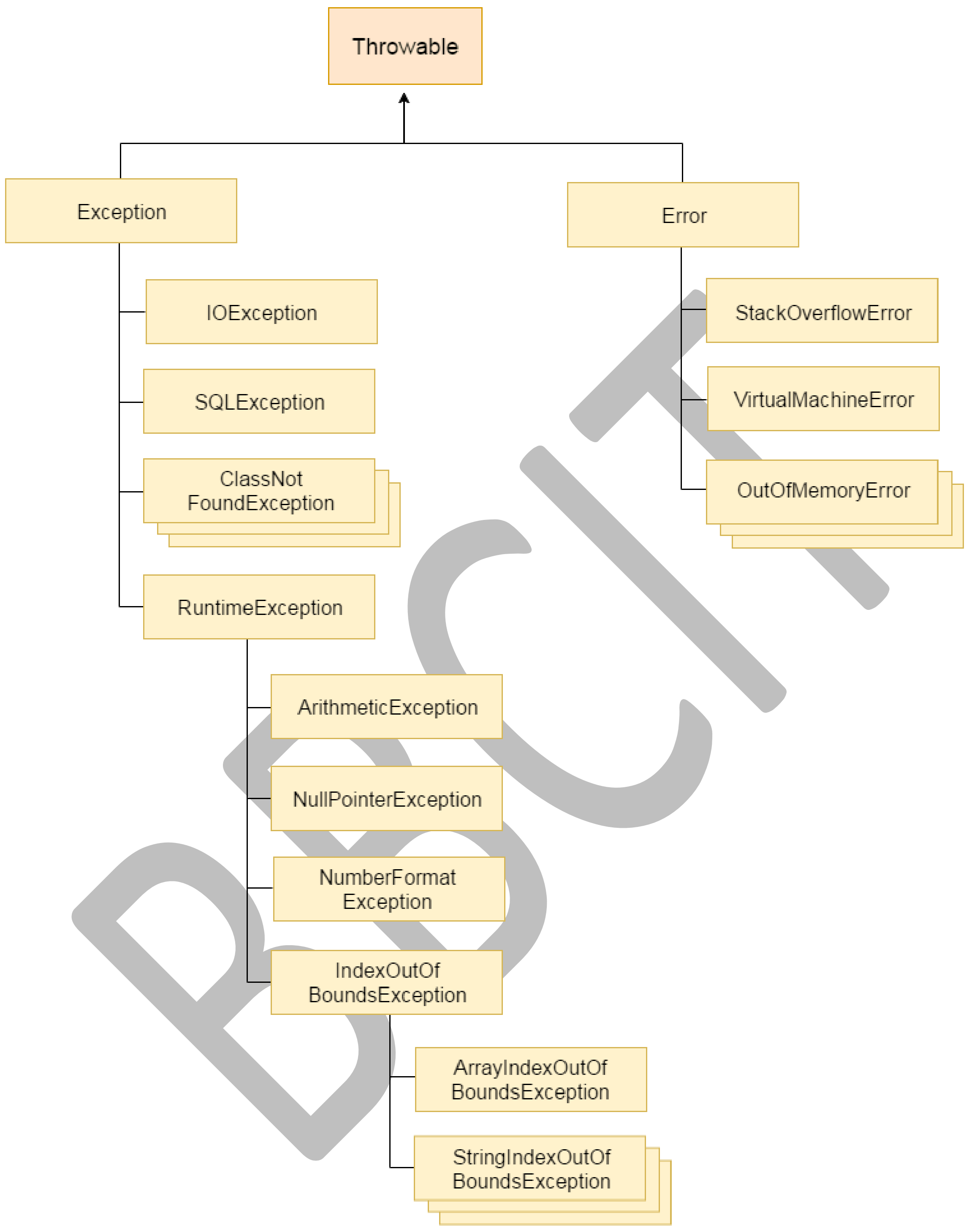
```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
 ExceptionDemo : The class name
 main : The method name
 ExceptionDemo.java : The filename
 java:5 : Line number
```

Advantage of exception handling

To maintain the normal flow of the program**.**

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException
StackOverflowError
VirtualMachineError
OutOfMemoryError

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Unchecked Exception
Checked Exception
Error
Exception Handling

<u>1.Checked Exception</u>

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

```
class ExceptionDemo
{
  public static void main(String args[])
  {
      try
      {
             int data=100/0;
      }
      catch(ArithmeticException e)
      {
             System.out.println(e);
      }
      System.out.println("rest of the code...");
   }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Q) Explain about Custom(Userdefined) Exceptions with example.

If we are creating our own Exception that is known as custom exception or user-defined exception.

Custom exceptions are used to customize the exception according to user need.

By the help of custom exception, we can have our own exception and message.

Example:

```
import java.util.Scanner;

class IllegalLicenceException extends Exception {

  String msg;

  IllegalLicenceException(String s)
  {
     msg = s;
  }
  public String toString(){
          return ("IllegalLicenceException Occurred: "+msg) ;
  }
}
class UserException {
 public static void main(String args[]) {
   Scanner sc = new Scanner(System.in);
   System.out.println("Enter age:");
   int age = sc.nextInt();
    try
    {
                if (age<18)
                        throw   new    IllegalLicenceException("Person is minor.");
                System.out.println("You are eligible for driving");
        }
        catch(IllegalLicenceException e)
        {
                System.out.println(e);
        }
 }
}
```

Output :-

```
                    Run  1:
Enter age:
21
You are eligible for driving
                    Run  2:
Enter age:
15
IllegalLicenceException Occurred: Person is minor.
```

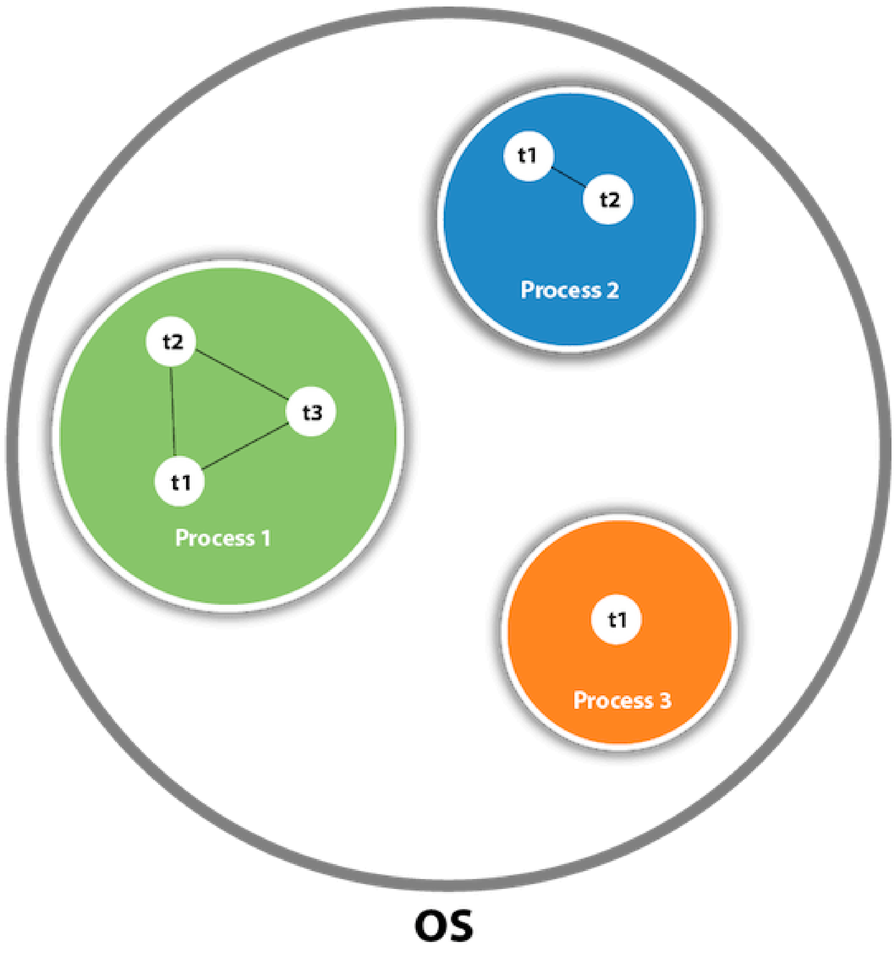
Q) Explain about multithreading in Java .

A **thread** is a light-weight sub-process, smallest part of processing.

One thread can **run concurrently** with the other smallest parts(other threads) of the same process.

The process of executing multiple threads simultaneously is known as **multithreading.**

Threads are independent that's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads.



Advantages of Java Multithreading

1. Multithreading provides simultaneous execution of two or more parts of a program to **maximum utilize the CPU time**.

2. All threads of a process share the common memory.

Thread Life-Cycle / States of a Thread :

A thread can be in one of the following states:

NEW – The thread is created but not started.

```
ThreadDemo  obj=new  ThreadDemo( );
```

RUNNABLE – The thread is ready for execution by JVM. But it is waiting in the queue for getting the processor. A thread enters this state after start() method is called.

```
obj.start( );
```

NOT RUNNABLE - A thread is said to be in NOT RUNNABLE state if it is in any of the following 3 states - WAITING, TIMED_WAITING, BLOCKED.

WAITING – A thread is waiting indefinitely for another thread to perform a particular action i.e., notify. Threads enter this stated by calling

```
obj. wait( )     or   Thread. join( )
```

***TIMED_WAITING* -** A thread is waiting for another thread to perform an action up to a specified waiting time. Threads enter this stated by calling

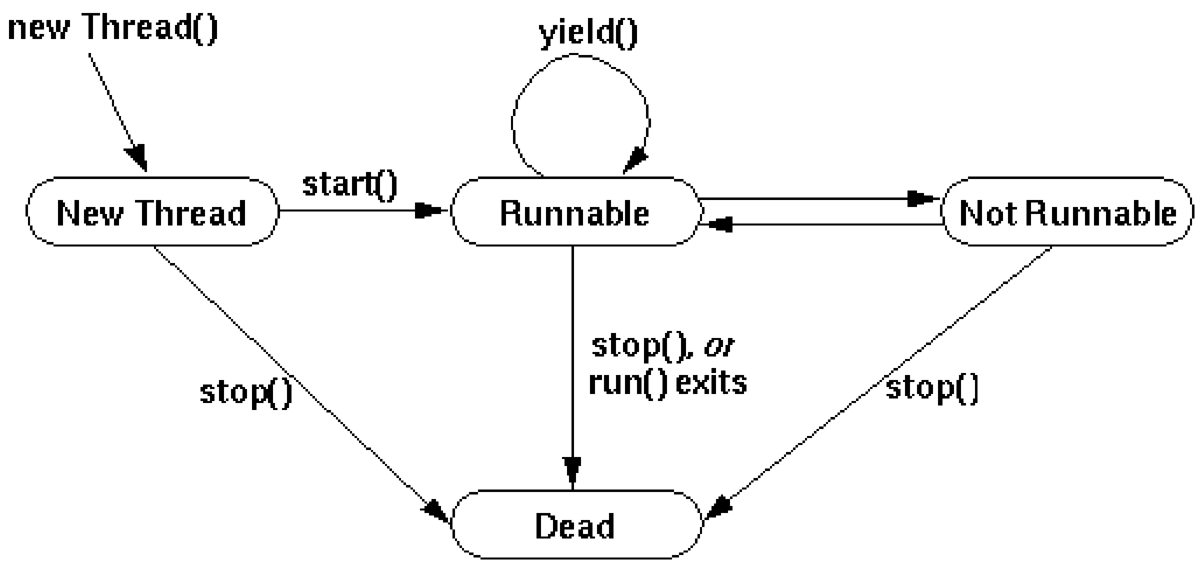
```
obj. wait( )     or   Thread. join( )       or     Thread.sleep( )
```

***BLOCKED* -** A thread is waiting for a resource that is held with another thread. Threads enter this stated by calling

```
obj. wait( )
```

TERMINATED – This state is reached when the thread has finished its execution.

A thread can be in only one state at a given point of time.



Thread class

Java provides **Thread class** in java.lang package to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Some of them are given below:

• getName()	: It is used for Obtaining a thread's name
• getPriority()	: Obtain a thread's priority
• isAlive()	: Determine if a thread is still running
• join()	: Wait for a thread to terminate
• run()	: Entry point for the thread
• sleep()	: suspend a thread for a period of time
• start()	: start a thread by calling its run() method

Creating a thread

There are two ways to create a thread in Java:
1) By extending Thread class.
2) By implementing Runnable interface.

Q) Explain about main thread.

Even though programmer does not create a thread, every Java program has a thread called the **main thread**.

When a Java program starts executing, JVM created the **main thread** and calls the main( ) method.

The **main thread** spawns the other threads. These spawned threads are called **child threads**.

The **main thread** is always the last to finish executing because it is responsible for releasing the resources such as network connections.

The programmer can control the **main thread** using the **currentThread( )** method. This method is defined in java.lang.Thread class.

Sample Program:

```
class  MainThreadDemo
{
    public static void main(String args[])
    {
         Thread   obj=Thread.currentThread( );
         System.out.println("Current  thread: " + obj);
         obj.setName("New Thread");
         System.out.println("Renamed  thread: " + obj);
    }
}
```

Output :-

```
Current thread:  Thread[main, 5, main]
Renamed thread: Thread[New Thread, 5, main]
```

Q) Explain how to create a thread using Thread class and Runnable Interface.

Or

Explain the methods to create a thread with examples.

There are two ways to create a thread in Java:
1) By extending Thread class.
2) By implementing Runnable interface.

Method 1: Thread creation by extending Thread class

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

Steps to follow for thread creation:

1. Create a subclass derived from **Thread** class
2. Override(i.e., Define) the **run( )** method
3. Create thread object
4. Call the **start( )** method.

Example :

```
class Multi extends Thread               // step 1
{
      public void run()                     //  step  2
     {
             System.out.println("thread is running...");
     }

     public static void main(String args[])
    {
             Multi t1=new Multi();          //  step  3
             t1.start();                    //  step  4
    }
}
```

Output:

```
thread is running...
```

Method 2: Thread creation by implementing Runnable Interface

Runnable interface is defined in java.lang package. It is a public interface. There must be a class that must implement Runnable interface.

Steps to follow for thread creation:

1. Create a class that implements **Runnable** interface
2. Override(i.e., Define) the **run( )** method
3. Create thread object
4. Call the **start( )** method.

```
class Multi implements Runnable                // step 1
{
   public void run()                          // step 2
   {
         System.out.println("thread is running...");
   }

   public static void main(String args[])
  {
         Multi  m1=new Multi();
         Thread t1 =new Thread(m1);           // step 3
          t1.start();                         // step 4
    }
}
```

Output:

```
thread is running...
```

Q) Explain about thread synchronization.

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**.

Synchronization is a mechanism *to control the access of multiple threads to any shared resource*.

Understanding the problem without Synchronization:

In this example, there is no synchronization, so output is inconsistent.

```
class Table
{
   void printTable(int n)      //method not synchronized
   {
      for(int i=1;i<=5;i++)
      {
                System.out.println(n*i);
           try
           {
                 Thread.sleep(400);
           }
           catch(Exception e)
          {
                 System.out.println(e);
           }
      }

  }
}

class MyThread1 extends Thread
{
     Table t;

     MyThread1(Table t)
    {
         this.t=t;
    }

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

class MyThread2 extends Thread
{
     Table t;

     MyThread2(Table t)
    {
        this.t=t;
    }

    public void run()
   {
       t.printTable(100);
    }
}

class NoSynch
{
     public static void main(String args[])
     {
         Table obj = new Table();
         MyThread1 t1=new MyThread1(obj);
         MyThread2 t2=new MyThread2(obj);
         t1.start();
          t2.start();
     }
}
```

Output: 5

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

synchronized method:

The synchronization keyword in java creates a block of code referred to as critical section.

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

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

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

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

  }
}

class MyThread1 extends Thread
{
     Table t;

     MyThread1(Table t)
    {
        this.t=t;
    }

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

class MyThread2 extends Thread
{
     Table t;

     MyThread2(Table t)
    {
        this.t=t;
    }

    public void run()
   {
       t.printTable(100);
    }
}

class SyncDemo
{
     public static void main(String args[])
     {
        Table obj = new Table();
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
     }
}
```

Output: 5

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

About Java I/O

- The two most important parts of a computer are input and output.

- java.io package provides separate classes & interfaces for reading & writing data.
- So, if we use any I/O class, we need to write **import java.io.*;** statement at the top of the program.

- Data may be byte data or character data.

- Java I/O facility is based on streams.

- Stream is a continuous flow of data.

- Java I/O package has both byte stream classes & character stream classes.

- Byte stream classes handle reading & writing of bytes to files, sockets, etc.

- Character stream classes handle reading & writing of characters to files, sockets, etc.

- java.io package contains two top level byte stream abstract classes:
    - java.io.InputStream (for reading bytes data)
    - java.io.OutputStream (for writing bytes data)

- It also contains two top level character stream abstract classes:
    - java.io.Reader (for reading character data)
    - java.io.Writer (for writing character data)

- The subclasses of the above classes are actually used for reading & writing data.

Q) Explain about File class and its methods in detail.
File Class

The File class from the java.io package, allows us to work with files.
This class is used to know the file properties such as:

- pathname of a file
- a file exists or not
- given a name whether it is a file or a directory
- length of a file

<u>Methods of File class:</u>

The File class has many useful methods for creating and getting information about files. Some of them are :

Return Type	**Method**	**Description**
boolean	createNewFile()	It atomically creates a new, empty file if and only if a file with this name does not exist.
boolean	canWrite()	It tests whether the file can be modified or not
boolean	canExecute()	It tests whether the file can be executed or not
boolean	canRead()	It tests whether the file can be read or not
boolean	isDirectory()	It tests whether the file is a directory or not.
boolean	isFile()	It tests whether the file is a normal file or not.
String	getName()	It returns the name of the file.
boolean	exists()	Tests whether the file exists
long	length()	Returns the size of the file in bytes

boolean	mkdir()	It creates the directory named by this abstract pathname.

Q) Explain about FileInputStream & FileOutputStream classes with suitable programs.

FileInputStream Class

FileInputStream class is used for reading data from a file.
It is used for reading byte-oriented data (streams of raw bytes) such as images, audio, video etc.
It is derived from InputStream class.

Useful methods :

Method	Description
int available()	Returns the number of bytes that can be read.
int read()	Reads a byte of data from the input stream.
int read(byte[] b)	Reads up to **b.length** bytes of data from the input stream and stores it in ***b*** array.
int read(byte[] b, int off, int len)	Reads up to **len** bytes of data from the input stream into ***b*** array starting at ***off***.
long skip(long x)	Skips over and discards x bytes of data from the input stream.
void finalize()	Ensures that the close method is called when there are no more reference to the file input stream.
void close()	Closes the stream.

Sample Program: To read a single byte from a file

```
import java.io.FileInputStream;

 class FISDemo
{
    public static void main(String args[])
    {
        try
        {
             FileInputStream fin=new FileInputStream("D:\\test.txt");
             int i=fin.read();
             System.out.print((char)i);
             fin.close();
        }
        catch(Exception e)
        {
              System.out.println(e);
        }
   }
}
```

Contents of test.txt :
Welcome to Java.

Output:

```
W
```

FileOutputStream Class

FileOutputStream class is used for writing data to a file.
It is used for writing byte-oriented data .
It is derived from OutputStream class.

Useful methods :

Method	Description
void finalize()	Cleans up the connection with the file output stream.
void write(byte[] ary)	Writes **ary.length** bytes from the byte array ary to the file output stream.
void write(byte[] ary, int off, int len)	Writes **len** bytes from the byte array ary starting at **off** to the file output stream.
void write(int b)	Writes the specified byte to the file output stream.
void close()	Closes the stream

Sample Program: To write a single byte to a file

```
import java.io.FileOutputStream;
class FileOutputStreamExample
{
   public static void main(String args[])     {
        try
        {
          FileOutputStream fout=new FileOutputStream("D:\\test.txt");
          fout.write(65);
          fout.close();
          System.out.println("success...");
       }
       catch(Exception e)
       {
           System.out.println(e);
       }
  }    }
```

Output:

```
Success...
```

Contents of a test.txt :

```
A
```

Q) What is a random access file? Why do we need it? Write a suitable program using it.

Definition : Random Access File allows us to read or write data from or to a file at any location directly. It provides the operations to move around the file, and read from it and write to it, whereever we want.

RandomAccessFile class

This class allows the program to read or write files from any location directly.
It has seek( ) method that sets the file pointer at the specified position. Data read or write starts from that position.

Sample Program :

```
import  java.io.*;
public class RandomAccess
{
        public static void main(String ar[])
        {
                try
                {
                   RandomAccessFile fo = new RandomAccessFile("a.txt", "rw") ;
                   String  s=”Hello World”;
                   char  ch;
                   fo.write(s.getBytes());
                   fo.seek(5);
                   ch = (char)fo.read();
                   System.out.println(ch);
                   fo.close();
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

Output :
W

Q) Explain about RandomAccessFile class and its methods.

RandomAccessFile class

This class allows the program to read or write files from any location directly.

It is defined in java.io package.

It has seek( ) method that sets the file pointer at the specified position. Data read or write starts from that position.

Constructors

1) **RandomAccessFile(File fileObj, String accessType) throws FileNotFoundException**

Here, *fileObj* specifies the name of the file to open as a File object.

2) **RandomAccessFile(String filename, String accessType) throws**

FileNotFoundException

Here, the name of the file is passed in filename as string.

Methods

Modifier and Type	Method	Method
void	close()	Closes this random access file stream

int	readInt()	Reads a signed 32-bit integer from this file.
byte	readByte()	Reads a signed 8-bit value from this file.
double	readDouble()	Reads a double
void	seek(long pos)	It sets the file-pointer offset at pos, measured from the beginning of this file, at which the next read or write occurs.
void	writeDouble(double v)	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
void	writeFloat(float v)	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void	write(int b)	It writes the specified byte to this file.
int	read()	Reads a byte of data from this file.
long	length()	Returns the length of this file.