TED (10)-3069 Reg. No. …….

(REVISION-2010) Signature ……..……

FORTH SEMESTER DIPLOMA EXAMINATION IN ENGINEERING/ TECHNOLIGY- OCTOBER, 2012

OOP THROUGH JAVA

(Common to CT, CM and IF)

[*Time:* 3 hours

(Maximum marks: 100)

Marks

PART –A
(Maximum marks: 10)

I. Answer all questions in a sentence

1. Define class and object

   Objects are basic runtime entity like tes in an object oriented system. They may represent a person, a place, abank account or any item that the program may handle. They may also represent user-defined types such as vectors and lists.

   Classes are user-defined data types and behave like the built in type.

   Eg: Fruit mango;

   will create an object mango belonging to the class Fruit.

2. State the use of super keyword

   - To call the constructor of the super class (parent class)
   - To call the overridden ,method in the parent class

3. List the name of any 4 java API packages

   - Language Support Package (java.lang)
   - Networking Package (java.net)

- Input Output Package (java.io)
- Utilities Package (java.util)

4. Differentiate between local and remote applet

   Local Applet

   An applet developed locally and stored in a local system is known as local applet. It does not require the internet connection.

   Remote Applet

   A remote applet is developed by someone else and stored on a remote computer connected to the internet. If our system is connected to the internet, we can download the remote applet into our system and run it.

5. State Exceptions

   An exception is a condition that is caused by a run-time error. When the Java interpreter finds an error, it creates an exception object and throws it.

   Eg: division by zero errors.

PART – B

II. Answer *any five* questions. Each question carries 6 marks

1. Define constructor and explain its special properties

   A constructor creates an Object of the class that it is in by initializing all the instance variables and creating a place in memory to hold the Object. It is always used with the keyword *new* and then the Class name.

   For instance, new String(); constructs a new String object.

   Sometimes in a few classes you may have to initialize a few of the variables to values apart from their predefined data type specific values. If java initializes variables it would default them to their variable type specific values. For example you may want to initialize an integer variable to 10 or 20 based on a certain condition, when your class is created. In such a case you cannot hard code the value during variable declaration. Such kind of code can be placed inside the constructor so that the initialization would happen when the class is instantiated.

   Syntax:

```
access NameOfClass(parameters)
{
        initialization code
}
```

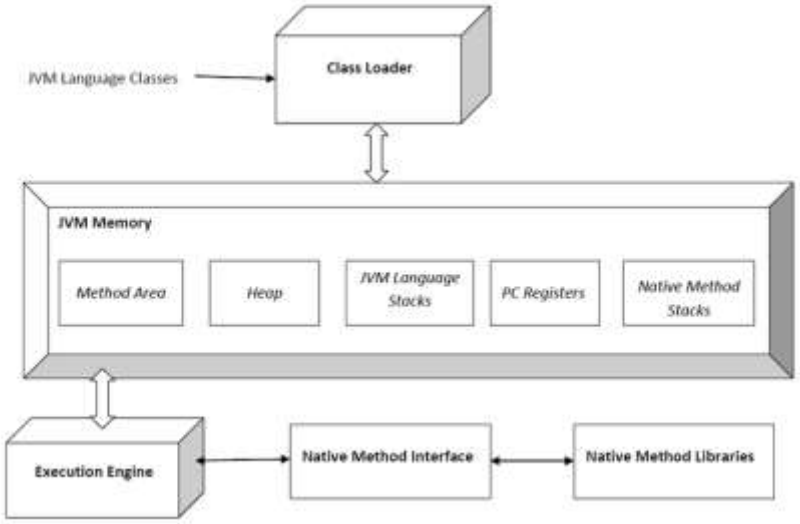
where

- access is one of public, protected, "package" (default), or private;
- *NameOfClass* must be identical to the name of the class in which the constructor is defined; and
- the *initialization code* is ordinary Java declarations and statements.

Properties:

1. You need not code them explicitly. Java will automatically place a default constructor
2. You can pass arguments to the constructor
3. They can return only an object of type of that class
4. They can be made private
5. They would be executed always (Every time a class is instantiated)

2. Write note on java virtual machine



A Java virtual machine (JVM) is an abstract computing machine. There are three notions of the JVM: specification, implementation, and instance. The specification is a book that formally describes what is required of a JVM implementation. Having a single specification ensures all implementations are interoperable. A JVM implementation is a computer program that implements requirements of the JVM specification in a compliant

and preferably per formant manner. An instance of the JVM is a process that executes a computer program compiled into Java byte code. JVM -- a machine within a machine -- mimics a real Java processor, enabling Java byte code to be executed as actions or operating system calls on any processor regardless of the operating system. For example, establishing a socket connection from a workstation to a remote machine involves an operating system call. Since different operating systems handle sockets in different ways, the JVM translates the programming code so that the two machines that may be on different platforms are able to connect.

3. Explain method overriding with example
   - In Method Overriding, sub class has the same method with same name and exactly the same number and type of parameters and same return type as a super class.
   - Method Overriding means method of base class is re-defined in the derived class having same signature.
   - Method Overriding is to "Change" existing behavior of method.
   - It is a run time polymorphism**.**
   - It always requires inheritance in Method Overriding.
   - In Method Overriding, methods must have same signature**.**
   - In Method Overriding, relationship is there between methods of super class and sub class.
   - In Method Overriding, methods have same name and same signature but in the different class.Method Overriding requires at least two classes for overriding.

```
Eg: Class A          // Super Class
{
void display(intnum)
 {
printnum ;
  }
}                    //Class B inherits Class A
Class B              //Sub Class
{
void display(intnum)
 {
printnum ;
  }
```

4. Explain import classes from other packages with example

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the importstatement:

import *pkg1*[.*pkg2*].(*classname*|*);

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

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

Example:

```
import MyPack.*;
class TestBalance
{
  public static void main(String args[])
  {
  /* Because Balance is public, you may use Balance
  class and call its constructor. */
  Balance test = new Balance("J. J. Jaspers", 99.88);
  test.show(); // you may also call show()
  }
}
```

5. Describe thread exception

Thread is the independent path of execution run inside the program. Many Thread run concurrently in the program. Multithread are those group of more than one thread that runs concurrently in a program. Thread in a program is imported from java.lang.thread class. In Multithread, the thread runs concurrently, synchronous or asynchronous

Understand Exception in Threads.

1. A class name RunnableThread implements the Runnable interface gives you the run( ) method executed by the thread. Object of this class is runnable
2. The Thread constructor is used to create an object of RunnableThread class by passing runnable object as parameter.. The Thread object has a Runnable object that implements the run( ) method.
3. The start( ) method is invoked on the Thread object . The start( ) method returns immediately once a thread has been spawned.
4. The thread ends when the run( ) method ends which is to be normal termination or caught exception.
5. runner = new Thread(this,threadName) is used to create a new thread
6. runner. start( ) is used to start the new thread.
7. public void run( ) is overrideable method used to display the information of particular thread
8. Thread.currentThread().sleep(2000) is used to deactivate the thread untill the next thread started execution or used to delay the current thread.

```
try
{
   .....
   .....
}
catch (ThreadDeath e)
{
   ..... // kill thread
}
catch (InterruptedException e)
{
   ..... // cannot handle it in the current state
}
catch (IllegalArgumentException e)
{
   ..... // illegal method argument
}
catch (Exceptio e)
{
   ..... // any other
}
```

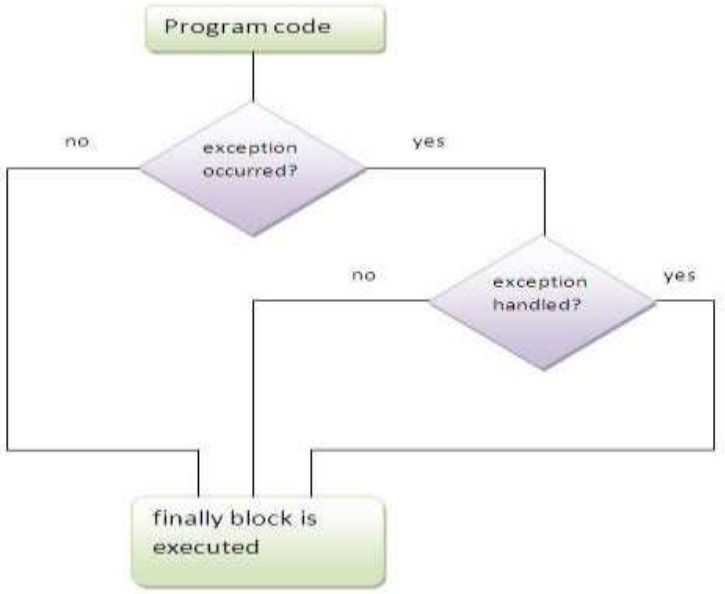
6. Explain the use of finally statement in exception handling

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block.

- The finally block always executes immediately after try-catch block exits.
- The finally block is executed incase even if an unexpected exception occurs.
- The main usage of finally block is to do clean up job. Keeping cleanup code in a finally block is always a good practice, even when no exceptions are occurred.
- The runtime system always executes the code within the finally block regardless of what happens in the try block. So it is the ideal place to keep cleanup code



Eg(1):

```
try
{
   ..................
   ..................
}
finally
{
   ..................
   ..................
}
```

Eg(2):

```
try
{
   ..................
```

```
    ..................
}
catch(……)
{
    ..................
    ..................
}
catch(……)
{
    ..................
    ..................
}
finally
{
    ..................
    ..................
}
```

7. Write the steps involved in developing the testing applets

- Create a project for the applet.
- Use the Applet wizard to create an AWT applet.
- Compile and run the applet.
- Customize the applet's UI.
- Add AWT components, such as Choice, Label, and Button.
- Edit the source code.
- Deploy the applet.
- Modify the HTML file.
- Run the deployed applet from the command line.
- Test the applet.

(5 x 6 = 30)

PART – C

(Answer *one* full question from each unit. Each question carries 15 marks)

UNIT – I

III.

a) Summarize any 8 features of java

Simple :

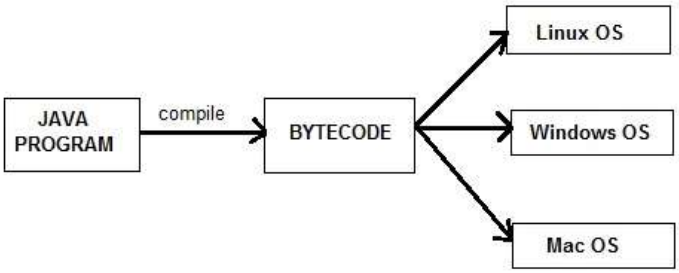
- Java is Easy to write and more readable and eye catching.
- Java has a concise, cohesive set of features that makes it easy to learn and use.
- Most of the concepts are drew from C++ thus making Java learning simpler.

Secure :

- Java program cannot harm other system thus making it secure.
- Java provides a secure means of creating Internet applications.
- Java provides secure way to access web applications.

Portable :

- Java programs can execute in any environment for which there is a Java run-time system.(JVM)
- Java programs can be run on any platform (Linux, Window, Mac)
- Java programs can be transferred over world wide web (e.g applets)



Object-oriented :

- Java programming is object-oriented programming language.
- Like C++ java provides most of the object oriented features.
- Java is pure OOP. Language. (while C++ is semi object oriented)

Robust :

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

Multithreaded :

- Java provides integrated support for multithreaded programming.

Architecture-neutral :

- Java is not tied to a specific machine or operating system architecture.

- Machine Independent i.e Java is independent of hardware .

Interpreted :

- Java supports cross-platform code through the use of Java bytecode.
- Bytecode can be interpreted on any platform by JVM

b) Illustrate method overriding with example

Method overloading

Writing two or more methods in the same class with same name but different parameters list, is known as method overloading

Eg:

```
class Complex
{
        intreal,image;
        void assign()
        {
            real=10;
            image=3;
        }
        void assign(int x, int y)
        {
            real = x;
            image = y;
        }
        void show()
        {
            System.out.println(real+"+"+image+"i");
        }
}
class Test
{
        public static void main(String args[])
        {
            Complex c = new Complex();
            c.assign(6);
            c.show();
        }
```

}

OR

IV.

a) Explain basic concepts of object oriented programming

9

Object- Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation
- Dynamic Binding

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convense the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Dynamic Binding

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method. Thus while calling the overridden method, the compiler gets confused between parent and child class method(since both the methods have same name).

b) State scope of variable

6

Instance variables

Instance variables are those that are defined within a class itself and not in any method or constructor of the class. They are known as instance variables because every instance of the class (object) contains a copy of these variables. The scope of instance variables is determined by the access specifier that is applied to these variables. We have already seen about it earlier. The lifetime of these variables is the same as the lifetime of the object to which it belongs. Object once created do not exist for ever. They are destroyed by the garbage collector of Java when there are no more reference to that object. We shall see about Java's automatic garbage collector later on.

Class variables

In Java, there are some variables that you want to be able to access from anywhere within a Java class. The scope of these variables will need to be at the class level, and there is only one way to create variables at that level − just inside the class but outside of any methods. This is simply a convention; you can define your class-level variables anywhere in the class (so long as it's outside of any methods in the class).

Local variables

A local variable is the one that is declared within a method or a constructor (not in the header). The scope and lifetime are limited to the method itself.

One important distinction between these three types of variables is that access specifies can be applied to instance variables only and not to argument or local variables.

In addition to the local variables defined in a method, we also have variables that are defined in bocks life an if block and an else block. The scope and is the same as that of the block itself.

Eg:

```
class Example
{
  int a,b;          //instance variables(a,b)
  static int count=0;          //class variable(count)
  void assign(int x, int y)    //local varibles(x,y)
  {
                    ……..
                    ……..
  }
}
```

UNIT – II

V.

a) Explain different visibility modes in java

8

a. Public

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe

```
class Example
{
publicint a;  //public field a
public void show()  //public method show
  {Body;}
}
```

b. Private

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

```
class Example
{
privateint a;  //private field a
private void show()  //private method show
  {Body;}
}
```

c. Default/friendly access

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public

```
class Example
{
int a;  //default field a
void show()  //default method show
  {Body;}
}
```

d. Protected

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

```
class Example
{
int a;  //protected field a
void show()  //protected method show
  {   Body;  }
}
```

e. Private protected

Private protected members' visibility lies in between protected and private access. These members are visible in all sub-classes regardless of what package they are in.

b) Describe the use of keyword implements with suitable example

7

The Java programming language has total of 50 reserved keywords which have special meaning for the compiler and cannot be used as variable names. Following is a list of Java keywords in alphabetical order, click on an individual keyword to see its description and usage example.

abstract	**assert**	**boolean**	**break**	**byte**
case	**catch**	**char**	**class**	**const**
continue	**default**	**do**	**double**	**else**
enum	**extends**	**final**	**finally**	**float**
for	**goto**	**if**	**implements**	**import**
instanceof	**int**	**interface**	**long**	**native**
new	**package**	**private**	**protected**	**public**
return	**short**	**static**	**strictfp**	**super**
switch	**synchronized**	**this**	**throw**	**throws**
transient	**try**	**void**	**volatile**	**while**

Some noteworthy points regarding Java keywords:

- const and goto are resevered words but not used.

- true, false and null are literals, not keywords.
- all keywords are in lower-case.

The following table shows the keywords grouped by category:

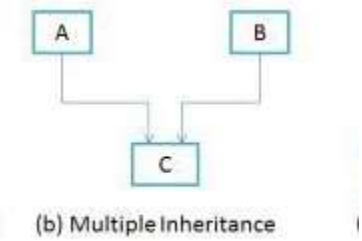
Category	Keywords
Access modifiers	private, protected, public
Class, method, variable modifiers	abstract, class, extends, final, implements, interface, volatile , native,new, static, strictfp, synchronized, transient
Flow control	break, case, continue, default, do, else, for, if , instanceof, return,switch, while
Package control	import, package
Primitive types	boolean, byte, char, double, float, int, long, short
Error handling	assert, catch, finally, throw, throws, try
Enumeration	enum
Others	super, this, void
Unused	const, goto

OR

VI.

a) Explain the implementation of multiple inheritance in java with example 7

"**Multiple Inheritance**" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.



(b) Multiple Inheritance

The Java programming language supports *multiple inheritance of type*, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements. This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface. As with multiple inheritance of implementation, a class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends. In this case, the compiler or the user must decide which one to use.

Eg:

```
class A
{
                        void showA()
                        {
                              System.out.println("class A");
                        }
}
Interface B
{
                        void showB();
}
class C extends A implements B
{
                        void showC()
                        {
                              System.out.println("class C");
                        }
                        public void showB()
                        {
                              System.out.println("interface B");
                        }
}
class Multi
{
                        public static void main(String args[])
                        {
                              C c1=new C();
                              c1.showA();
                              c1.showB();
                              c1.showC();
```

```
                    }
}
```

b) Describe the syntax of multilevel inheritance with appropriate example

8

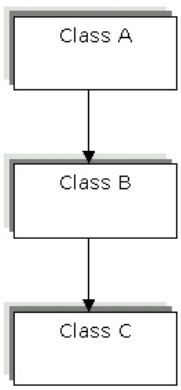
In multilevel, **one-to-one** ladder increases. Multiple classes are involved in inheritance, but one class extends only one. The lowermost subclass can make use of all its super classes' members. Multilevel inheritance is an indirect way of implementing multiple inheritance.

In multilevel inheritance, a subclass inherits the properties of another subclass. For example, Class ***A*** is a superclass for the Class ***B***; and Class ***B*** is a superclass for the subclass, Class ***C***. You can include any number of levels in multilevel inheritance. The following syntax shows how to implement multilevel inheritance:

```
class A
{
//Body of class A
}
class B extends A
{
//Body of class B
}
class C extends B
{
//Body of class C
}
```

In the preceding syntax, class ***A*** is the superclass and class ***C*** is the subclass. The class ***B*** acts as a subclass for the class ***A*** and superclass for the class ***C***.

The following figure shows the structure of multilevel inheritance:



Multi Level Inheritance

UNIT – III

VII.

a) Illustrate the Runnable interface with example

8

A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run**() method** and no other Thread methods.

- A Thread can be created by extending Thread class also. But Java allows only one class to extend, it won't allow multiple inheritances. So it is always better to create a thread by implementing Runnable interface. Java allows you to implement multiple interfaces at a time.
- By implementing Runnable interface, you need to provide implementation for run() method.
- To run this implementation class, create a Thread object, pass Runnable implementation class object to its constructor. Call start() method on thread class to start executing run() method.
- Implementing Runnable interface does not create a Thread object; it only defines an entry point for threads in your object. It allows you to pass the object to the Thread (Runnable implementation) constructor.

Eg:

```
public class RunnableThread implements Runnable
{
 private int countDown = 5;
 public String toString()
 {
  return "#" + Thread.currentThread().getName() +
    ": " + countDown;
 }
 public void run()
```

```
  {
    while(true) {
      System.out.println(this);
      if(--countDown == 0) return;
    }
  }
  public static void main(String[] args)
  {
    for(int i = 1; i <= 5; i++)
      new Thread(new RunnableThread(), "" + i).start();
  }
}
```

b) Illustrate the role of synchronization in Threads with an example

7

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block.

Syntax:

```
synchronized (objectidentifier)
{
   // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

OR

VIII.

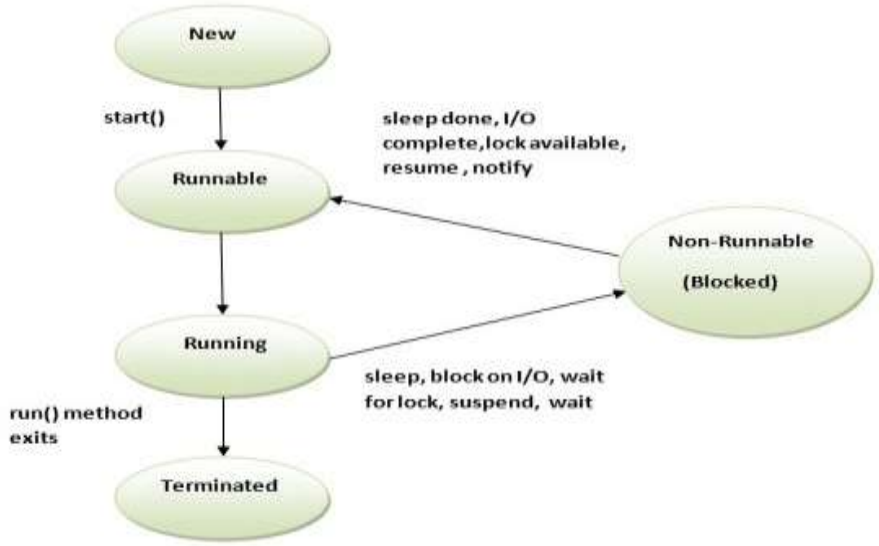
a) Explain the life cycle of a thread with state transition diagram 10

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



New

When we create a new Thread object using *new* operator, thread state is New Thread. At this point, thread is not alive and it's a state internal to Java programming.

Runnable

When we call start() function on Thread object, it's state is changed to Runnable and the control is given to Thread scheduler to finish it's execution. Whether to run this thread instantly or keep it in runnable thread pool before running it depends on the OS implementation of thread scheduler.

Running

When thread is executing, it's state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it's state to Running and CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

Blocked/Waiting

A thread can be waiting for other thread to finish using thread join or it can be waiting for some resources to available, for example producer consumer problem or waiter notifier implementation or IO resources, then it's state is changed to Waiting. Once the thread wait state is over, it's state is changed to Runnable and it's moved back to runnable thread pool.

Dead

Once the thread finished executing, it's state is changed to Dead and it's considered to be not alive.

Above are the different **states of thread** and it's good to know them and how thread changes it's state.

b) Describe briefly about multithreading

5

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc. Java is a *multithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

Advantage of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at same time.

2) You can perform many operations together so it saves time.

3) Threads are independent so it doesn't affect other threads if exceptions occur in a single thread.

UNIT – IV

IX.

a) State and explain the attributes of APPLET tag

8

Here's the complete syntax for the APPLET tag. Required elements are in **bold**. Optional elements are in regular typeface. Elements your specify are in *italics*.

```
<APPLET
    CODEBASE = codebaseURL
    ARCHIVE = archiveList
    CODE = appletFile ...or...  OBJECT = serializedApplet
```

```
    ALT = alternateText
    NAME = appletInstanceName
    WIDTH = pixels  HEIGHT = pixels
    ALIGN = alignment
    VSPACE = pixels  HSPACE = pixels
>
<PARAM NAME = appletAttribute1 VALUE = value>
<PARAM NAME = appletAttribute2 VALUE = value>
. . .
alternateHTML
</APPLET>
```

CODE, CODEBASE, and so on are attributes of the applet tag; they give the browser information about the applet. The only mandatory attributes are CODE, WIDTH, and HEIGHT. Each attribute is described below.

CODEBASE = *codebaseURL*

This OPTIONAL attribute specifies the base URL of the applet--the directory that contains the applet's code. If this attribute is not specified, then the document's URL is used.

ARCHIVE = *archiveList*

This OPTIONAL attribute describes one or more archives containing classes and other resources that will be "preloaded". The classes are loaded using an instance of an AppletClassLoader with the given CODEBASE.

CODE = *appletFile*

This REQUIRED attribute gives the name of the file that contains the applet's compiled Applet subclass. This file is relative to the base URL of the applet. It cannot be absolute. One of CODE or OBJECT must be present.

OBJECT = *serializedApplet*

This attribute gives the name of the file that contains a serialized representation of an Applet. The Applet will be deserialized.

ALT = *alternateText*

This OPTIONAL attribute specifies any text that should be displayed if the browser understands the APPLET tag but can't run Java applets.

NAME = *appletInstanceName*

This OPTIONAL attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.

WIDTH = *pixels* HEIGHT = *pixels*

These REQUIRED attributes give the initial width and height (in pixels) of the applet display area, not counting any windows or dialogs that the applet brings up.

ALIGN = *alignment*

This OPTIONAL attribute specifies the alignment of the applet. The possible values of this attribute are the same as those for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.

VSPACE = *pixels* HSPACE = *pixels*

These OPTIONAL attributes specify the number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE)

<PARAM NAME = *appletAttribute1* VALUE = value>

<PARAM NAME = *appletAttribute2* VALUE = value> . . .

This tag is the only way to specify an applet-specific attribute. Applets access their attributes with the getParameter() method.

b) Illustrate the use of FilOutputStream class for writing bytes to a file

7

A file output stream is an output stream for writing data to a File or to FileDescriptor. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileOutputStream (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

Eg:

```
import java.io.*;
Class WriteBytes
{
    public static void main(String args[]) throw IOException
    {
        byte b[]={'a','k','b','a','r'};
        FileOutputStream f = new FileOutputStream("wb.txt");
        f.write(b);
```

```
            f.close();
        }
    }
```

OR

X.

a) Illustrate the use of FileWriter and FileReader classes for copying characters from one file to another

10

FileReader

This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters.

This class has several constructors to create required objects.

Following syntax creates a new FileReader, given the File to read from.

```
FileReader(File file)
```

Following syntax creates a new FileReader, given the FileDescriptor to read from.

```
FileReader(FileDescriptor fd)
```

Following syntax creates a new FileReader, given the name of the file to read from.

```
FileReader(String fileName)
```

Once you have *FileReader* object in hand then there is a list of helper methods which can be used manipulate the files

FileWriter

This class inherits from the OutputStreamWriter class. The class is used for writing streams of characters.

This class has several constructors to create required objects.

Following syntax creates a FileWriter object given a File object.

```
FileWriter(File file)
```

Following syntax creates a FileWriter object given a File object.

```
FileWriter(File file, boolean append)
```

Following syntax creates a FileWriter object associated with a file descriptor.

```
FileWriter(FileDescriptor fd)
```

Following syntax creates a FileWriter object given a file name.

```
FileWriter(String fileName)
```

Following syntax creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

```
FileWriter(String fileName, boolean append)
```

Once you have *FileWriter* object in hand, then there is a list of helper methods, which can be used manipulate the files.

```
import java.io.*;
class CopyCharacters
{
   public static void main(string args[]) throw IOException
    {
        FileReader fin = new FileReader("input.txt");
        FileWriter fin = new FileWriter("input.txt");
        int ch;

        While((ch=fin.read())!=-1)
                fout.write(ch);
        fin.close();
        fout.close();

    }
}
```

b) Illustrate the use of multiple catch statement with example

A single try block can have multiple catch blocks. This is required when the try block has statements that generates different types of exceptions

Syntax:

```
try {
   // execute code that may throw 1 of the 3 exceptions below.
} catch(ExceptionType-1 e) {
```

```
    statement;
} catch(ExceptionType-2 e) {
    statement;
} catch(ExceptionType-N e) {
    statement;
}
```

Example

```
public class ExceptionExample {
public static void main(String argv[]) {
    int num1 = 10;
    int num2 = 0;
    int result = 0;
    int arr[] = new int[5];
    try {
        arr[0] = 0;
        arr[1] = 1;
        arr[2] = 2;
        arr[3] = 3;
        arr[4] = 4;
        arr[5] = 5;
    result = num1 / num2;
    System.out.println("Result of Division : " + result);
    }catch (ArithmeticException e) {
            System.out.println("Err: Divided by Zero");
    }catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Err: Array Out of Bound");
    }
  }
}
```