Internal Assessment Test 1 – Sept. 2017
Solution Scheme

Sub:	JAVA & J2EE					Sub Code:	10IS753	Branch:	ISE
Date:	22/09/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	A&B		OBE

Answer any FIVE FULL Questions

MARKS

1 List and explain features of JAVA [10M]

Solution:

Explanation -10M

Features of JAVA:

1. Simple
2. Object-Oriented
3. Robust
4. Architecture neutral
5. Interpreted &High Performance
6. Multithreaded
7. Distributed
8. Dynamic

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common cause's f programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multi process synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

Acentral issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI).* This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

2 What is java exception? Explain Exception handling mechanism with try, catch & throw with example program. [10M]

Solution:

Explanation of Exceptions -2M

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

The Three Kinds of Exceptions :
1) Checked exceptions are subject to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, Runtime Exception, and their subclasses.
2) Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.
3) Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by Runtime Except ion and its subclasses.
Valid Java programming language code must honor the Catch or Specify equirement. This means that code that might throw certain exceptions must be enclosed by either of the following:
• A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions.
A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method. Code that fails to honor the Catch or Specify Requirement will not compile. This example describes how to use the three exception handler components — the try, catch, and finally blocks

Explanation of tryblock—3M

try block :
The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following.

```
try
{
    code
} catch and finally blocks . . .
```

```
Example : private Vector vector;
private static final int SIZE = 10;
PrintWriter out = null;
  try
 {
       System.out.println("Entered try statement");
       out = new PrintWriter(new FileWriter("OutFile.txt"));
     for (int i = 0; i < SIZE; i++)
    {
        out.println("Value at: " + i + " = " + vector.elementAt(i));
    }
}
```

Explanation of Catchblock – 2M

The catch Blocks:

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

```
try
{
} catch (ExceptionType name)
{
} catch (ExceptionType name)
{
}
```

Each catch block is an exception handler and handles the type of exception indicated by its argument.

Example:

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

. Explanation of Throw—3M

Throw:

It is possible for your program to throw an exception explicitly using throw statement. The general form of throw is:

throw ThrowableInstance

ThrowableInstance must be an object of type Throwable or a subclass of Throwable. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

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

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

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exceptionhandling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo
```

3 Explain how arrays are defined and used in JAVA with an example. [10M]

Solution:

Explanation of Arrays - 5M

Arrays in Java are actual objects that can be passed around and treated just like other objects. Arrays are a way to store a list of items. Each slot of the array holds an individual element, and you can place elements into or change the contents or those slots as you need to.
Three steps to create an array:
1. Declare a variable to hold the array.
2. Create a new array object and assign it to the array variable.
3. Store things in that array.

E.g. String[] names;
names = new String[10];
names [1] = “n1”;
names[2] = “n2”;
. . .

Example:
```
import java.util.Scanner;
public class JavaProgram
{
   public static void main(String args[])
   {
       int arr[] = new int[50];
       int n, i;

       Scanner scan = new Scanner(System.in);
       System.out.print("How Many Element You Want to Store in array ? ");
       n = scan.nextInt();
       System.out.print("Enter " + n + " Element to Store in Array : ");

       for(i=0; i<n; i++)
       {
           arr[i] = scan.nextInt();
       }
       System.out.print("Elements in Array is :\n");

        for(i=0; i<n; i++)
       {
           System.out.print(arr[i] + "  ");
       }

   }
}
```

Multidimensional Arrays:

Java does not support multidimensional arrays. However, you can declare and create an array of arrays (and those arrays can contain arrays, and so on, for however many dimensions you need), and access them as you would C-style multidimensional arrays:

```
int coords[] [] = new int[12] [12];
 coords[0] [0] = 1; coords[0] [1] = 2;
```

Example:

```
/* Java Program Example - Two Dimensional Array */

import java.util.Scanner;
public class JavaProgram
{
   public static void main(String args[])
   {
       int row, col, i, j;
       int arr[][] = new int[10][10];
       Scanner scan = new Scanner(System.in);

       System.out.print("Enter Number of Row for Array (max 10) : ");
       row = scan.nextInt();
       System.out.print("Enter Number of Column for Array (max 10) : ");
       col = scan.nextInt();

       System.out.print("Enter " +(row*col)+ " Array Elements : ");
       for(i=0; i<row; i++)
       {
           for(j=0; j<col; j++)
           {
               arr[i][j] = scan.nextInt();
           }
       }

       System.out.print("The Array is :\n");
       for(i=0; i<row; i++)
       {
           for(j=0; j<col; j++)
           {
               System.out.print(arr[i][j]+ "  ");
           }
           System.out.println();
       }
   }
}
```

4 (a) Write a Java program to create multiplication table of a number given by user during execution through keyboard [06]

Solution:

Program -6M

```
import java.util.Scanner;
public class Multiplication_Table
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
   System.out.print("Enter number:");
   int n=s.nextInt();
        for(int i=1; i <= 10; i++)
        {
            System.out.println(n+" * "+i+" = "+n*i);
        }
    }
}
```

(b) Develop a Java program to find factorial of a number using recursion. [04]

Solution:

Program -4M

```
import java.util.Scanner;
class FactorialDemo{
 public static void main(String args[])
{
      //Scanner object for capturing the user input
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter the number:");

      //Stored the entered value in variable
      int num = scanner.nextInt();
      //Called the user defined function fact
      int factorial = fact(num);
      System.out.println("Factorial of entered number is: "+factorial);
   }
   static int fact(int n)
   {
       int output;
       if(n==1){
         return 1;
       }
            //Recursion: Function calling itself!!
       output = fact(n-1)* n;
       return output;
   }
}
```

5 Explain life cycle of an applet. Develop a Java applet (with HTML tag) that sets background colour cyan, foreground colour red & outputs a string message “Welcome to CMRIT” [10]

Solution:

Explanation-6M

Life Cycle of an Applet:

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. AWT-based applets will also override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when the applet’s output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

An Applet Skeleton

```
import java.awt.*;
 import javax.swing.*;

/* <applet code="AppletSkel" width=300 height=100>
 </applet> */

public class AppletSkel extends JApplet
{

      // Called first.
     public void init()
    {
       // initialization
    }

   /* Called second, after init(). Also called whenever the applet is restarted.*/

  public void start()
      {
            // start or resume execution
       }


  // Called when the applet is stopped.

   public void stop ()
  {
         // suspends execution
  }
 /* Called when applet is terminated. This is the last method executed. */
  public void destroy()
  {
         // perform shutdown activities
  }

  // Called when an applet's window must be restored.

 public void paint(Graphics g)
  {
      // redisplay contents of window
  }

 }
```

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:
1. stop( )
2. destroy( )

<u>init( ):</u>

The init( ) method is the first method to be called. This is where you should initialize variables.This method is called only once during the run time of your applet.

<u>start( ):</u>

The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet’s HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

<u>paint( ):</u>

The paint( ) method is called each time your applet’s output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used
whenever output to the applet is required.

stop( ):

The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When stop( ) is called, the applet is probably running. You should use stop( ) to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.

destroy( ):

The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

Overriding update( ):

In some situations, your applet may need to override another method defined by the AWT, called update( ). This method is called when your applet has requested that a portion of its window be redrawn. The default version of update( ) simply calls paint( ). However, you can override the update( ) method so that it performs more subtle repainting. In general, overriding update( ) is a specialized technique that is not applicable to all applets.

Program -4M

Program:

```
 import java.awt.*;
 import javax.swing.*;

/* <applet code="AppletSkel" width=300 height=100>
 </applet> */

public class AppletSkel extends JApplet
{

      public void init()
      {
         setBackground(Color.cyan);
         setForeground(Color.red);

      }
      public void paint(Graphics g)
     {
        g.drawString("Welcome to CMRIT");
     }
  }
```

6 Explain final, finally & finalize of Java

<u>Solution:</u>

Explanation-3M

final:

Sometimes you will want to prevent a class from being inherited. To do this, recede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

example of a final class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

Explanation-3M

finally block :

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup. The following finally block for the write List method cleans up and then closes the PrintWriter.

```
Finally
 {
    if (out != null)
   {
        System.out.println("Closing PrintWriter");
       out . close();
   }
  else
      {
          System.out.println("PrintWriter not open");
      }
 }
```

Explanation-4M

finalize:

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize( ) method on the object.

The finalize( ) method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class. It is important to understand that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—finalize( ) will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize( ) for normal program operation.

7 (a) Correct Error in following code segment with explanation: [04]

```
 byte b;
 int i = 257*b;
System.out.println(b);
```

Solution:

```
  byte b;
 int i = 257*b;   //b contains junk value,so it need to be initialized before it
                       is used(b=1). Java by default performs narrowing type
                     casting, byte can be stored in integer datatype.
System.out.println(b); //b = 1.
```

(b) Write the output of following code segment with explanation: [06]

```
 byte b;
 int i = 200;
 b = (byte) i;
System.out.println(b);
```

Solution:

```
  byte b;
 int i = 200;
 b = (byte) i;      //storing integer value in byte variable so we need to perform type casting.
System.out.println(b);  //b=200
```