CMR
INSTITUTE OF
TECHNOLOGY

USN

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

**Internal Assessment Test I – SEP 2016**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Sub:** | **JAVA AND J2EE** | | | | | | | **Code:** | **10CS753** |
| **Date:** | **08/ 09/2016** | **Duration:** | **90 mins** | **Max Marks:** | **50** | **Sem:** | **VII-A&B** | **Branch:** | **CSE** |

**Note: Answer any five full questions.**

**1.** **a) Explain any five object oriented features supported by Java with examples** 7M

**Ans)** All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. Java is an Object-Oriented Language; Java supports the following fundamental concepts:

1. Polymorphism
2. Inheritance
3. Encapsulation
4. Abstraction
5. Classes
6. Objects
7. Instance
8. Method

**1. Encapsulation**

*Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. In Java, the basis of encapsulation is the class. When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables.* The code that operates on that data is referred to as *member methods* or just *methods.*

**2. Inheritance**

*Inheritance* is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. A new subclass inherits all of the attributes of all of its ancestors.

3. **Polymorphism**

The concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*. It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation.

4. **Object -** An object is an instance of a class. Objects have states and behaviours. Example: A dog has states - colour, name, breed as well as behaviours -wagging, barking, eating.

- **Objects in Java:**

All these objects have a state and behaviour. If we consider a dog, then its state is - name, breed, colour, and the behaviour is - barking, wagging, running.
If you compare the software object with a real world object, they have very similar characteristics. Software objects also have a state and behaviour. A software object's state is stored in fields and behaviour is shown via methods. So in software development, methods operate on the internal state of an object and the object-to object communication is done via methods.

5. **Class -** A class can be defined as a template/blue print that describes the behaviours/states that object of its type support. A class is a blue print from which individual objects are created.

A sample of a class is given below:

```
public class Dog
{
   String breed;
   int ageC;
   String colour;
   void barking() { }
   void hungry() { }
   void sleeping() { }
}
```

A class can contain any of the following variable types.

- **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables:** Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables:** Class variables are variables declared with in a class, outside any method, with the static keyword. A class can have any number of methods to access the value of various kinds of methods. In the above example, barking, hungry and sleeping are methods.

**b) How “Write Once Run Anywhere” is implemented in JAVA.** 3M

**Ans)**

- 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.
- 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.

**2. a) How arrays are defined and used in JAVA. Explain with examples.** 4M

**Ans)** *Arrays* are a way to store a list of similar items. Arrays in Java are actual objects that can be passed around and treated just like other objects. Each index of the array holds an individual element, and you can place elements into or change the contents or those index 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’;**

**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.

**E.g.**

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

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

**int twoD[][] = new int[4][5];**

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**. When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

**E.g.**
```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.

**b) Briefly explain the following** 6M

**i) Type Casting.**

**ii) Labelled break and continue statements.**

**Ans)**

**i) Type Casting.**

There are two types,

1. Implicit

   When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

   • The two types are compatible.

   • The destination type is larger than the source type.

   When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

2. Explicit

   Although the automatic type conversions are helpful, they will not fulfill all needs. E.g. the conversion from **int** to **byte** will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion,* since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion.

   It has this general form:

**(*target-type*) *value***

Here, *target-type* specifies the desired type to convert the specified value to.

```
int a;
byte b;
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

**ii) Labelled break and continue statements.**

In Java, the **break** statement has three uses.

1. It terminates a statement sequence in a **switch** statement.
2. It can be used to exit a loop.
3. It can be used as a "civilized" form of goto.

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations. There are, however, a few places where the goto is a valuable and legitimate construct for flow control.

The general form of the labeled **break** statement is shown below:
**break *label*;**

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing block. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block.

```
class Break
{
   public static void main(String args[])
   {
      boolean t = true;
      first: {
      second: {
       third: {
                System.out.println("Before the break.");
                if(t) break second; // break out of second block
```

```
                    System.out.println("This won't execute");
                  }
                  System.out.println("This won't execute");
                }
                System.out.println("This is after second block.");
              }
    }
}
```

- **Continue**

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

```
class Continue
{
   public static void main(String args[])
   {
      for(int i=0; i<10; i++)
      {
         System.out.print(i + " ");
         if (i%2 == 0) continue;
         System.out.println("--");
       }
    }
}
```

**3.** **a) What is the difference between Method Overloading and Method Overriding. Explain with suitable examples.** 10M

**Ans) Method Overriding**

In a class hierarchy, when a method in a subclass has the same name and signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

**E.g.**

```
class A
{
   int i, j;
   A(int a, int b)
  {
      i = a;
      j = b;
  }

// display i and j
   void show()
   {
         System.out.println("i and j: " + i + " " + j);
   }
}

class B extends A
{
   int k;
   B(int a, int b, int c)
   {
       super(a, b);
       k = c;
    }

// display k – this overrides show() in A
   void show()
   {
       System.out.println("k: " + k);
   }
}

class Override
{
   public static void main(String args[])
   {
         B subOb = new B(1, 2, 3);
         subOb.show(); // this calls show() in B
    }
}
```

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.

- **Overloading Methods**

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

```
// Demonstrate method overloading.
class OverloadDemo
{
        void test()
        {
                System.out.println("No parameters");
        }

        // Overload test for one integer parameter.
        void test(int a)
        {
                System.out.println("a: " + a);
        }

        // Overload test for two integer parameters.
        void test(int a, int b)
        {
                System.out.println("a and b: " + a + " " + b);
        }

        // overload test for a double parameter
        double test(double a)
        {
                System.out.println("double a: " + a);
                return a*a;
        }
}
class Overload
{
        public static void main(String args[])
        {
                OverloadDemo ob = new OverloadDemo();
                double result;
                // call all versions of test()
                ob.test();
                ob.test(10);
                ob.test(10, 20);
                result = ob.test(123.25);
                System.out.println("Result of ob.test(123.25): " + result);
        }
}
```

**test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double**

parameter. The fact that the fourth version of **test( )** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution. When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution. Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.

**Rules for Method Overriding**

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.

**4.** **a) Explain the constructors in Java. How is it different from other member functions.** 6M

**Ans)**

It can be tedious to initialize all of the variables in a class each time an instance is created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor. A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box
{
        double width;
        double height;
        double depth;
```

```
	// This is the constructor for Box.
	Box()
	{
		System.out.println("Constructing Box");
		width = 10;
		height = 10;
		depth = 10;
	}

	// compute and return volume
	double volume()
	{
		return width * height * depth;
	}
}
class BoxDemo
{
	public static void main(String args[])
	{
		// declare, allocate, and initialize Box objects
		Box mybox1 = new Box();
		Box mybox2 = new Box();
		double vol;

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

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

Both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

- **Parameterized Constructors**

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box
{
        double width;
        double height;
        double depth;

        // This is the constructor for Box.
        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }

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

        class BoxDemo
{
        public static void main(String args[])
        {
                // declare, allocate, and initialize Box objects
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box(3, 6, 9);
                double vol;

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

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

**}**

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,
**Box mybox1 = new Box(10, 20, 15);**
the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

**b) With an example, explain call to this() and call to super().** 4M

**Ans)**
In Java, this is a **reference variable** that refers to the current object. this() can be used to invoke current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

```
class Student
{
        int id;
        String name;
        Student()
        {
                System.out.println("default constructor is invoked");
        }

        Student(int id,String name)
        {
                this ();//it is used to invoked current class constructor.
                this.id = id;
                this.name = name;
        }
        Student(int id, String name, String city)
        {
                this(id, name);//now no need to initialize id and name
                this.city=city;
        }
        void display()
        {
                System.out.println(id+" "+name);
        }

        public static void main(String args[])
        {
                Student e1 = new Student(001,"Nirmal");
                Student e2 = new Student(002,"Pandey");
                e1.display();
                e2.display();
        }
}
```

- **Super()**

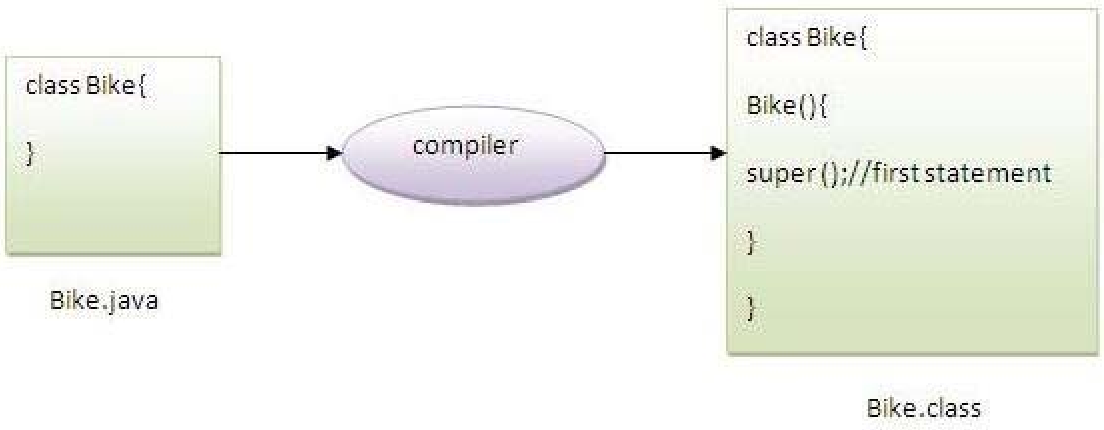
The **super** keyword in java is a reference variable that is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

- **Usage of java super Keyword**

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

- **super is used to invoke parent class constructor.**

```
class Vehicle
{
    Vehicle()
{
            System.out.println("Vehicle is created");
    }
}
class Bike extends Vehicle
{
    Bike()
    {
            super();//will invoke parent class constructor
            System.out.println("Bike is created");
    }
    public static void main(String args[])
    {
            Bike b=new Bike();
    }
}
```



A default constructor is provided by compiler automatically but it also adds super() for the first statement. If we are creating our own constructor and if we don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

**5.** **a) What is Java Exception? Explain the exception handling mechanism with an example.** 10M

**Ans)** Exception is an abnormal condition. In Java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. Exception normally disrupts the normal flow of the application that is why we use exception handling. The **exception handling in Java** is one of the powerful ***mechanism to handle the runtime errors so that normal flow of the application can be maintained***. There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

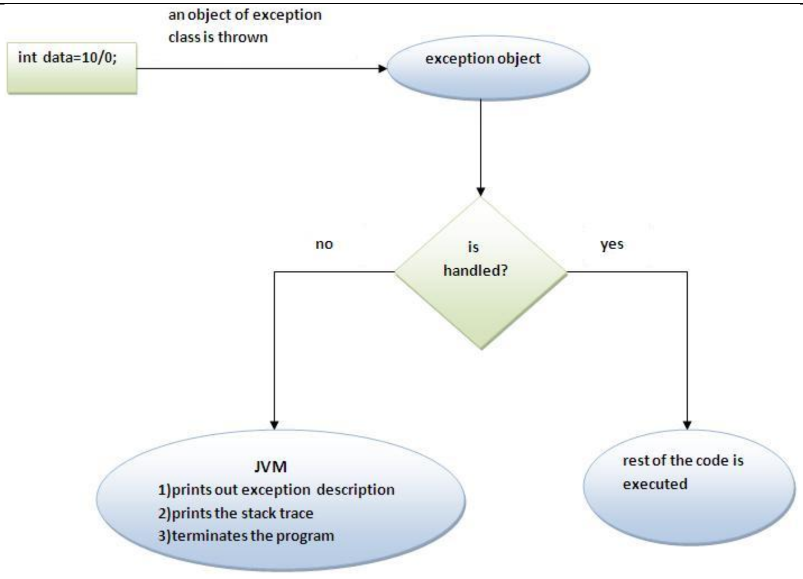
- **Java try block**

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block. Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

```
public class Test_try_catch
{
     public static void main(String args[])
     {        try
               {
                         int data=50/0;
               }catch(ArithmeticException e)
                {
                         System.out.println(e);
                }
                finally
                {
                     System.out.println("rest of the code...");
                 }
     }
}
```

**Output:**

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


an object of exception class is thrown
int data=10/0;
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

6. **a) Explain the applet architecture with the program example.** 10M

**Ans)** An applet is a window-based program; its architecture is different from the console-based Programs

- **First, applets are event driven.**

An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point. For the most part applet should not enter a “mode” of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations, in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window),

- **Second, the user initiates interaction with an applet—not the other way around.**

In a non-windowed program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( )**. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet’s window, a mouse-clicked event is generated. If the user presses a key while the applet’s window has input focus, a keypress event is generated.

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.

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet
{
        // Called first.
        // set the foreground and background colors.
        public void init()
        {
                setBackground(Color.cyan);
                setForeground(Color.red);
                msg = "Inside init( ) --";
        }

        /* Called second, after init(). Also called whenever
        the applet is restarted. */
        // Initialize the string to be displayed.
        public void start()
        {
                msg += " Inside start( ) --";
        }

        // 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.
        // Display msg in applet window.
        public void paint(Graphics g)
        {
                msg += " Inside paint( ).";
                g.drawString(msg, 10, 30);
        }
}
```

- **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( )**

1. **init( ) -** 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.
2. **start( ) -** 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( )**.
3. **paint( ) -** 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.
4. **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.
5. **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( )**.

7. **a) Explain in detail, the syntax of Applet tag.** 6M

**Ans)** The APPLET tag be used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional.

**< APPLET**
**[CODEBASE = *codebaseURL*]**
**CODE = *appletFile***
**[ALT = *alternateText*]**
**[NAME = *appletInstanceName*]**
**WIDTH = *pixels* HEIGHT = *pixels***
**[ALIGN = *alignment*]**
**[VSPACE = *pixels*] [HSPACE = *pixels*]**
**>**
**[< PARAM NAME = *AttributeName* VALUE = *AttributeValue*>]**
**[< PARAM NAME = *AttributeName2* VALUE = *AttributeValue*>]**
**. . .**
**[*HTML Displayed in the absence of Java*]**
**</APPLET>**

- **CODEBASE-**it is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.
- **CODE-** it is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
- **ALT-**The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.
- **NAME-** it is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.
- **WIDTH and HEIGHT**-are required attributes that give the size (in pixels) of the applet display area.
- **ALIGN-**it is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.
- **VSPACE and HSPACE-**These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

- **PARAM NAME and VALUE-**The PARAM tag allows you to specify applet-specific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method. Other valid APPLET attributes include ARCHIVE, which lets you specify one or more archive files, and OBJECT, which specifies a saved version of the applet. In general, an APPLET tag should include only a CODE or an OBJECT attribute, but not both.

**b) Give the different forms of repaint() method.** 4M

**Ans)** An applet writes to its window only when its **update( )** or **paint( )** method is called by the AWT. The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**. For example, if part of your applet needs to output a string, it can store this string in a **String** variable and then call **repaint( )**. Inside **paint( )**, you will output the string using **drawString( )**. The **repaint( )** method has four forms. The simplest version of **repaint( )** is :

**void repaint( )**

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

**void repaint(int *left*, int *top*, int *width*, int *height*)**

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top,* and the width and height of the region are passed in *width* and *height.* These dimensions are specified in pixels. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region. Calling **repaint( )** is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:

**void repaint(long *maxDelay*)**

**void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)**

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update()** is called. Beware, though. If the time elapses before **update()** can be called, it isn't called. There's no return value or exception thrown, so you must be careful.