MAHARAJA INSTITUTE OF TECHNOLOGY MYSORE

BELAWADI, SRIRANGAPATNA Taluk, MANDYA-571477

Department of Computer Science and Business System

(Academic Year – 2023-24)

Third Semester B.E. Degree Examination

Object Oriented Programming with JAVA (BCS306A)

TIME: 03 Hours **Max. Marks: 100**

Note:

01.Answer any FIVE full questions, choosing at least ONE question from each MODULE.

MODULE-1

1. (a). Explain different lexical issues in JAVA. **7Marks**

Answer:

Lexical Issues:

Whitespace:

Java is a free-form language. This means that you do not need to follow any special indentation rules. For instance, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator.

Identifiers:

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so VALUE is a different identifier than Value.

Literals:

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	'X'	"This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments:

There are **three types of comments defined by Java**. You have already seen two: **single-line and multiline**. The third type is called a **documentation comment**. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a **/** and ends with a */**.

Separators:

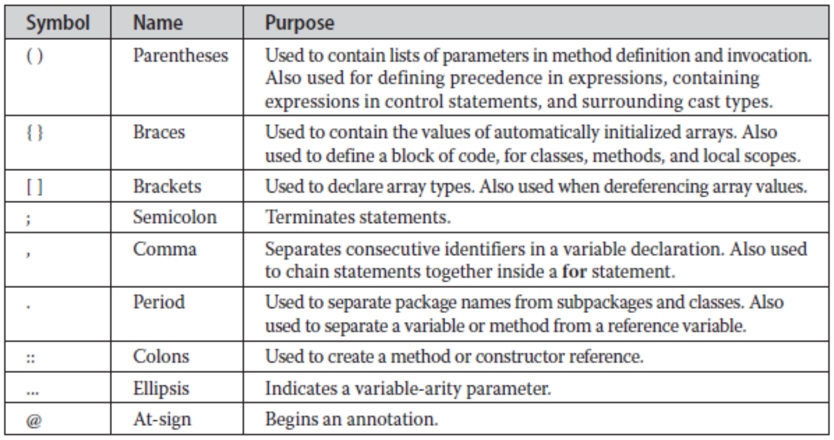
In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is often used to terminate statements.

The separators are shown in the following table:

The Java Keywords:

Dr. Honnaraju B, Dept. of CS&BS, MITM

There are 67 keywords currently defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. In general, keywords cannot be used as identifiers, meaning that they cannot be used as names for a variable, class, or method.



Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference.
...	Ellipsis	Indicates a variable-arity parameter.
@	At-sign	Begins an annotation.

Solution:

Explain six lexical issues of Java **7 Marks**

Scheme [7 = 7 Marks]

(b). Define Array. Write a Java program to implement the addition of two matrixes.

Answer: **7 Marks**

An array is a group of **like-typed variables that are referred to by a common name**. Arrays of any type can be created and may have **one or more dimensions**. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

```
//Java program to perform addition of 2 Matrix
import java.util.Scanner;
public class Program1
{
       public static void main(String args[])
       {

              int n;
              int i, j;
              Scanner in = null;
              n = Integer.parseInt(args[0]);
             try
              {
               in = new Scanner(System.in);
               System.out.println("Size of the matrix is "+n);
               // Declare the matrix
               int a[][] = new int[n][n];
               int b[][] = new int[n][n];
               int c[][] = new int[n][n];
               // Read the matrix A values
               System.out.println("Enter the elements of the matrix A");
               for (i = 0; i < n; i++)
                  for (j = 0; j < n; j++)
```

Dr. Honnaraju B, Dept of CS&BS, MITM

```
                a[i][j] = in.nextInt();
            // Read the matrix B values
             System.out.println("Enter the elements of the matrix B");
             for (i = 0; i < n; i++)
                 for (j = 0; j < n; j++)
                     b[i][j] = in.nextInt();
           //Addition of 2 matrices
          for(i=0;i<n;i++)
          {
                  for(j=0;j<n;j++)
                  {
                        c[i][j] =a[i][j] + b[i][j];
                  } //end of j loop
          }
             // Display the elements of the matrix
             System.out.println("Elements of resultant matrix are");
             for (i = 0; i < n; i++)
             {
                 for (j = 0; j < n; j++)
                     System.out.print(c[i][j] + "  ");
                 System.out.println();
             }
         }
         catch (Exception e) {
         }
         finally
         {
             in.close();
         }
     }
}
```

Solution:

Define Array	**1 Marks**
Write a program to add 2 matrix	**6 Marks**

Scheme [1 + 6 = 7 Marks]

(c). Explain the following operations with examples. (i) << (ii) >> (iii)>>> **6 Marks**

Answer:

The Left Shift (<<):

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form:

value << num

Here, num specifies the number of positions to left-shift the value in value. That is, the **<< moves all of the bits in the specified value to the left by the number of bit positions specified by num**. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an int operand, bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63.

The Right Shift (>>):

Dr. Honnaraju B, Dept. of CS&BS, MITM

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

value >> num

Here, num specifies the number of positions to right-shift the value in value. That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num.

The following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8:

```
int a = 32;
a = a >> 2; // a now contains 8
```

The Unsigned Right Shift (>>>):

As you have just seen, the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift. To accomplish this, you will use Java's unsigned, shift-right operator,>>>, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the >>>. Here, a is set to –1, which sets all32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
int a = -1;
a = a >>> 24;
```

Solution:

Explain >>, <<, and >>> operators in detail 3 * 2 = 6 Marks

Scheme [6 = 6 Marks]

OR

2. (a). Explain object-oriented principles. 7 Marks

Answer:

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are **encapsulation, inheritance, and polymorphism**.

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 p**rotective 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. A class defines the **structure and behavior (data and code) that will be shared by a set of objects**. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class.

Inheritance:

Dr. Honnaraju B, Dept. of CS&BS, MITM

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**.

For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. 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.

Polymorphism:

Polymorphism is a feature that allows one interface to be used for a **general class of actions**. The specific action is **determined by the exact nature of the situation**. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non–object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of **polymorphism**, in Java you can specify a general set of **stack routines that all share the same names**.

Solution:

Explain the object oriented principals 7 Marks

Scheme [7 = 7 Marks]

(b). Write a Java program to sort the elements using a for loop. 7 Marks

Answer:

```
// Java Program to Sort Elements of an Array in Ascending Order
class Sorting
 {
    static int length;
    public static void printArray(int[] array)
   {
        for (int i = 0; i < length; i++)
        {
          System.out.print(array[i] + " ");
       }
       System.out.println();
   }

   public static void sortArray(int[] array)
   {
       int temporary = 0;
       for (int i = 0; i < length; i++)
        {
          for (int j = i + 1; j < length; j++)
         {
              if (array[i] > array[j])
             {
                  temporary = array[i];
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
                    array[i] = array[j];
                    array[j] = temporary;
                }
            }
        }
         System.out.println("Elements of array sorted in ascending order: ");
        printArray(array);
    }
     public static void main(String[] args)
    {
        int[] array = new int[] { -5, -9, 8, 12, 1, 3 };
        length = array.length;
        System.out.print("Elements of original array: ");
        printArray(array);
        sortArray(array);
    }
}
```

Solution:

Program to sort the elements using a for loop 7 Marks

Scheme [7 = 7 Marks]

(c). Explain different types of if statements in JAVA. 6 Marks

Answer:

if

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if (condition)
        statement1;
else
        statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the **condition is true**, **then statement1 is executed**. Otherwise, **statement2 (if it exists) is executed**. In no case will both statements be executed. For example, consider the following:

```
int a, b;
//...
if(a < b)
        a = 0;
else
        b = 0;
```

Nested ifs

A nested if is an **if statement that is the target of another if or else**. Nested ifs are very **common in programming**. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that **is not already associated with an else**. Here is an example:

```
if(i == 10)
{
```

Dr. Hombaraju B, Dept. of CS&BS, MITM

```
if(j < 20)
        a = b;
if(k > 100)
        c = d; // this if is
else
        a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

The if-else-if Ladder
A common programming construct that is based upon a sequence of nested **ifs** is the *if-else-if* ladder. It looks like this:

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement; . . .
else statement;
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Solution:

Explain different types of if statement 6 Marks

Scheme [6 = 6 Marks]

MODULE-2

3. (a). What are constructors? Explain two types of constructors with an example program. 7 Marks

Answer:

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 when 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**.

There are 2 types of constructor:

I. Simple Constructor
II. Parameterized Constructors

Simple Constructor:

Let's begin by defining a **simple constructor** that sets the dimensions of each box to the same values. This version is shown here:

Dr. Honnaraju B, Dept. of CS&BS, MITM

/* Here, Box uses a constructor to initialize the dimensions of a box. */

```
	class Box
	{
		double width;
		double height;
		double depth;
		Box()
		 {
			System.out.println("Constructing Box");
			width = 10;
			height = 10;
			depth = 10;
		}
		double volume()
		{
			return width * height * depth;
		}
	}
	class Constructor
	{
		public static void main(String[] args)
		{
			Box mybox1 = new Box();
			Box mybox2 = new Box();
			double vol;
			vol = mybox1.volume();
			System.out.println("Volume is " + vol);
			vol = mybox2.volume();
			System.out.println("Volume is " + vol);
		}
	  }
```

As you can see, 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.
When you allocate an object, you use the following general form:

class-var = new classname( );

What is actually happening is that the constructor for the class is being called. Thus, in the line

Box mybox1 = new Box();

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.
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.

Dr. Honnaraju B, Dept. of CS&BS, MITM

/* Here, Box uses a parameterized constructor to initialize the dimensions of a box. */

```
class Box
 {
        double width;
        double height;
        double depth;
        Box(double w, double h, double d)
         {
                width = w;
                height = h;
                depth = d;
        }
        double volume()
         {
                return width * height * depth;
        }
}
class Constructor
{
        public static void main(String[] args)
         {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box(3, 6, 9);
                double vol;
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

Solution:

Define constructor	**1 Marks**
Explain two types of constructor	**6 Marks**

Scheme [1 + 6 = 7 Marks]

(b). Define recursion. Write a recursive program to find nth Fibonacci number.

Answer: **7 Marks**

Java supports recursion. **Recursion is the process of defining something in terms of itself.** As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

```
// Recursive implementation of Fibonacci series program in java
class Fibonacci
 {
   static int fib(int n)
   {
      if (n <= 1)
         return n;
      // Recursive call
      return fib(n - 1) + fib(n - 2);
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
        }
        public static void main(String args[])
        {
            int N = 10;
            for (int i = 0; i < N; i++)
              {
                System.out.print(fib(i) + " ");
              }
        }
    }
```

Solution:

Define recursion 1 Marks

Write a program to find Fibonacci Series 6 Marks

Scheme [1 + 6 = 7 Marks]

(c). Explain the various access specifiers in Java. **6 Marks**

Answer:

Java's access modifiers are **public, private, and protected**. Java also defines a default access level. **protected applies** only when **inheritance is involved**.

When a member of a class is modified by public, then that member can be **accessed by any other code**. When a member of a class is specified **as private**, then that member can only be accessed by **other members** of **its class**. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;
private double j;

private int myMethod(int a, char b)
        {
        //...
```

Example:

```
/* This program demonstrates the difference between public and private. */
class Test
 {
        int a;                  // default access
        public int b;         // public access
        private int c;        // private access

        // methods to access c
        void setc(int i)
        {
                // set c's value
                c = i;
        }
        int getc()
        {
                // get c's value
                return c;
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
        }
}
class AccessTest
{
        public static void main(String[] args)
        {
                Test ob = new Test();
                // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;
                // This is not OK and will cause an error
                // ob.c = 100;                              // Error!
                // You must access c through its methods
                ob.setc(100);                            // OK
                System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
        }
 }
```

Inside the Test class, a uses **default access**, which for this example is the same as specifying **public**. **b is explicitly specified as public**. Member **c is given private access**. This means that it **cannot be accessed by code outside** of its class. So, inside the **AccessTest** class, c cannot be used directly.

It must be accessed through its public methods:

setc( ) and getc( ).

If you were to remove the comment symbol from the beginning of the following line,

// ob.c = 100; // Error!

then you would not be able to compile this program because of the access violation.

Solution:

Explain different access specifiers 3 Marks
Write example programs 3 Marks

Scheme [3 + 3 = 6 Marks]

OR

4. (a). Explain call by value and call by reference with an example program. 7 Marks

Answer:

Call by Value:

This approach **copies the value of an argument into the formal parameter of the subroutine**. Therefore, **changes made to the parameter of the subroutine have no effect on the argument**.

```
// Primitive types are passed by value.
class Test
 {
        void meth(int i, int j)
        {
                i *= 2;
                j /= 2;
        }
}
class CallByValue
{
```

Dr. Honnaraju B. Dept. of CS&BS, MITM

```
        public static void main(String[] args)
         {
                Test ob = new Test();
                int a = 15, b = 20;
                System.out.println("a and b before call: " + a + " " + b);
                ob.meth(a, b);
                System.out.println("a and b after call: " + a + " " + b);
        }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

Call by Reference:

In this approach, a **reference to an argument (not the value of the argument) is passed to the parameter**. Inside the subroutine, this reference is used to **access the actual argument specified in the call**. This means that **changes made to the parameter will affect the argument used to call the subroutine**.

When you pass this **reference to a method**, the parameter that receives it will refer to the **same object as that referred to by the argument**. This effectively means that objects act as if they are passed to methods by use of **call-by-reference**. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program:

```
        // Objects are passed through their references.
        class Test
        {
                int a, b;
                Test(int i, int j)
                {
                        a = i;
                        b = j;
                }
                // pass an object
                void meth(Test o)
                {
                        o.a *= 2;
                        o.b /= 2;
                }
        }
        class PassObjRef
        {
                public static void main(String[] args)
                 {
                Test ob = new Test(15, 20);
                System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
                ob.meth(ob);
                System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
                }
        }
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
This program generates the following output:
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

Solution:

Explain Call-by value and Call by reference **4 Marks**

Write the example programs **3 Marks**

Scheme [4 + 3 = 7 Marks]

(b). Write a program to perform Stack operations using proper class and Methods.

Answer: **7 Marks**

```
// This class defines an integer stack that can hold 10 values.
class Stack
{
private int[] stck = new int[10];
private int tos;

Stack()
{
        tos = -1;
}
void push(int item)
{
        if(tos==9)
                System.out.println("Stack is full.");
        else
                stck[++tos] = item;
}
int pop()
{
        if(tos < 0)
        {
                System.out.println("Stack underflow.");
                return 0;
        }
        else
                return stck[tos--];
        }
}
class TestStack
{
        public static void main(String[] args)
         {
        Stack mystack = new Stack();
        for(int i=0; i<10; i++)
                mystack.push(i);
        System.out.println("Stack in mystack:");
        for(int i=0; i<10; i++)
                System.out.println(mystack.pop());
        }
}
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

Solution:

Write a program to implement stack operation **7 Marks**

Scheme [7 = 7 Marks]

(c) Explain the use of this in JAVA with an example. **6 Marks**

Answer:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class' type is permitted. To better understand what this refers to, consider the following version of Box( ):

```
// A redundant use of this.
Box(double w, double h, double d)
{
        this.width = w;
        this.height = h;
        this.depth = d;
}
```

this will always refer to the invoking object.

However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. This is why **width, height, and depth were not used as the names** of the parameters to the Box( ) constructor inside the **Box class**. If they had been, then width, for example, would have referred to the formal parameter, **hiding the instance variable** width. While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables.

For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth)
{
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

Solution:

Explain use of this in Java with example. **6 Marks**

Scheme [6 = 6 Marks]

MODULE-3

5. (a). Write a Java program to implement multilevel inheritance with 3 levels of hierarchy. **7 Marks**

Answer:

```
// Extend BoxWeight to include shipping costs.
// Start with Box.
class Box
 {
```

Dr. Hemaraju B, Dept. of CS&BS, MITM

```
    private double width;
    private double height;
    private double depth;
    // construct clone of an object
    Box(Box ob)
    { // pass object to constructor
            width = ob.width;
            height = ob.height;
            depth = ob.depth;
    }
    // constructor used when all dimensions specified
    Box(double w, double h, double d)
     {
            width = w;
            height = h;
            depth = d;
    }
    // constructor used when no dimensions specified
    Box()
    {
            width = -1; // use -1 to indicate
            height = -1; // an uninitialized
            depth = -1; // box
    }
    // constructor used when cube is created
    Box(double len)
    {
    width = height = depth = len;
    }
    // compute and return volume
    double volume()
     {
            return width * height * depth;
    }
}
// Add weight.
class BoxWeight extends Box
{
    double weight; // weight of box
    // construct clone of an object
    BoxWeight(BoxWeight ob)
     { // pass object to constructor
    super(ob);
    weight = ob.weight;
    }
    // constructor when all parameters are specified
    BoxWeight(double w, double h, double d, double m)
    {
    super(w, h, d); // call superclass constructor
    weight = m;
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
        }
        // default constructor
        BoxWeight()
        {
        super();
        weight = -1;
        }
    // constructor used when cube is created
    BoxWeight(double len, double m)
    {
    super(len);
    weight = m;
    }
    }
    // Add shipping costs.
    class Shipment extends BoxWeight
    {
        double cost;
        // construct clone of an object
        Shipment(Shipment ob)
         { // pass object to constructor
            super(ob);
            cost = ob.cost;
        }
        // constructor when all parameters are specified
        Shipment(double w, double h, double d, double m, double c)
        {
            super(w, h, d, m); // call superclass constructor
            cost = c;
        }
        // default constructor
        Shipment()
        {
            super();
            cost = -1;
        }
        // constructor used when cube is created
        Shipment(double len, double m, double c)
         {
            super(len, m);
            cost = c;
        }
    }
class DemoShipment
{
    public static void main(String[] args)
     {
    Shipment shipment1 =
    new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 =
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
    new Shipment(2, 3, 4, 0.76, 1.28);
    double vol;
    vol = shipment1.volume();
    System.out.println("Volume of shipment1 is " + vol);
    System.out.println("Weight of shipment1 is " + shipment1.weight);
    System.out.println("Shipping cost: $" + shipment1.cost);
    System.out.println();
    vol = shipment2.volume();
    System.out.println("Volume of shipment2 is " + vol);
    System.out.println("Weight of shipment2 is " + shipment2.weight);
    System.out.println("Shipping cost: $" + shipment2.cost);
    }
}
```

Solution:

Program to implement multilevel inheritance 7 Mark

Scheme [7 = 7 Marks]

(b). Explain how an interface is used to achieve multiple Inheritances in Java.

Answer: 6 Marks

Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when methods with the same signature exist in both the superclasses and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- That is, using **interface**, you can specify what a class must do, but not how it does it.
- **Interfaces are syntactically similar to classes**, but they **lack instance variables, and, as a general rule, their methods are declared without any body**.
- In practice, this means that you can define interfaces that don't **make assumptions about how they are implemented**.

Once it is defined, **any number of classes can implement** an **interface**. Also, one class can implement any number of interfaces.

- To implement an interface, a class must provide the **complete set of methods required by the interface**.

Syntax for implementing multiple inheritance.

```
interface Interface1
 {
   void method1();
}

interface Interface2
 {
   void method2();
}
```

Dr. Hommaraju B, Dept. of CS&BS, MITM

```
public class MyClass implements Interface1, Interface2
 {
    public void method1()
     {
            // implementation of method1
     }

    public void method2()
    {
           // implementation of method2
    }
}
```

Create an object of the class and call the interface methods

```
MyClass obj = new MyClass();
obj.method1();
obj.method2();
```

Example:

```
// Declare the interfaces
interface Walkable
 {
    void walk();
}
interface Swimmable
{
    void swim();
}
// Implement the interfaces in a class
class Duck implements Walkable, Swimmable
 {
    public void walk()
    {
        System.out.println("Duck is walking.");
    }

    public void swim()
    {
        System.out.println("Duck is swimming.");
    }
}

// Use the class to call the methods from the interfaces
class Main
 {
    public static void main(String[] args)
    {
        Duck duck = new Duck();
        duck.walk();
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
            duck.swim();
        }
    }
```

Solution:

Explain how multiple inheritance implemented using interface **4 Marks**
Write an example program **3 Marks**
Scheme [4 + 3 = 7 Marks]

c. Explain the method overriding with a suitable example. **6 Marks**

Answer:

In a class hierarchy, when a method in a **subclass has the same name and type 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 through its 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. Consider the following:

```
// Method overriding.
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)
         {
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
                        B subOb = new B(1, 2, 3);
                        subOb.show(); // this calls show() in B
                    }
            }
```
The output produced by this program is shown here:

```
k: 3
```

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**.

If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A
{
    int k;
    B(int a, int b, int c)
    {
            super(a, b);
            k = c;
    }
    void show()
    {
            super.show(); // this calls A's show()
            System.out.println("k: " + k);
     }
}
```

Here, **super.show( )** calls the superclass version of **show( )**.

Solution:

Explain method overriding with example program 6 Marks

Scheme [3 + 3 = 6 Marks]

OR

6. (a). What is single-level inheritance? Write a Java program to implement single-level inheritance. 7 Marks

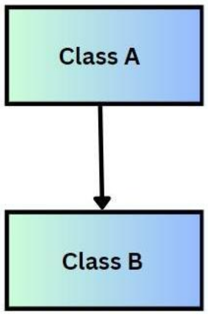
Answer:

Java, Inheritance is an important pillar of **OOP (Object-Oriented Programming).** It is the mechanism in Java by which one class is allowed to inherit the features (**fields and methods**) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that **inherits from another class can reuse the methods and fields of that class**. In addition, you can add new fields and methods to your current class as well.

Single-level inheritance in Java is a fundamental concept where a subclass inherits from only one superclass. This means that the subclass can acquire the properties and methods of the superclass, enabling code reuse and the extension of existing functionality.

A graphical illustration of single-level inheritance is given below:

Dr. Honnaraju B, Dept. of CS&BS, MITM.



Syntax of Single inheritance:

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

Example Program:

```
class Animal
{
        void eat()
        {
                System.out.println("eating...");
        }
 }
class Dog extends Animal
{
        void bark()
        {
                System.out.println("barking...");
        }
 }
 class TestInheritance
 {
        public static void main(String args[])
        {
                Dog d=new Dog();
                d.bark();
                d.eat();
        }
}
```

Solution:

Explain single inheritance with syntax.	**3 Marks**
Write an example program to illustrate inheritance.	**7 Marks**

Scheme [3 + 4 = 7 Marks]

(b). What is the importance of the super keyword in inheritance? Illustrate with a suitable example. 7 Marks

Answer:

Whenever a subclass needs to refer to its immediate superclass (When using inheritance), it can do so by use of the keyword **super**.

Dr. Hommaraju B, Dept. of CS&BS, MITM

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors
A subclass can call a constructor defined by its superclass by use of the following form of **super**:

super(*arg-list*);

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

To see how **super( )** is used, consider this improved version of the **BoxWeight** class:

```
// BoxWeight now uses super to initialize its Box attributes.
    class BoxWeight extends Box
    {
      double weight; // weight of box

       // initialize width, height, and depth using super()
      BoxWeight(double w, double h, double d, double m)
       {
                super(w, h, d); // call superclass constructor
                weight = m;
       }
    }
```

A Second Use for super
The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
   // Using super to overcome name hiding.
   class A
   {
    int i;
   }

// Create a subclass by extending class A.

    class B extends A
    {
       int i; // this i hides the i in A
      B(int a, int b)
      {
                super.i = a; // i in A
                i = b; // i in B
      }
      void show()
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
 {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
}
 }

class UseSuper
{
        public static void main(String[] args)
         {
         B subOb = new B(1, 2);
         subOb.show();
         }
  }
```

Solution:

Explain the use of super in inheritance. **4 Marks**
Write an example program. **3 Marks**
Scheme [4 + 3 = 7 Marks]

(c). What is abstract class and abstract method? Explain with an example. 6 Marks

Answer:

To declare a class abstract, you simply use the abstract **keyword in front of the class keyword** at the beginning of the class declaration. There can **be no objects of an abstract** class. That is, an **abstract class cannot be directly instantiated** with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare **abstract constructors, or abstract static methods**. Any subclass of an **abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself**.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
    // A Simple demonstration of abstract.
    abstract class A
    {
        abstract void callme();
        // concrete methods are still allowed in abstract classes
        void callmetoo()
        {
        System.out.println("This is a concrete method.");
        }
    }
    class B extends A
    {
        void callme()
        {
        System.out.println("B's implementation of callme.");
        }
    }
  class AbstractDemo
  {
       public static void main(String[] args)
```

Dr. Hanumaraju B, Dept of CS&BS, MITM

```
        {
                B b = new B();
                b.callme();
                b.callmetoo();
        }
    }
```

Notice that no objects of class A are declared in the program. It is not possible to instantiate an abstract class. One other point: class A implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach **to run-time polymorphism** is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Solution:

Explain abstract class and abstract method with example. 6 Marks

Scheme [6 = 6 Marks]

MODULE-4

7. (a). Define package. Explain the steps involved in creating a user-defined package with an example. 7 Marks

Answer:

❑ Java provides a **mechanism for partitioning** the class name space into more **manageable chunks**. This mechanism is the **package**. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are **not accessible by code outside** that package.

❑ You can also define **class members that are exposed only to other members of the same package**. This allows your classes to have **intimate knowledge of each other, but not expose that knowledge** to the rest of the world.

Define Package:

- ❑ To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored.
- ❑ If you omit the **package** statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications.
- ❑ Most of the time, you will define a package for your code.

    This is the general form of the **package** statement:

    package *pkg*;
- ❑ Here, *pkg* is the name of the package. For example, the following statement creates a package called **mypackage**:

    package mypackage;
- ❑ Typically, Java uses file system directories to store packages.

Dr. Honnaraju B, Dept. of CS&BS, MITM

Import Packages:

- 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 import statement:
- **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.

Example Program:

```
/* Package mypackage */
package mypackage;

public class MyPackageClass
{
        public void displayMessage()
        {
                System.out.println("Hello from MyPackageClass in mypack package!");
        }
        // New utility method
        public static int addNumbers(int a, int b)
        {
                return a + b;
    }
}

//Main program outside the mypackage folder
import mypackage.MyPackageClass;
//import mypackage.*;
public class PackageDemo
{
        public static void main(String[] args)
        {
        // Creating an instance of MyPackageClass from the mypackage package
                MyPackageClass myPackageObject = new MyPackageClass();
        // Calling the displayMessage method from MyPackageClass
                myPackageObject.displayMessage();
        // Using the utility method addNumbers from MyPackageClass
                int result = MyPackageClass.addNumbers(5, 3);
                System.out.println("Result of adding numbers: " + result);
         }
}
```

Solution:

Define package. **1 Mark**

With example program, explain the steps to create package. **6 Marks**

Scheme [1 + 6 = 7 Marks]

Dr. Honnaraju B, Dept. of CS&BS, MITM

(b). Write a program that contains one method that will throw an IllegalAccessException and use proper exception handles so that the exception should be printed. 7 Marks

Answer:

```
class Test
{
        static void fun() throws IllegalAccessException
        {
                System.out.println("Inside the function");
                throw new IllegalAccessException("Illegal Exception");
        }
        public static void main(String args[])
        {
                try
                {
                        fun();
                }
                catch(IllegalAccessException e)
                {
                        System.out.println(e);
                }
        }
}
```

Solution:

Write a program to illustrate exception. 7 Marks

Scheme [7 = 7 Marks]

(c). Define an exception. What are the key terms used in exception handling?

Answer: **6 Marks**

An exception is an **abnormal condition that arises in a code sequence at run time**. In other words, an exception is a **run-time error**. A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

When an **exceptional condition arises**, an object representing that exception is created and thrown in the method that **caused the error**. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Various terms used in handling the exception are –

try – A block of source code that is to be monitored for the exception.

catch – The catch block handles the specific type of exception along with the try block. Note that for each corresponding try block there exists the catch block.

finally – It specifies the code that must be executed even though an exception may or may not occur.

throw – This keyword is used to throw a specific exception from the program code.

throws – It specifies the exceptions that can be thrown by a particular method.

This is the general form of an exception-handling block:

```
try
 {
```

Dr. Hontaraju B, Dept. of CS&BS, MITM

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

Solution:

Define exception. **1 Mark**

Explain the terms used in exception handling **6 Marks**

Scheme [1 + 5 = 6 Marks]

OR

8. **(a). Explain the concept of importing packages in Java and provide an example demonstrating the usage of the import statement.** **7 Marks**

Answer:

Import Packages:

- 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 import statement:
- **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.

Example Program:

```
/* Package mypackage */
package mypackage;
public class MyPackageClass
{
    public void displayMessage()
    {
        System.out.println("Hello from MyPackageClass in mypack package!");
    }
    // New utility method
    public static int addNumbers(int a, int b)
    {
        return a + b;
  }
}
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
//Main program outside the mypackage folder
import mypackage.MyPackageClass;
//import mypackage.*;
public class PackageDemo
{
        public static void main(String[] args)
        {
        // Creating an instance of MyPackageClass from the mypackage package
                MyPackageClass myPackageObject = new MyPackageClass();
        // Calling the displayMessage method from MyPackageClass
                myPackageObject.displayMessage();
       // Using the utility method addNumbers from MyPackageClass
                int result = MyPackageClass.addNumbers(5, 3);
                System.out.println("Result of adding numbers: " + result);
         }
}
```

Solution:

Explain the concept of import package. 3 Mark

Write the example program 4 Marks

Scheme [3 + 4 = 7 Marks]

(b). How do you create your own exception class? Explain with a program. 7 Marks

Answer:

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. You may also wish to override one or more of these methods in exception classes that you create.

Exception defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:

Exception( ) Exception(String *msg*)

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to override **toString( )**. Here's why: The version of **toString( )** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString( )**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing a carefully tailored description of the exception to be displayed.

```
        // This program creates a custom exception type.
        class MyException extends Exception
```

Dr. Honnaraju B, Dept of CS&BS, MITM

```
	{
		private int detail;
		MyException(int a)
		 {
			detail = a;
		}
		public String toString()
		{
			return "MyException[" + detail + "]";
		}
	}
	class ExceptionDemo
	{
		static void compute(int a) throws MyException
		{
			System.out.println("Called compute(" + a + ")");
			if(a > 10)
			throw new MyException(a);
			System.out.println("Normal exit");
		}
		public static void main(String[] args)
		 {
			try
			 {
				compute(1);
				compute(20);
			}
			catch (MyException e)
			{
				System.out.println("Caught " + e);
			}
		}
}
```

Solution:

Explain how to create own exception. **4 Marks**
Write an example program. **3 Marks**
Scheme [4 + 3 = 7 Marks]

(c). Demonstrate the working of a nested try block with an example. **6 Marks**

Answer:

The try statement can be **nested**. That is, a **try statement can be inside the block of another try**. Each time a try statement is entered, the context of that **exception is pushed on the stack**. If an inner try statement **does not have a catch handler** for a particular exception, the stack is unwound and the **next try statement's catch handlers are inspected for a match**. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.

If **no catch statement matches**, then the Java run-time system will handle the exception. Here is an example that uses nested try statements:

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
// An example of nested try statements.
class NestTry
{
          public static void main(String[] args)
           {
          try
          {
          int a = args.length;
                 /* If no command-line args are present, the following statement will
                                        generate a divide-by-zero exception. */
          int b = 42 / a;
          System.out.println("a = " + a);
          try
           {
            // nested try block
            /* If one command-line arg is used, then a divide-by-zero exception will be
                                        generated by the following code. */
          if(a==1)
                    a = a/(a-a);    // division by zero
              /* If two command-line args are used, then generate an out-of-bounds
                                                     exception. */
          if(a==2)
           {
                    int[] c = { 1 };
                    c[42] = 99;            // generate an out-of-bounds exception
          }
          }
          catch(ArrayIndexOutOfBoundsException e)
          {
                    System.out.println("Array index out-of-bounds: " + e);
          }
          }
          catch(ArithmeticException e)
          {
                    System.out.println("Divide by 0: " + e);
          }
          }
}
```

Solution:

Demonstrate the working of nested try block   6 Marks

Scheme [6 = 6 Marks]

Dr. Honnaraju B, Dept. of CS&BS, MITM

MODULE-5

---

9. (a). What do you mean by a thread? Explain the different ways of creating threads. 7 Marks

Answer:

Java provides built-in support for **multithreaded programming**. A multithreaded program **contains two or more parts that can run concurrently**. Each part of such a program is **called a thread**, and each thread **defines a separate path of execution**. Thus, multithreading is **a specialized form of multitasking**.

In the most general sense, we create a thread by **instantiating an object of type Thread**. Java **defines two ways in which this can be accomplished**:

1. You can implement the **Runnable interface**.
2. You can extend the **Thread class, itself**.

Implementing Runnable:

❑ The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class **need only implement a single method** called **run( )**, which is declared like this:

public void run()

❑ Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns. After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb*, String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by ***threadName***.

❑ After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** initiates a call to **run( )**. The **start( )** method is shown here:

void start( )

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
        class NewThread implements Runnable
                {
                        Thread t;
                        NewThread()
                        {
                                // Create a new, second thread
                                t = new Thread(this, "Demo Thread");
                                System.out.println("Child thread: " + t);
                        }
                        // This is the entry point for the second thread.
                        public void run()
                         {
                                try
                                {
                                        for(int i = 5; i > 0; i--)
                                        {
                                                System.out.println("Child Thread: " + i);
                                                Thread.sleep(500);
                                        }
                                }
                        catch (InterruptedException e)
                         {
```

Dr. Honnaraju B. Dept. of CS&BS, MITM

```
                System.out.println("Child interrupted.");
            }
            System.out.println("Exiting child thread.");
            }
    }
    class ThreadDemo
    {
            public static void main(String[] args)
            {
                    NewThread nt = new NewThread(); // create a new thread
                    nt.t.start(); // Start the thread
                    try
                    {
                            for(int i = 5; i > 0; i--)
                             {
                                    System.out.println("Main Thread: " + i);
                                    Thread.sleep(1000);
                            }
                    }
                    catch (InterruptedException e)
                    {
                            System.out.println("Main thread interrupted.");
                    }
                    System.out.println("Main thread exiting.");
            }
    }
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

t = new Thread(this, "Demo Thread");

Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object. Inside **main( )**, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin. Next the main thread enters its **for** loop. Both threads continue running, sharing the CPU in single-core systems, until their loops finish.

Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. As before, a call to **start( )** begins execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread
{
        NewThread()
         {
                // Create a new, second thread
                super("Demo Thread");
                System.out.println("Child thread: " + this);
        }
        // This is the entry point for the second thread.
```

Dr. Honnaraju B, Dept of CS&BS, MITM

```
        public void run()
        {
                try
                {
                        for(int i = 5; i > 0; i--)
                        {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                }
                catch (InterruptedException e)
                 {
                        System.out.println("Child interrupted.");
                }
                System.out.println("Exiting child thread.");
        }
    }
    class ExtendThread
     {
        public static void main(String[] args)
         {
                NewThread nt = new NewThread(); // create a new thread
                nt.start(); // start the thread
                try
                 {
                        for(int i = 5; i > 0; i--)
                        {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                }
                 catch (InterruptedException e){
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
    }
```

Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)

Here, *threadName* specifies the name of the thread.

Solution:

Explain two different implementation of threads with example 7 Marks

Scheme [7 = 7 Marks]

(b). What is the need of synchronization? Explain with an example how synchronization is implemented in JAVA. 7 Marks

Answer:

Because multithreading introduces an **asynchronous behavior to your programs**, there must be a way for you to enforce **synchronicity when you need it**. For example, if

Dr. Honnaraju B, Dept. of CS&BS, MITM

you want two threads to communicate and share a complicated data structure, such as a linked list, you need **some way to ensure that they don't conflict with each other**. That is, you must **prevent one thread** from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the monitor.

Synchronization is **easy in Java**, because all objects have their **own implicit monitor associated with them**. To enter an object's monitor, just call a method that has been modified with the **synchronized keyword**. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and **relinquish control of the object to the next waiting thread**, the owner of the monitor simply returns from the synchronized method.

```
// A correct implementation of a producer and consumer.
class Q
{
        int n;
        boolean valueSet = false;
        synchronized int get()
        {
                while(!valueSet)
                try
                {
                        wait();
                }
                catch(InterruptedException e)
                {
                        System.out.println("InterruptedException caught");
                }
                System.out.println("Got: " + n);
                valueSet = false;
                notify();
                return n;
        }
        synchronized void put(int n)
         {
                while(valueSet)
                try
                {
                        wait();
                }
         catch(InterruptedException e)
        {
                System.out.println("InterruptedException caught");
        }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
        }
}
```

Dr. Honnaraju B, Dept. of CS&BS, MITM.

```
class Producer implements Runnable
 {
        Q q;
        Thread t;
        Producer(Q q)
        {
                this.q = q;
                t = new Thread(this, "Producer");
        }
        public void run()
        {
                int i = 0;
                while(true)
                {
                q.put(i++);
                }
        }
}
class Consumer implements Runnable
 {
        Q q;
        Thread t;
        Consumer(Q q)
        {
                this.q = q;
                t = new Thread(this, "Consumer");
        }
        public void run()
        {
                while(true)
                {
                q.get();
                }
        }
}
class PCFixed
{
        public static void main(String[] args)
         {
                Q q = new Q();
                Producer p = new Producer(q);
                Consumer c = new Consumer(q);
                // Start the threads.
                p.t.start();
                c.t.start();
                System.out.println("Press Control-C to stop.");
        }
}
```

Solution:

Explain the need of synchronization.   2 Marks

Dr. Honnaraju B, Dept. of CS&BS, MITM

Write an example program. **5 Marks**

Scheme [2 + 5 = 7 Marks]

(c). Discuss values() and valueOf() methods in Enumerations with suitable examples.

Answer: **6 Marks**

All enumerations automatically contain two predefined methods: **values( )** and **valueOf( )**. Their general forms are shown here:

public static *enum-type* [ ] values( ) public static *enum-type* valueOf(String *str*)

The **values( )** method returns an array that contains a list of the enumeration constants. The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values( )** and **valueOf( )** methods:

```
// Use the built-in enumeration methods. An enumeration of apple varieties.
        enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }
        class EnumDemo2
        {
                public static void main(String[] args)
                 {
                 Apple ap;
                System.out.println("Here are all Apple constants:");        // use values()
                Apple[] allapples = Apple.values();
                 for(Apple a : allapples)
                System.out.println(a);
                System.out.println();                              // use valueOf()
                ap =  Apple.valueOf("Winesap");
                System.out.println("ap contains " + ap);
                 }
}
```

Scheme:

Discuss values() and valueOf() methods **6 Marks**

Scheme [6 = 6 Marks]

OR

10. (a). What is multithreading? Write a program to create multiple threads in JAVA. **7 Marks**

Answer:

Java provides built-in support for **multithreaded programming**. A multithreaded program **contains two or more parts that can run concurrently**. Each part of such a program is **called a thread**, and each thread **defines a separate path of execution**. Thus, multithreading is **a specialized form of multitasking**.

However, there are two distinct types of multitasking: process-based and thread-based. It is important to **understand the difference between the two**. For many readers, process-based multitasking is the more familiar form. A ***process*** is, in essence, a program that is executing. Thus, ***process-based* multitasking** is the feature that allows your computer to run two or more programs concurrently.

Dr. Honnaraju B, Dept. of CS&BS, MITM

In a ***thread-based*** **multitasking environment**, the thread is the smallest unit of **dispatchable code**. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

```
class NewThread implements Runnable
 {
        String name;                     // name of thread
        Thread t;
        NewThread(String threadname)
         {
                name = threadname;
                t = new Thread(this, name);
                System.out.println("New thread: " + t);
        }
        public void run()
         {
                try
                {
                        for(int i = 5; i > 0; i--)
                         {
                                System.out.println(name + ": " + i);
                                Thread.sleep(1000);
                        }
                }
                 catch (InterruptedException e)
                {
                        System.out.println(name + "Interrupted");
                }
        System.out.println(name + " exiting.");
        }
}
class MultiThreadDemo
{
        public static void main(String[] args)
        {
                NewThread nt1 = new NewThread("One");
                NewThread nt2 = new NewThread("Two");
                NewThread nt3 = new NewThread("Three");
                nt1.t.start();
                nt2.t.start();
                nt3.t.start();
                try
                 {
                        Thread.sleep(10000);
                }

                 catch (InterruptedException e)
                {
                        System.out.println("Main thread Interrupted");
                }
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
                    System.out.println("Main thread exiting.");
                }
        }
```

Solution:

Define multithreading. **2 Marks**

Write a program to implement multithread programming. **5 Marks**

Scheme [2 + 5 = 7 Marks]

(b). Explain with an example how inter-thread communication is implemented in JAVA. **7 Marks**

Answer:

Multithreading introduces an **asynchronous behavior to your programs**, there must be a way for you to enforce **synchronicity when you need it**. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need **some way to ensure that they don't conflict with each other**. That is, you must **prevent one thread** from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the monitor.

Synchronization is **easy in Java**, because all objects have their **own implicit monitor associated with them**. To enter an object's monitor, just call a method that has been modified with the **synchronized keyword**. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and **relinquish control of the object to the next waiting thread**, the owner of the monitor simply returns from the synchronized method.

Thread Communication:

```
// A correct implementation of a producer and consumer.
        class Q
        {
                int n;
                boolean valueSet = false;
                synchronized int get()
                {
                        while(!valueSet)
                        try {
                        wait();
                        }
                        catch(InterruptedException e)
                        {
                        System.out.println("InterruptedException caught");
                        }
                        System.out.println("Got: " + n);
                        valueSet = false;
                        notify();
                        return n;
                }
                synchronized void put(int n)
                 {
```

Dr. Honnaraju B, Dept of CS&BS, MITM.

```
        while(valueSet)
        try {
        wait();
        }
         catch(InterruptedException e)
        {
        System.out.println("InterruptedException caught");
        }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
        }
}
class Producer implements Runnable
 {
        Q q;
        Thread t;
        Producer(Q q)
        {
                this.q = q;
                t = new Thread(this, "Producer");
        }
        public void run()
        {
                int i = 0;
                while(true)
                {
                q.put(i++);
                }
        }
}
class Consumer implements Runnable
 {
        Q q;
        Thread t;
        Consumer(Q q)
        {
                this.q = q;
                t = new Thread(this, "Consumer");
        }
        public void run()
        {
                while(true)
                {
                q.get();
                }
        }
}
```

Dr. Honnaraju B, Dept. of CS&BS, MITM

```
class PCFixed
{
        public static void main(String[] args)
         {
                Q q = new Q();
                Producer p = new Producer(q);
                Consumer c = new Consumer(q);
                p.t.start();
                c.t.start();
                System.out.println("Press Control-C to stop.");
        }
}
```

Solution:

Explain access matrix with its implmentation **6 Marks**

Scheme [6 = 6 Marks]

(c). Explain auto-boxing/unboxing in expressions. **6 Marks**

Answer:

In general, **autoboxing and unboxing** take place whenever a conversion into an object or **from an object is required**. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The **outcome of the expression is reboxed**, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.
    class AutoBox
    {
        public static void main(String[] args)
        {
                Integer iOb, iOb2;
                int i;
                iOb = 100;
                System.out.println("Original value of iOb: " + iOb);
                // The following automatically unboxes iOb, performs the increment, and then
                // reboxes the result back into iOb.
                ++iOb;
                System.out.println("After ++iOb: " + iOb);
                // Here, iOb is unboxed, the expression is
                // evaluated, and the result is reboxed and  assigned to iOb2.
                iOb2 = iOb + (iOb / 3);
                System.out.println("iOb2 after expression: " + iOb2);
                // The same expression is evaluated, but the result is not reboxed.
                i = iOb + (iOb / 3);
                System.out.println("i after expression: " + i);
        }
    }
```

Solution:

Explain auto-boxing/unboxing in expressions. **6 Marks**

Scheme [6 = 6 Marks]

GOOD LUCK

Dr. Honnaraju B, Dept. of CS&BS, MITM