

Scheme and Solutions - Improvement Test

|                                       |                                 |               |               |
|---------------------------------------|---------------------------------|---------------|---------------|
| Sub:                                  | <b>Object Oriented Concepts</b> | Code:         | <b>15CS45</b> |
| Date:                                 | Duration: 90 mins               | Max Marks: 50 | Sem: IV       |
|                                       |                                 | Branch:       | <b>ISE</b>    |
| Answer Any <b>FIVE FULL</b> Questions |                                 |               |               |

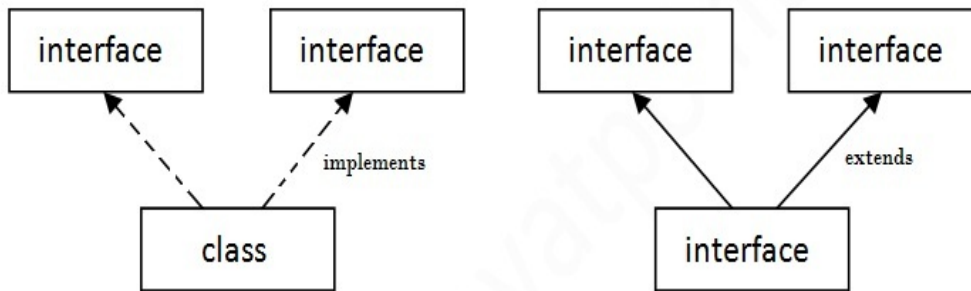
|       |     |     |
|-------|-----|-----|
| Marks | OBE |     |
|       | CO  | RBT |

1 (a) Which is the alternative approach to implement multiple inheritance in Java? [10]  
Explain, with an example.

CO3      L4

**Multiple inheritance in Java by interface**

If a class implements multiple interfaces, or an interface extends multiple Interface that is known as multiple inheritance.



**Multiple Inheritance in Java**

```

interface Printable{
1 void print();
2 }
3 interface Showable{
4 void show();
5 }
6 class A7 implements Printable,Showable{
7 public void print(){System.out.println("Hello");}
8 public void show(){System.out.println("Welcome");}
9
10 public static void main(String args[]){
11     A7 obj = new A7();
12     obj.print();
13     obj.show();
14 }
15 }
16

```

Output:Hello  
Welcome

Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class ambiguity as implementation is provided by the implementation class. For example:

```
1 interface Printable{
2 void print();
3 }
4 interface Showable{
5 void print();
6 }
7
8 class TestInterface3 implements Printable, Showable{
9 public void print(){System.out.println("Hello");}
10     public static void main(String args[]){
11         TestInterface1 obj = new TestInterface1();
12         obj.print();
13     }
14 }
```

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implemen

**Interface inheritance**

A class implements interface but one interface extends another interface .

```
1 interface Printable{
2 void print();
3 }
4 interface Showable extends Printable{
5 void show();
6 }
7 class TestInterface4 implements Showable{
8 public void print(){System.out.println("Hello");}
9 public void show(){System.out.println("Welcome");}
10
11     public static void main(String args[]){
12         TestInterface4 obj = new TestInterface4();
13         obj.print();
14         obj.show();
15     }
16 }
```

2 (a) What is the need of synchronization? Explain with an example, how [10]  
synchronization is implemented in JAVA?

Multithreading is a conceptual programming concept where a program (process)  
is divided into two or more subprograms (process), which can be implemented

CO4

L4

at the same time in parallel. 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. A **process** consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process.

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called thread synchronization. The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

Example without Synchronization

[10]

```

class update
{
    void updatesum(int i)
    {
        Thread t = Thread.currentThread();
        for(int n=1; n<=5; n++)
        {
            System.out.println(t.getName()+" : "+(i+n));
        }
    }
}
class A extends Thread
{
    update u = new update();
    public void run()
    {
        u.updatesum(10);
    }
}
class syntest
{
    public static void main(String args[])
    {
        A a = new A();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
    }
}

```

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

|     |    |
|-----|----|
| CO5 | L4 |
|-----|----|

|     |    |
|-----|----|
| CO3 | L3 |
|-----|----|

### Example with Synchronization

```
class update
{
    synchronized void updatesum(int i)
    {
        Thread t = Thread.currentThread();
        for(int n=1; n<=5; n++)
        {
            System.out.println(t.getName()+" : "+(i+n))
        }
    }
}
class A extends Thread
{
    update u = new update();
    public void run()
    {
        u.updatesum(10);
    }
}
class syntest
{
    public static void main(String args[])
    {
        A a = new A();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
    }
}
```

3 (a) What are applets? Explain the different stages in the life cycle of an applet.

[10] CO5 L4

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()**

**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.

**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()**.

**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.

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

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

4 (a) Define Exception. Demonstrate the working of exception handling and nested try blocks, with suitable examples. [10]

CO3

L3

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 {

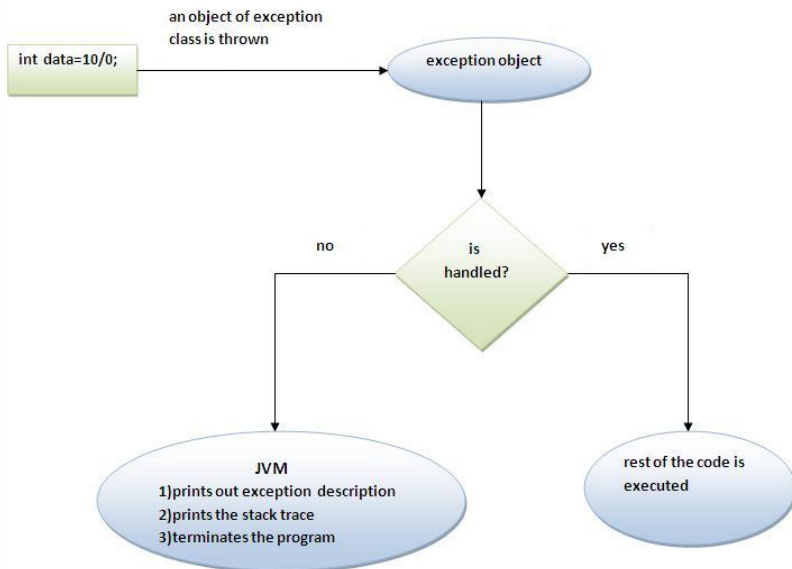
System.out.println("rest of the code..."); }

}}

```

### Output:

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



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.



- 5 (a) What is the difference between Method Overloading and Method Overriding.[10]  
Explain with suitable examples.

**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 {**

CO2

L2

```
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 overriding method in the subclass 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.

6 (a) Explain the constructors in Java. How is it different from other member functions. [06]

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;
```

CO2

L4

```

}

// 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().

[04]

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

CO3

L4

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

super is used to refer immediate parent class instance variable.

super() is used to invoke immediate parent class constructor.

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() {  
  
        System.out.println("Bike is created"); }  
  
    public static void main(String args[]) {  
  
        Bike b=new Bike(); }  
  
}
```

**super();//will invoke parent class constructor**

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.

