CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 2– Sep 2021

Sub:	Object Oriented Programming with Java					Sub Code:	**20MCA22**	Branch:	MCA
Date:	20/09/2021	Duration:	90 min's	Max Marks:	50	Sem	II		OBE

Q1 What are interfaces? What are their benefits? Explain how it is implemented in JAVA with a suitable example. (10 Marks)

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

The interface keyword is used to declare an interface. Here is a simple example to declare an interface −

```
public interface NameOfInterface
{
// Any number of final, static fields
// Any number of abstract method declarations\
}
```

Interfaces have the following properties −

 An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
 Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
 Methods in an interface are implicitly public.

```
interface Animal
{
public void eat();
public void travel(); }
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract. A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
public class MammalInt implements Animal {
public void eat() {
System.out.println("Mammal eats");
}
public void travel() {
System.out.println("Mammal travels");
}
public int noOfLegs() {
return 0;
}
public static void main(String args[]) {
MammalInt m = new MammalInt();
m.eat();
m.travel();
}
}
```

This will produce the following result −

Output

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed –

2. Explain how to create custom exceptions, Given an Example

User-defined exceptions in Java are also known as Custom Exceptions.

Steps to create a Custom Exception with an Example

- CustomException class is the custom exception class this class is extending Exception class.
- Create one local variable message to store the exception message locally in the class object.
- We are passing a string argument to the constructor of the custom exception object. The constructor set the argument string to the private string message.
- toString() method is used to print out the exception message.

- We are simply throwing a CustomException using one try-catch block in the main method and observe how the string is passed while creating a custom exception. Inside the catch block, we are printing out the message.

Example

```
class CustomException extends Exception {
   String message;
   CustomException(String str) {
      message = str;
   }
   public String toString() {
      return ("Custom Exception Occurred : " + message);
   }
}
public class MainException {
   public static void main(String args[]) {
      try {
         throw new CustomException("This is a custom message");
      } catch(CustomException e) {
         System.out.println(e);
      }
   }
}
```

Output

```
Custom Exception Occurred : This is a custom message
```

3. **What is an exception? Explain the exception handling mechanism with suitable example**

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. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.

Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

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

```
// 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
}
```

4. What is multithreading? Write a JAVA program to create multithreads in JAVA by implementing runnable interface. (10 Marks)

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.
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( ) executes 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);
t.start(); // Start the thread
}
// 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) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new 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.");
}
}
```

The output produced by this program is as follows

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

5. With a suitable programming example explain inter-thread communication.

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 is achieved is called synchronization. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time.

When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. You can synchronize your code in two ways.

- Using Synchronized Methods
- The synchronized Statement

While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add synchronized to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a synchronized block.

This is the general form of the synchronized statement:

```
synchronized(object) {
// statements to be synchronized
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a
member of object occurs only after the current thread has successfully entered object's monitor. Here is an alternative
version of the preceding example, using a synchronized block within the run( ) method:

```
// This program uses a synchronized block.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {

System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
```

```
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( )
method. Output

```
[Hello]
[Synchronized]
[World]
```

6.

Write the following java program to creat package and import it in other program :

i)Create a package called shape

ii)Write a class called Triangle.java in shape package. Triangle.java should calculate the area of a triangle

iii)Compile and import shape

```
package shape;
public class triangle {
   public void display3()
   {
      System.out.println("formula of triangle =0.5*length*breadth");
   }
 }
```

```
package shape;
```

```
public class square {
   public void display2()
   {
      System.out.println("formula of Square =lengthv*width")
   }
}
```

```
package shape;
public class circle {
   public void display1()
   {
      System.out.println("formula of circle is 3.142*r*r");
   }
}
```

```
import  shape.*;
public class prgrm7
{
   public static void main(String arg[])
   {
      circle ob1 = new circle();
      square ob2 =new square();
      triangle ob3 =new triangle();
      ob1.display1();
      ob2.display2();
      ob3.display3();
   }
}
```

7. What is autoboxing? Illustrate with a programming example. (06 Marks)

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

```
class AutoBox2 {
// Take an Integer parameter and return
// an int value;
static int m(Integer v) {
return v ; // auto-unbox to int
}
```

```
public static void main(String args[]) {
// Pass an int to m() and assign the return value
// to an Integer. Here, the argument 100 is autoboxed
// into an Integer. The return value is also autoboxed
// into an Integer.
Integer iOb = m(100);
System.out.println(iOb);
}
}
```
This program displays the following result:
100

In the program, notice that m( ) specifies an Integer parameter and returns an int result. Inside main( ), m( ) is passed the value 100. Because m( ) is expecting an Integer, this value is automatically boxed. Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed. Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.

8. What is meant by generic class? Illustrate with a programming example.

At its core, the term generics means parameterized types. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.

```
class Gen<T>
{ T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.

Gen(T o)
{ ob = o;
}
// Return ob.
T getob()
{ return ob;
}
// Show type of T.
void showType()
{ System.out.println("Type of T is " +
ob.getClass().getName());
}
}

class GenDemo
{ public static void main(String args[])
{ // Create a Gen reference for Integers.
Gen<Integer> iOb;
```

```
// Create a Gen<Integer> object and assign its
// reference to iOb. Notice the use of autoboxing
// to encapsulate the value 88 within an Integer object.
iOb = new Gen<Integer>(88);
// Show the type of data used by iOb.
iOb.showType();
// Get the value in iOb. Notice that
// no cast is needed.
int v = iOb.getob();
System.out.println("value: " + v);
System.out.println();
// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String>("Generics Test");
// Show the type of data used by strOb.
strOb.showType();
// Get the value of strOb. Again, notice
// that no cast is needed.
String str = strOb.getob();
System.out.println("value: " + str);
}
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

9 .Explain Arraylist class and Linkedlist class with an example

Linked List is a part of the Collection framework present in java.util package. This class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.

Example: The following implementation demonstrates how to create and use a linked list.

- Java

```
import java.util.*;

public class Test {

    public static void main(String args[])
    {
        // Creating object of the
        // class linked list
        LinkedList<String> ll
```

```
            = new LinkedList<String>();

        // Adding elements to the linked list
        ll.add("A");
        ll.add("B");
        ll.addLast("C");
        ll.addFirst("D");
        ll.add(2, "E");

        System.out.println(ll);

        ll.remove("B");
        ll.remove(3);
        ll.removeFirst();
        ll.removeLast();

        System.out.println(ll);
    }
}
```

- **ArrayList** [ArrayList class implements List interface] →
  - ✓ Supports dynamic array that can grow dynamically.
  - ✓ Can contain duplicate elements.
  - ✓ Heterogeneous objects are allowed.(Except TreeSet & TreeMap)
  - ✓ Insertion order preserved & Provides more powerful insertion and search mechanisms than arrays.
  - ✓ Null insertion is possible.

Syntax:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, newInteger(42));
int total=list.get(0).intValue();
```

- **Eg.:**

```
import java.util.*;
public class ArrayListTest{
    public static void main(String[] args) {
        ArrayList<String> test = new ArrayList<String>();
        String s = "hi";
        test.add("hello");
        test.add(s);
        test.add(s+s);
        System.out.println(test);      // [hello, hi, hihi]
        System.out.println(test.size());    // 3
        System.out.println(test.contains(42));  // false
        System.out.println(test.contains("hihi"));  // true
        test.remove("hi");
        System.out.println(test.size());    // 2
    }
}
```

10.Write a short note on i) URL Connection ii) Collection interface

URL Connection

The **Java URLConnection** class represents a communication link between the URL and the application. It can be used to read and write data to the specified resource referred by the URL.

What is the URL?

- URL is an abbreviation for Uniform Resource Locator. An URL is a form of string that helps to find a resource on the World Wide Web (WWW).
- URL has two components:

1. The protocol required to access the resource.
2. The location of the resource.

How to get the object of URLConnection Class

The openConnection() method of the URL class returns the object of URLConnection class.

Syntax:

```
public URLConnection openConnection()throws IOException{}
```

Displaying Source Code of a Webpage by URLConnecton Class

The URLConnection class provides many methods. We can display all the data of a webpage by using the getInputStream() method. It returns all the data of the specified URL in the stream that can be read and displayed.

Example of Java URLConnection Class

```
1. import java.io.*;
2. import java.net.*;
3. public class URLConnectionExample {
4. public static void main(String[] args){
5. try{
6. URL url=new URL("http://www.javatpoint.com/java-tutorial");
7. URLConnection urlcon=url.openConnection();
8. InputStream stream=urlcon.getInputStream();
9. int i;
10. while((i=stream.read())!=-1){
11. System.out.print((char)i);
12. }
13. }catch(Exception e){System.out.println(e);}
14. }
15. }
```

Collection interface

COLLECTIONS….

- **Collection** → Group of objects
  - → To represent a group of "individual objects" as a single entity.
  - → Root interface of entire collection framework.

  - There is no concrete class which implements Collection interface directly.

  - ***Consists of 3 parts*** :
    1. Core Interfaces.
    2. Concrete Implementation
    3. Algorithms such as searching and sorting.

- **ADVANTAGES →**
  - ✓ Reduces programming Effort.
  - ✓ Increases performance.
  - ✓ Provides interoperability between unrelated APIs
  - ✓ Reduces the effort required to learn APIs
  - ✓ Reduces the effort required to design and implement APIs
  - ✓ Fosters Software reuse.

- **Remember Interface / Class & Child Class →**