CMR INSTITUTE OF TECHNOLOGY

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

Internal Assessment Test – II

Sub:	OBJECT ORIENTED CONCEPTS							Code:	15CS45
Date:	19/ 04/ 2018	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	CSE (All sec)

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1 (a)	Define Synchronization? Explain how Synchronization is achieved in JAVA?	[10]	CO4	L3
2 (a)	Briefly explain the role of interface in implementing Multiple Inheritance in JAVA.	[05]	CO3	L3
(b)	Explain function overloading with example.		CO1	L3
3 (a)	Define Custom Exception? Explain How Custom Exception is created with an example.	[10]	CO3	L3
4 (a)	Define Package? Describe the various levels of access protection for Packages and their implication.	[10]	CO3	L3
5(a)	What you mean by Thread? Explain how thread can be created in JAVA?	[10]	CO4	L3
6(a)	Define Reference variable? Explain. Write C++ program to swap two integers. Display the values Before and after swapping using reference variable.	[05]	CO1	L2
(b)	Differentiate between Procedure Oriented Programming and Object Oriented programming.	[05]	CO1	L2
7 (a)	Define Inline function? Explain with an example.	[10]	CO2	L3
	b) Explain how to define member function outside class in C++.			

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1:	Explain the object oriented concepts	2	1	2	0	1	0	0	0	0	0	1	0
CO2:	Explain Classes, objects ,constructor	2	1	2	0	1	0	0	0	0	0	1	1
CO3:	Implement inheritance, interfaces and exception in java	2	1	2	0	1	0	0	0	0	0	0	0
CO4:	Implement Multi threaded programming and event handling in java	2	1	2	0	1	0	0	0	0	0	0	0
CO5:	Develop computer programs to solve real world problems in java	2	1	2	0	1	0	0	0	0	0	0	0
CO6:	Develop GUI interfaces using Applet and swings	2	1	2	0	1	0	0	0	0		0	0

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1 - *Engineering knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 - *Conduct investigations of complex problems*; PO5 - *Modern tool usage*; PO6 - *The Engineer and society*; PO7- *Environment and sustainability*; PO8 – *Ethics*; PO9 - *Individual and team work*; PO10 - *Communication*; PO11 - *Project management and finance*; PO12 - *Life-long learning*

Solution:

1a. Define Synchronization? Explain how Synchronization is achieved in JAVA?

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*

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods

```
// This program is not synchronized.
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();
}
public void run() {
target.call(msg);
}
}
class Synch {
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 is the output produced by this program:
```
Hello[Synchronized[World]
]
]
```

After **synchronized** has been added to **call( )**, the output of the program is as follows:
```
[Hello]
[Synchronized]
[World]
```

The synchronized Statement

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");
}
}
        }
```

2a. briefly explain the role of interface in implementing Multiple Inheritance in JAVA.

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no 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. TestInterface3 obj = new TestInterface3();
12. obj.print();
13.  }
14. }
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```
15. interface Printable{
```

```
16. void print();
17. }
18. interface Showable extends Printable{
19. void show();
20. }
21. class TestInterface4 implements Showable{
22. public void print(){System.out.println("Hello");}
23. public void show(){System.out.println("Welcome");}
24.
25. public static void main(String args[]){
26. TestInterface4 obj = new TestInterface4();
27. obj.print();
28. obj.show();
29.  }
30. }
```

2b. Explain function overloading with example

In C++ programming, two functions can have same name if number and/or type of arguments passed are different.

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }

int test(int a) { }

float test(double a) { }

int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code

int test(int a) { }

double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example 1: Function Overloading

```
#include <iostream>
using namespace std;
void display(int);
void display(float);
void display(int, float);
int main() {
   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);
   return 0;
}
void display(int var) {
   cout << "Integer number: " << var << endl;
}
void display(float var) {
   cout << "Float number: " << var << endl;
}
void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5

Float number: 5.5
```

```
Integer number: 5 and float number: 5.5
```

3a. Define Custom Exception? Explain How Custom Exception is created with an example.

User-defined Custom Exception in Java

Java provides us facility to create our own exceptions which are basically derived classes of Exception. For example MyException in below code extends the Exception class.
We pass the string to the constructor of the super class- Exception which is obtained using "getMessage()" function on the object created.

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

Run on IDE
Output:

```
Caught

GeeksGeeks
```

In the above code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

```
// A Class that represents use-defined expception
class MyException extends Exception
{

}

// A Class that uses above MyException
public class setText
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException();
        }
        catch (MyException ex)
        {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
}
```

Run on IDE

Output:

```
Caught
null
```

4a. Define Package? Describe the various levels of access protection for Packages and their implication.

Access modifiers define the scope of the class and its members (data and methods). For example, private members are accessible within the same class members (methods). Java provides many levels of security that provides the visibility of members (variables and methods) within the classes, subclasses, and packages.

Packages are meant for encapsulating, it works as containers for classes and other subpackages. Class acts as containers for data and methods. There are four categories, provided by Java regarding the visibility of the class members between classes and packages:

1. Subclasses in the same package
2. Non-subclasses in the same package
3. Subclasses in different packages
4. Classes that are neither in the same package nor subclasses

The three main access modifiers *private, public* and *protected* provides a range of ways to access required by these categories.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
same package subclass	No	Yes	Yes	Yes
same package non - subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package mon-subclass	No	No	No	Yes

A class can have only two access modifier, one is default and another is public. If the class has default access then it can only be accessed within the same package by any other code. But if the class has public access then it can be access from any where by any other code.

Example:

```
//PCKG1_ClassOne.java
package pckg1;
public class PCKG1_ClassOne{
int a = 1;
private int pri_a = 2;
protected int pro_a = 3;
public int pub_a = 4;
public PCKG1_ClassOne() {
System.out.println("base class constructor called");
System.out.println("a = " + a);
System.out.println("pri_a = " + pri_a);
System.out.println("pro_a "+ pro_a);
System.out.println("pub_a "+ pub_a);
}
}
```

The above file PCKG1_ClassOne belongs to package pckg1, and contains data members with all access modifiers.

*//PCKG1*_ClassTwo.java

```
package pckg1;
class PCKG1_ClassTwo extends PCKG1_ClassOne {
PCKG1_ClassTwo() {
System.out.println("derived class constructor called");
System.out.println("a = " + a);
II accessible in same class only
II System.out.println("pri_a = " + pri_a);
System.out.println("pro_a "+ pro_a);
System.out.println("pub_a =" + pub_a);
```

```
}
}
```

The above file PCKG1_ClassTwo belongs to package pckg1, and extends PCKG1_ClassOne, which belongs to the same package.

*//PCKG1*_ClasslnSamePackage

```
package pckg1;
class PCKG1_ClassInSamePackage {
PCKG1_ClassInSamePackage() {
PCKG1_ClassOne co = new PCKG1_ClassOne();
System.out.println("same package class constructor called");
System.out.println("a = " + co.a);
II accessible in same class only
II System.out.println("pri_a = " + co.pri_a);
System.out.println("pro_a "+ co.pro_a);
System.out.println("pub_a = " + co.pub_a);
}
}
```

The above file PCKG1_ClassInSamePackage belongs to package pckg1, and having an instance of PCKG1_ClassOne.

```
package PCKG1;
//Demo package PCKG1
public class DemoPackage1 {
public static void main(String ar[]) {
PCKG1_ClassOne obl = new PCKG1_ClassOne();
PCKG1_ClassTwo ob2 = new PCKG1_ClassTwo();
PCKG1_ClassInSamePackage ob3 = new PCKG1_ClassxnSamePackage();
}
}
```

The above file DemoPackageI belongs to package pckgI, and having an instance of all classes in pckg1.

```
package pckg2;
class PCKG2_ClassOne extends PCKG1.PCKG1_ClassOne {
PCKG2_ClassOne() {
System.out.println("derived class of other package constructor
```

```
called");
II accessible in same class or same package only
II System.out.println("a = " + a);
II accessible in same class only
II System.out.println("pri_a = " + pri_a);
System.out.println("pro_a = " + pro_a);
System.out.println("pub_a = " + pub_a);
}
}
```

The above file PCKG2_ClassOne belongs to package pckg2. extends PCKG I_ClassOne, which belongs to PCKG1, and it is trying to access data members of the class PCKGI_ClassOne.

*IIPCKG2_* ClassInOtherPackage

```
package pckg2;
class PCKG2_ClassInOtherPackage {
PCKG2_ClassInOtherpackage() {
PCKG1.PCKG1_ClassOne co = new PCKG1.PCKG1_ClassOne();
System.out.println("other package constructor");
II accessible in same class or same package only
II System.out.println("a ,= " + co.a);
II accessible in same class only
II System.out.println("pri_a = " + co.pri_a);
II accessible in same class, subclass of same or other package
II System.out.println("pro_a = " + co.pro_a);
System.out.println("pub_a = " + co.pub_a);
}
}
```

The above file PCKG2_ClassInOtherPackage belongs to package pckg2, and having an instance of PCKG LClassOne of package pckg I, trying to access its some data members.

```
II Demo package pckg2.
package pckg2;
public class DemoPackage2 {
public static void main(String ar[]) {
PCKG2_ClassOne obl = new PCKG2_ClassOne();
PCKG2_ClassInOtherPackage ob2 = new PCKG2_ClassInOtherPackage();
```

}
}

The above file DemoPackage2 belongs to package pckg2, and having an instance of all classes of pckg2 .

5a. What you mean by Thread? Explain how thread can be created in JAVA?

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

Mutual Exclusive

Synchronized method.

Synchronized block.

static synchronization.

Cooperation (Inter-thread communication in java)

Class Table{

```
void printTable(int n){//method not synchronized
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

6a. Define Reference variable? Explain. Write C++ program to swap two integers. Display the values Before and after swapping using reference variable.

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

```
#include <iostream>

using namespace std;
```

```
int main()
{
   int a = 5, b = 10, temp;
   cout << "Before swapping." << endl;
   cout << "a = " << a << ", b = " << b << endl;
   temp = a;
   a = b;
   b = temp;
   cout << "\nAfter swapping." << endl;
   cout << "a = " << a << ", b = " << b << endl;
   return 0;
}
```

Output

```
Before swapping.

a = 5, b = 10

After swapping.

a = 10, b = 5
```

6b. Differentiate between Procedure Oriented Programming and Object Oriented programming.

1. Overloading happens at **compile-time** while Overriding happens at **runtime**: The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. **Static binding** is being used for overloaded methods and **dynamic binding** is being used for overridden/overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.

6. private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type.
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

7a. Define Inline function? Explain with an example.

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime. Using the inline keyword is simple, just put it before the name of a function. Then, when you use that function, pretend it is a non-inline function: #include <iostream>

```
 using namespace std;
inline void hello()
{
cout<<"hello";
}
 int main() {
hello(); //Call it like a normal function
 return 0;
}
```

However, once the program is compiled, the call to hello(); will be replaced by the code making up the function. So the main() will look like this after the compile: int main() { cout<<"hello"; return 0; }

7b. Explain how to define member function outside class in C++

Member functions can be defined within the class definition or separately using **scope resolution operator, :** −. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume()** function as below −

```
class Box {
  public:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box

    double getVolume(void) {
      return length * breadth * height;
    }
};
```

If you like, you can define the same function outside the class using the **scope resolution operator** (::) as follows −

```
double Box::getVolume(void) {
  return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows −

```
Box myBox;          // Create an object
```

```
myBox.getVolume();  // Call member function for the object
```

```
#include <iostream>

using namespace std;

class Box {
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void) {
   return length * breadth * height;
}

void Box::setLength( double len ) {
   length = len;
}
void Box::setBreadth( double bre ) {
   breadth = bre;
```

```
}
void Box::setHeight( double hei ) {
   height = hei;
}

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
```