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

13MCA32

Third Semester MCA Degree Examination, Dec.2016/Jan.2017
Programming Using JAVA

Time: 3 hrs. Max. Marks:100

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

1 a. Discuss the various primitive data types used in JAVA. Give suitable example. **(08 Marks)**

b. Explain implicit and explicit type conversion. Give suitable example. **(06 Marks)**

c. Evaluate the following expression where a = 5, b =10 and c = 6

i) $a+b-c*b/a-c+a-b$

ii) $((a-b)*c)>(c-a*b)\,||\,(b+c*a)<(b/a+c)$

iii) $(a-b)*c+c*b-a+15*(4-c)/3$ **(06 Marks)**

2 a. Write a JAVA program to evaluate the following $1^2/3! + 2^3/4! + 3^4/5! + ------ + n^{n+1}/(n+2)!$ **(10 Marks)**

b. What is a class? Give the general form of a class. **(05 Marks)**

c. Explain the following with suitable example
i) new operator ii) this keyword. **(05 Marks)**

3 a. Write a JAVA program to find the sum and average of even numbers in a given matrix. Print the result with the suitable heading along with the given matrix in the matrix form. **(08 Marks)**

b. Write a JAVA program to find the sum and average of the element i.e {8, 6, 4, 2, 1, 3, 17, 18, 15} using enhanced for loop. **(06 Marks)**

c. Explain any three functions that operate on string. **(06 Marks)**

4 a. Explain method overloading and method overriding. Give suitable example. **(08 Marks)**

b. Explain the following : i) super ii) final **(06 Marks)**

c. What is abstract class? Illustrate with a programming example. **(06 Marks)**

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

b. Explain different access specifiers used in JAVA. Give suitable example. **(10 Marks)**

6 a. What is multithreading? Write a JAVA program to create multiple threads in JAVA by implementing runnable interface. **(08 Marks)**

b. With a suitable programming example explain inter-thread communication. **(06 Marks)**

c. What is an exception? How exceptions can be handled in JAVA? **(06 Marks)**

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

b. What is meant by generic class? Illustrate with a programming example. **(06 Marks)**

c. What is an applet? With the help of a skeleton, explain the life cycle of an applet. **(08 Marks)**

8 a. Write a JAVA swing program to create a frame which contains 2 buttons named "Alpha" and "Beta". When either of button is pressed, it should display "Alpha is pressed" and "Beta is pressed". **(10 Marks)**

b. Write a JAVA program which demonstrates utilities of linked list class. **(10 Marks)**

* * * * *

1a. Discuss the various primitive data types used in JAVA. Give suitable example. (08 Marks)

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types. These can be put in four groups:
• Integers- This group includes byte, short, int, and long, which are for whole-valued signed numbers.
• Floating- Point numbers This group includes float and double, which represent numbers with fractional precision.
• Characters- This group includes char, which represents symbols in a character set, like letters and numbers.
• Boolean- This group includes boolean, which is a special type for representing true/false values.
You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

```
// Demonstrate boolean values.
class BoolTest {
        public static void main(String args[]) {
                boolean b;
                b = false;
                System.out.println("b is " + b);
                b = true;
                System.out.println("b is " + b);
                // a boolean value can control the if statement if(b) System.out.println("This is executed.");
                b = false;
                if(b)     System.out.println("This is not executed.");
                // outcome of a relational operator is a boolean value
                System.out.println("10 > 9 is " + (10 > 9));
        }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

1b. Explain implicit and explicit type conversion. Give suitable example. (06 Marks)

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;

byte b = 50;

byte c = 100;

int d = a * b / c;
```

The result of the intermediate term a * b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression. This means that the subexpression a * b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte. As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;

b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type. In cases where you understand the consequences of overflow, you should use an explicit cast, such as

byte b = 50;

b = (byte)(b * 2);

which yields the correct value of 100.

1c. Evaluate the following expression where a=5, b=10 and c=6

i) a+b-c*b/a-c+a-b
ii) ((a-b)*c)>(c-a*b)||(b+c*a)<(b/a+c)
iii) (a-b)*c+c*b-a+15*(4-c)/3 **(06 Marks)**

i) a+b-c*b/a-c+a-b = -8
5+10-6*10/5-6+5-10
5+10-60/5-6+5-10
5+10-12-6+5-10
15-12-6+5-10
3-6+5-10
-3+5-10
2-10
-8

ii) ((a-b)*c) > (c-a*b) || (b+c*a) < (b/a+c) = true
((5-10)*6) > (6-5*10) || (10+6*5) < (10/5+6)
(-5*6) > (6-50) || (10+30) < (2+6)
-30 > -44 || 40 < 8
True || false
True

iii) (a-b)*c+c*b-a+15*(4-c)/3 = 15
(5-10)*6+6*10-5+15*(4-6)/3
-5*6+6*10-5+15*-2/3
-30+60-5-30/3
-30+60-5-10
30-15
15

2a. Write a JAVA program to evaluate the following

$1^2/3!+2^3/4!+3^4/5!+-------+n^{n+1}/(n+2)!$ (10 Marks)

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class exam
{       public static void main(String[] args) throws IOException
        {       int n,sum=0;
                BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
                System.out.println("Enter the value of n");
                n=Integer.parseInt(br.readLine());
                for(int i=n; i>=1; i--)
                        sum=sum+(i^(i+1)/exam.fact(i+2));
                System.out.println("Sum of series 1^2/3!+2^3/4!+3^4/5!+-------+n^n+1/(n+2)! is"          +sum);
        }
        public static int fact(int n)
        {       int output;
                if(n==1)
                {
                        return 1;
                }
                output = exam.fact(n-1)* n;
                return output;
        }
}
```

2b.What is a class? Give the general form of a class. (05 Marks)

Class − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.
A class is a blueprint from which individual objects are created. Following is a sample of a class.

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

A class can contain any of the following variable types.

Local variables − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Class variables − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Constructors: When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class. Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor −

```
public class Puppy {
  public Puppy() {
  }
  public Puppy(String name) {
    // This constructor has one parameter, name.
  }
}
```

Creating an Object: As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects. There are three steps when creating an object from a class

- Declaration − A variable declaration with a variable name with an object type.
- Instantiation − The 'new' keyword is used to create the object.
- Initialization − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object −

```
public class Puppy {
  public Puppy(String name) {
    // This constructor has one parameter, name.
    System.out.println("Passed Name is :" + name );
  }
  public static void main(String []args) {
    // Following statement would create an object myPuppy
    Puppy myPuppy = new Puppy( "tommy" );
  }
}
```

If we compile and run the above program, then it will produce the following result −

Output

```
Passed Name is :tommy
```

2c. Explain the following with suitable example

i) New operator ii) this keyword (05 Marks)

i) new operator: dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. As just explained, the new operator dynamically allocates memory for an object. It has this general form:

class-var = new classname( );

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class.

ii) this keyword : The ‘this’ keyword is used for two purposes

- It is used to point to the current active object.

- Whenever the formal parameters and data members of a class are similar, to differentiate the data members of a class from formal arguments the data members of a class are preceeded with 'this'.

This(): It is used for calling current class default constructor from current class parameterized constructor.

This(…): It is used for calling current class parameterized constructor from other category constructors of the same class.

```
Ex:  class  Test
     {
                int a,b;
                Test()
                {
                        This(10);
System.out.println("I am from default constructor");
a=1;
b=2;
System.out.println("Value of  a="+a);
System.out.println("Value of b="+b);
                }
                Test(int x)
                {
                        This(100,200);
System.out.println("I am from parameterized constructor");
a=b=x;
System.out.println("Value of  a="+a);
System.out.println("Value of b="+b);
                }
                Test(int a,int b)
                {
System.out.println("I am from double parameterized constructor")
this.a=a+5;
This.b=b+5;
System.out.println("Value of  instance variable a="+this.a);
System.out.println("Value of instance variable b="+this.b);
System.out.println("Value of  a="+a);
System.out.println("Value of b="+b);
                }
}
Class TestDemo3
{
        Public static void main(String k[])
        {
                Test t1=new Test();
        }
}
```

3a. Write a JAVA program to find the sum and average of even numbers in a given matrix. Print the result with the suitable heading along with the given matrix in the matrix form. (08 Marks)

```
import java.io.BufferedReader;
import java.io.IOException;
```

```
import java.io.InputStreamReader;

public class exam
{	public static void main(String[] args) throws IOException
	{
		int x,y,sum=0,no=0;
		BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
		System.out.println("Enter the number of rows");
		x=Integer.parseInt(br.readLine());
		System.out.println("Enter the number of columns");
		y=Integer.parseInt(br.readLine());
		int a[][]= new int[x][y];
		System.out.println("Enter" + (x*y) + "elements");
			for(int m=0;m<x;m++)
			{	for(int n=0;n<y;n++)
				{
					a[m][n]=Integer.parseInt(br.readLine());
				}
			}
		for(int [] i:a)
		{	for(int j:i)
			{
				System.out.print(j+"    ");
				if(j%2==0)
				{
					sum=sum+j;
					no++;
				}
			}
			System.out.println();
		}
		System.out.println("The sum of even  numbers is :" +sum);
		System.out.println("The average is : "+(sum/no));
	}
}
```

3b. Write a JAVA program to find the sum and average of the element i.e {8,6,4,2,1,3,17,18,15} using enhanced for loop. (06 Marks)

```
public class exam
{
	public static void main(String[] args)
	{
		int a[]={8,6,4,2,1,3,17,18,15};
		int avg=0,sum=0;;
		for(int i:a)
			sum=sum+i;
		System.out.println("Sum is: "+sum);
		System.out.println("Average is:" + (sum/a.length));
```

```
        }
    }
```

3c. Explain any three functions that operate on string. (06 Marks)

- **String Length**

The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method. It has this general form:
int length( )
The following fragment prints "3", since there are three characters in the string s:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

- **charAt( )**

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method. It has this general form:
char charAt(int where)
Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value "b" to ch.

- **concat( )**

You can concatenate two strings using concat( ), shown here:
String concat(String str)
This method creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string "onetwo" into s2. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

4a) Explain method overloading and method overriding. Give suitable example. (08 Marks)

Method Overloading:
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. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.
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. Here is a simple example that illustrates 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);
        }
    }
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, 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.

Method Overriding:

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 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. 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[]) {
        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.

4b.Explain the following: i) super ii) final (06 Marks)

i) super

The super keyword is similar to this keyword. Following are the scenarios where the super keyword is used.

- It is used to differentiate the members of superclass from the members of subclass, if they have same names.
- It is used to invoke the superclass constructor from subclass.

Differentiating the Members: If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable
super.method();
```

Invoking Superclass Constructor: If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

ii) final

The keyword final has three uses. First, it can be used to create the equivalent of a named constant. A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant. For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

The other two uses of final apply to inheritance. Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
```

```
}
}
```

Because meth( ) is declared as final, it cannot be overridden in B. If you attempt to do so, a compile-time error will result.

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with final methods. Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as final.

4c.What is abstract class? Illustrate with a programming example. (06 Marks)

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present. Any class that contains one or more abstract methods must also be declared abstract. 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 itself declared abstract. 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[]) {
                B b = new B();
                b.callme();
                b.callmetoo();
        }
}
```

Notice that no objects of class A are declared in the program. As mentioned, 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.

5a.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 −

5b. Explain different access specifiers used in JAVA. Give suitable example. (10 Marks)

Packages add another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
• Subclasses in the same package
• Non-subclasses in the same package
• Subclasses in different packages
• Classes that are neither in the same package nor subclasses
The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. Table below sums up the interactions. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

	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 non-subclass	No	No	No	Yes

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, p1 and p2. The source for the first package defines three classes: Protection, Derived, and SamePackage. The first class defines four int variables in each of the legal protection modes. The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public. Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access. The second class, Derived, is a subclass of Protection in the same package, p1. This grants Derived access to every variable in Protection except for n_pri, the private one. The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri.

```
//This is file Protection.java:
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
//This is file Derived.java:
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
//This is file SamePackage.java:
package p1;
class SamePackage {
SamePackage() {
```

```
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, p2. The two classes defined in p2 cover the other two conditions that are affected by access control. The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection's variables except for n_pri (because it is private) and n, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class OtherPackage has access to only one variable, n_pub, which was declared public.

```
//This is file Protection2.java:
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
//This is file OtherPackage.java:
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
```

```
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

6a.What is multithreading? Write a JAVA program to create multithreads in JAVA by implementing runnable interface. (08 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.
```

6b.With a suitable programming example explain inter-thread communication. (06 Marks)

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]
```

6c.What is an exception? How exceptions can be handled in JAVA? (06 Marks)

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
}
```

7a. 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.

7b. What is meant by generic class? Illustrate with a programming example. (06 Marks)

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

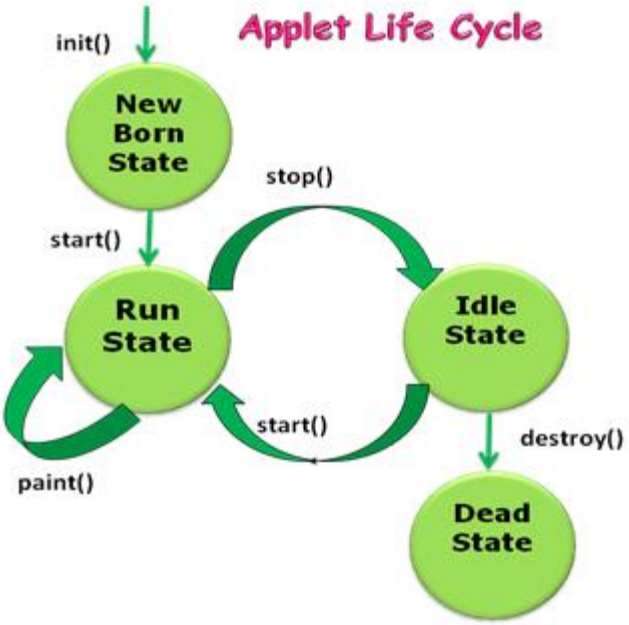
7c. What is an applet? With the help of a skeleton, explain the life cycle of an applet. (08 Marks)

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.

- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet



Four methods in the Applet class gives you the framework on which you build any serious applet −

init − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.

start − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.

stop − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.

destroy − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.

paint − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

8a. Write a JAVA swing program to create a frame which contains 2 buttons named “Alpha” and “Beta”. When either of buttons is pressed, it should display “Alpha is pressed” and “Beta is pressed”. (10 Marks)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=450>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
JLabel jlab;
public void init() {
```

```
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
}
);
} catch (Exception exc) {
System.out.println("Can't create because of " + exc);
}
}
private void makeGUI() {
// Change to flow layout.
setLayout(new FlowLayout());
// Add buttons to content pane.
ImageIcon alpha = new ImageIcon("alpha.gif");
JButton jb = new JButton(alpha);
jb.setActionCommand("Alpha");
jb.addActionListener(this);
add(jb);
ImageIcon beta = new ImageIcon("beta.gif");
jb = new JButton(beta);
jb.setActionCommand("Beta");
jb.addActionListener(this);
add(jb);
// Create and add the label to content pane.
jlab = new JLabel("Choose a button");
add(jlab);
}
// Handle button events.
public void actionPerformed(ActionEvent ae) {
jlab.setText(ae.getActionCommand() + ”is pressed”);
}
}
```

8b. Write a JAVA program which demonstrates utilities of linked list class. (10 Marks)

The following program illustrates several of the methods supported by LinkedList:

```
import java.util.*;
public class LinkedListDemo {
  public static void main(String args[]) {
    // create a linked list
    LinkedList ll = new LinkedList();
    // add elements to the linked list
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");
    ll.add(1, "A2");
    System.out.println("Original contents of ll: " + ll);
    // remove elements from the linked list
    ll.remove("F");
```

```
    ll.remove(2);
    System.out.println("Contents of ll after deletion: "
     + ll);
         // remove first and last elements
    ll.removeFirst();
    ll.removeLast();
    System.out.println("ll after deleting first and last: "
     + ll);
    // get and set a value
    Object val = ll.get(2);
    ll.set(2, (String) val + " Changed");
    System.out.println("ll after change: " + ll);
  }
}
```