SCHEME OF EVALUATION

SUBJECT: OBJECT ORIENTED PROGRAMMING WITH JAVA

CODE: 20EI605

MONTH, YEAR: AUGUST,2023

Prepared By:

Dr.M.V.N.Chakravarthi

1	a)	Define Polymorphism Ans: Polymorphism is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.
	b)	What is scope of a variable? Ans: In programming, scope of variable defines how a specific variable is accessible within the program or across classes.
	c)	What is the use of import statement? Ans: keyword 'import' in Java programming is used to import the built-in and user-defined packages, class or interface in Java programming
	d)	List the types of constructors. Ans: There are two types of constructors in Java: No-arg constructor, and parameterized constructor.
	e)	Define Class and Object. Ans:Class is a blueprint which defines some properties and behaviors. An object is an instance of a class which has those properties and behaviours attached.
	f)	Write the syntax of initialising a String. Ans:*By string literal:* Java String literal is created by using double quotes. For Example: String s = “Welcome”; *By new keyword:* Java String is created by using a keyword “new”. For example: String s=new String(“Welcome”);
	g)	What is Method Overloading? Ans:Method overloading in java is a feature that allows a class to have more than one method with the same name, but with different parameters.
	h)	What is the importance CLASSPATH? Ans:The CLASSPATH variable is an environment variable, meaning it's part of the operating system (e.g., Windows). It contains the list of directories. These directories contain any class you created, plus the delivered Java class file, called the Java Archive (JAR).
	i)	List the rules for defining the Constructor. Ans: • The constructor's and class's name must be identical. • You cannot define an explicit value to a constructor. • A constructor cannot be any of these: static, synchronized, abstract, or final.
	j)	Write the syntax for importing the package? Ans:import packageName;
	k)	What is user defined exception? Ans:User Defined Exception or custom exception is creating your own exception class and throws that exception using 'throw' keyword. This can be done by extending the class Exception.
	l)	What is Daemon Thread? Ans:Daemon thread in Java is a low-priority thread that performs background operations such as garbage collection, finalizer, Action Listeners, Signal dispatches, etc.
	m)	List the phases in Thread Life Cycle. Ans:The six states of the thread life cycle in Java are: New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated.
	n)	Define Multithreading Ans: In Java, Multithreading refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU.
2	a)	List and Explain the principles of Java. Ans: The basic three principles of JAVA are inheritance, polymorphism and encapsulation. **Inheritance:** Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Example:

```
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}

class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

Output:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

b) List and Explain the operators in Java.

Ans:

Arithmetic Operators, Assignment Operators, Relational Operators, Logical Operators, Unary Operators, Bitwise Operators.

Arithmetic Operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

Bitwise Operators:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Boolean Logical Operators:

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

3 a) Illustrate different control statements available in Java.

Ans:

- Decision Making statements: if statements, switch statement
- Loop statements: do while loop, while loop, for loop. for-each loop

- Jump statements: break statement, continue statement

If:
The if statement was introduced in Chapter 2. It is examined in detail here. The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.
Here is the general form of the if statement:

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

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

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

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

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

Switch:
The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
...
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

Nested switch Statements
You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch.
For example, the following fragment is perfectly valid:

```
switch(count) {
case 1:
switch(target) { // nested switch
case 0:
System.out.println("target is zero");
```

		break; case 1: // no conflicts with outer switch System.out.println("target is one"); break; } break; case 2: // ...
	b)	Distinguish the types of type conversion in Java with examples. *Ans: Type casting:* Convert a value from one data type to another data type is known as type casting. *Types of Type Casting:* Widening Type Casting, Narrowing Type Casting *Widening Type Casting*: Widening casting is done automatically when passing a smaller size type to a larger size type. Ex: public class Main { public static void main(String[] args) { int myInt = 9; double myDouble = myInt; // Automatic casting: int to double System.out.println(myInt); // Outputs 9 System.out.println(myDouble); // Outputs 9.0 } } *Narrowing Type Casting*: Narrowing casting must be done manually by placing the type in parentheses in front of the value. Ex: public class Main { public static void main(String[] args) { double myDouble = 9.78d; int myInt = (int) myDouble; // Manual casting: double to int System.out.println(myDouble); // Outputs 9.78 System.out.println(myInt); // Outputs 9 } }
4	a)	Demonstrate the usage of static keyword in Java. Ans: There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable. Methods declared as static have several restrictions: • They can only call other static methods. • They must only access static data. • They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.) If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables, and a static initialization block. Example: class UseStatic { static int a = 3; static int b; static void meth(int x) { System.out.println("x = " + x);

		System.out.println("a = " + a); System.out.println("b = " + b); } static { System.out.println("Static block initialized."); b = a * 4; } public static void main(String args[]) { meth(42); } }
	b)	Explain the use of this keyword in Java with an example program. Ans: Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class' type is permitted. To better understand what this refers to, consider the following version of Box( ): // A redundant use of this. Box(double w, double h, double d) { this.width = w; this.height = h; this.depth = d; } This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object.
5	a)	What is inner class? Explain with an example. Ans: The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block. There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used. The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner. Example: class Outer { int outer_x = 100; void test() { Inner inner = new Inner(); inner.display(); } // this is an inner class class Inner { void display() { System.out.println("display: outer_x = " + outer_x); } } } class InnerClassDemo { public static void main(String args[]) { Outer outer = new Outer(); outer.test(); } }
	b)	Write a program to demonstrate constructor overloading. Ans: In addition to overloading normal methods, you can also overload constructor methods. In fact, for most

real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the Box class developed in the preceding chapter. Following is the latest version of Box.

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

As you can see, the Box( ) constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box( ) constructor. For example, the following statement is currently invalid.

```
Box ob = new Box();
```

6 a) Define Inheritance. Explain about types of inheritance.

Ans:

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. To see how, let's begin with a short example. The following program creates a superclass called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.

```
// A simple example of inheritance.
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
```

		subOb.showk(); System.out.println(); System.out.println("Sum of i, j and k in subOb:"); subOb.sum(); } }
	b)	What is dynamic method dispatch? Explain with an example. Ans: When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch: // Dynamic Method Dispatch class A { void callme() { System.out.println("Inside A's callme method"); } } class B extends A { // override callme() void callme() { System.out.println("Inside B's callme method"); } } class C extends A { // override callme() void callme() { System.out.println("Inside C's callme method"); } } class Dispatch { public static void main(String args[]) { A a = new A(); // object of type A B b = new B(); // object of type B C c = new C(); // object of type C A r; // obtain a reference of type A r.callme(); // calls A's version of callme r = b; // r refers to a B object r.callme(); // calls B's version of callme r = c; // r refers to a C object r.callme(); // calls C's version of callme } } This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme( ) method.
7	a)	Define a Package. Explain how a package is imported. Ans: Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only

		exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world. To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the package statement: *package pkg;* Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage. *package MyPackage;* Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here: *package pkg1[.pkg2[.pkg3]];* A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as *package java.awt.image;* needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored. **Importing Packages:** Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing. In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement: *import pkg1[.pkg2].(classname|*);* Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
	b)	What is an interface? Example how interface is implemented. Ans: Using interface, you can specify a set of methods that can be implemented by one or more classes. The interface, itself, does not actually define any implementation. Although they are similar to abstract classes, interfaces have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise). Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. An interface is defined much like a class. This is the general form of an interface: access interface name { return-type method-name1(parameter-list);

		return-type method-name2(parameter-list); type final-varname1 = value; type final-varname2 = value; // ... return-type method-nameN(parameter-list); type final-varnameN = value; }
8	a)	Write a program to demonstrate the use of try, catch and finally. Ans: Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. 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 } Here, ExceptionType is the type of exception that has occurred.
	b)	Differentiate throw and throws with examples. Ans: **Throw:** So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here: *throw ThrowableInstance;* Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace. **Throws:** If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. Athrows clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause: type method-name(parameter-list) throws exception-list { // body of method } Here, exception-list is a comma-separated list of the exceptions that a method can throw.

9	a)	Write a program to demonstrate creation of multiple threads using Thread class. Ans: **Creating Multiple Threads:** So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads: // Create multiple threads. class NewThread implements Runnable { String name; // name of thread Thread t; NewThread(String threadname) { name = threadname; t = new Thread(this, name); System.out.println("New thread: " + t); t.start(); // Start the thread } // This is the entry point for thread. public void run() { try { for(int i = 5; i > 0; i--) { System.out.println(name + ": " + i); Thread.sleep(1000); } } catch (InterruptedException e) { System.out.println(name + "Interrupted"); } System.out.println(name + " exiting."); } } class MultiThreadDemo { public static void main(String args[]) { new NewThread("One"); // start threads new NewThread("Two"); new NewThread("Three"); try { // wait for other threads to end Thread.sleep(10000); } catch (InterruptedException e) { System.out.println("Main thread Interrupted"); } System.out.println("Main thread exiting."); } }
	b)	What is Deadlock. Explain about Thread Synchronisation. **Deadlock:** A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons: • In general, it occurs only rarely, when the two threads time-slice in just the right way. • It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.) **Synchronization:** 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. As you will see, Java provides unique, language-level support for it. 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 amonitor 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.