**Question**

1. With the aid of a schema distinguish between a modular and an object oriented programming paradigm.

**MODULAR PROGRAMMING**

Modular Programming: Many programs can be decomposed into a series of identifiable subtasks. It is a good programming practise to implement each of these subtasks as a separate program module. The idea of modular programming is to sub-divide a program into smaller units that are independently testable and that can be integrated to accomplish the overall programming objective (Abott, 1993). The use of modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations (Louden, 1993). One motivation for modularizing a program into methods is the divide-and-conquer approach, which makes program development more manageable by constructing programs from small, simple pieces. Another is software reusability - using existing methods as building blocks to create new programs.





**OBJECT ORIENTED PROGRAMMING PARADIGM.**

Object-oriented programming is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields, and code, in the form of procedures. A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated



**Encapsulation**

Encapsulation is the mechanism of hiding of data implementation by restricting access to public methods. Instance variables are kept private and accessor methods are made public to achieve this.

For example, we are hiding the name and dob attributes of person class in the below code snippet.

Encapsulation — private instance variable and public accessor methods.

**public** **class** **Employee** {
 **private** **String** name;
 **private** **Date** dob; **public** **String** getName() {
 **return** name;
 } **public** **void** setName(**String** name) {
 this.name = name;
 } **public** **Date** getDob() {
 **return** dob;
 } **public** **void** setDob(**Date** dob) {
 this.dob = dob;
 }
}

**Abstraction**

Abstract means a concept or an Idea which is not associated with any particular instance. Using abstract class/Interface we express the intent of the class rather than the actual implementation. In a way, one class should not know the inner details of another in order to use it, just knowing the interfaces should be good enough.

**Inheritance**

Inheritances expresses “is-a” and/or “has-a” relationship between two objects. Using Inheritance, In derived classes we can reuse the code of existing super classes. In Java, concept of “is-a” is based on class inheritance (using extends) or interface implementation (using implements).

For example, FileInputStream "is-a" InputStream that reads from a file.

**Polymorphism**

It means one name many forms. It is further of two types — static and dynamic. Static polymorphism is achieved using method overloading and dynamic polymorphism using method overriding. It is closely related to inheritance. We can write a code that works on the superclass, and it will work with any subclass type as well.

**Example**

Java collections framework has an interface called java.util.Collection, ArrayList and TreeSet are two different implementation of this interface. ArrayList maintains the insertion order of elements while TreeSet orders its elements by their natural order or comparator(if supplied). Now if we write a method that accepts a collection and prints its elements, the actual object (ArrayList or TreeSet) at runtime will decide the behavior of this method.

Polymorphic print method

**public** **void** print(**Collection**<**String**> collection) {
 **for** (**String** s : collection) {
 **System**.out.println("s = " + s);
 }
}

Passing an ArrayList

**Collection**<**String**> collection1 = **new** **ArrayList**<>();
collection1.add("A");
collection1.add("D");
collection1.add("B");
collection1.add("C");
print(collection1); **(1)**

1. elements will be printed as per the insertion order of elements into arraylist

Program output

s = A
s = D
s = B
s = C

Passing an TreeSet

**Collection**<**String**> collection2 = **new** **TreeSet**<>();
collection2.add("A");
collection2.add("D");
collection2.add("B");
collection2.add("C");
print(collection2); **(1)**

1. elements will be printed as per the natural order

Program output

s = A
s = B
s = C
s = D

We just saw that print() method’s behavior is determined by the actual type of object passed to it at run time. That’s polymorphism!

Important Facts

1. Other than objects of type java.lang.Object, all java objects are polymorphic i.e. they pass the IS-A test for their own type as well as for class Object.
2. A reference variable’s type determines the methods that can be invoked on the object that variable is referencing to. In the example above, print() method can only invoke methods that are listed on Collection interface irrespective the type of actual object passed to this method.
3. Polymorphic method invocation applies only to the instance methods (not to static methods, not to variables). Only overriden instance methods are dynamically invoked based on the real object’s type at runtime.



1. Use this schema to illustrate the modifications  required to translate an airline reservation program from a modular to an object oriented design:



## SAMPLE QUESTION:

Write a reservation system for an airline flight. Assume the airplane has 10 rows with 4 seats in each row. Use a two-dimensional array of strings to maintain a seating chart. In addition, create a array to be used as a waiting list in case the plane is full. The waiting list should be "first come first serve" that is people who are added early in the list get the priority over those who are added later. Allow the user to add three options

1. Add a passenger to the list.

a. Request the passenger's name.

b. Display a chart of the seats in the airplane in a tabular form

c. If seats are available let the passenger choose a seat. Add the passenger to the seating chart.

d. If no seats are available, place the passenger on the waiting list.

2. Remove a passenger from the list

a. Request the passengers name

b. Search the seating chart for the passenger's name and delete it.

c. If the waiting list is empty, update the array so that the seat is available,

d. If the waiting list is not empty remove the first person from the list and give him or her the newly vacated seat.

3. Quit You will need to use string methods defined in java.lang.String in the documentation reference. String comparision function names are as follows equal, equalIgnoreCase.

**SSOME STEPS TO SOLVE THE ABOVE QUESTION FOR EXAMPLE USING OBEJCETD ORIENTED PROGRAMMING**

You will need to apply most concepts including

1. OOP theory

2. File IO

3. Language constructs such as (Looping/Decision Making/Methods etc.)

4. Swing (Optional)

5. Interfaces

6. Exception Handing

How to approach this lab?

1. Read the lab requirement very carefully. The requirements are provided in the manner a Marketing Department in an organization would provide.

2. Identify all the entities (object types) that are involved in this lab.

3. Identify the relationship between each entity. You should identify what properties in a given object would be changed by what methods in a given class.

4. After this step you are ready to do your design. On a piece of paper draw a large box for each class and document what properties and methods would go in each class.

5. Identify opportunities for reusability.

- Look at duplication for properties and methods.

- Does the relationship make sense?

- Identify opportunities for using

- Inheritance/polymorphism.

- For e.g. can there be one add/remove method for passenger chart and waiting list.

- Or can these two methods be overridden?

- Do you see any opportunities for using abstract classes? These are just some questions to ask. You may consider asking additional question depending on what concepts you may end up applying. For e.g.

- Can we serialize some objects to a file to remember program state? etc.

Once your design is done, you are ready for coding. You just write your classes separately and unit test them. Unit testing means that you take each class and write a main that has a driver program that exercises the methods giving specific inputs (depending on datatype) and looking for expected outputs. Once you write the main for your application, you will automatically end up sequencing the method calls that will dictate the application usage. This setup is called a use case. It may not be a bad idea to think of your use case ahead of time. You can use the top-down design methodology or bottom-up. It your choice. I find the top-down to be more friendly than bottom-up in software engineering. Top-down methodology provides me with a big picture and allows me to drill down to specific components.

## NOTE: Object Oriented Programming

object-oriented programming (OOP) Object-oriented programming (OOP) is aprogramming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

## See sample codes below:

#### Coding:

1. /\*\*
2. \* @author
3. \*
4. \*/
5. public interface Reservation {
6. public abstract void addPax(String paxName, int row, int seatNo);
7. public abstract boolean...