TEMASEK JUNIOR COLLEGE

Temasek Junior College

H2 Computing

Object Oriented Programming

1 Evolution of programming languages

	Evolution of up to high level programming languages	Evolution of structured programming languages
Increase productivity		
Simply express the evolution of commands	Using 'high level' commands to increase expressiveness	
Increase maintainability		
Easy to understand the evolution of the program		Three control structures
Increase quality		
Reduce the evolution of complexity by adding restrictions		No GOTO programming
Promote reusability		
Remove duplication of logic and promote the evolution of componentization and reusability	Subroutines	Strengthen the independence of subroutines

1.1 Introduction to programming styles

1.1.1 Procedural and Structured programming

1.1.1.1 In procedural programming the programmer concentrates on the steps that must be undertaken to solve the problem rather than the data and the operations which apply to that data.

1.1.1.2 Structured programming is a type of procedural programming. In structured programming a programmer uses a limited set of control structures in the construction of the program. Examples include *while*, *if* and *for*. The advantage of a structured approach over plain procedural programming is that it supports a top-down approach to design.

1.1.1.2.1 Hierarchy or **structure charts** use a top-down approach to explain how a program is put together.

Starting from the program name, the programmer breaks the problem down into a series of steps.
Each step is then broken down into finer steps so that each successive layer of steps shows the subroutines that make up the program in more detail.

The overall hierarchy of **a** program might look like this:

- programs are made up of modules
- modules are made up of subroutines and functions
- subroutines and functions are made up of algorithms
- algorithms are made up of lines of code
- lines of code are made up of statements (instructions) and data.

In larger programs a **module** is a self-contained part of the program that can be worked on in isolation from all the other modules. This enables different programmers to work independently on different parts of large programs before they are put together at the end to create the program as a whole.

Example 1 Student Management System in Procedural and Structured programming

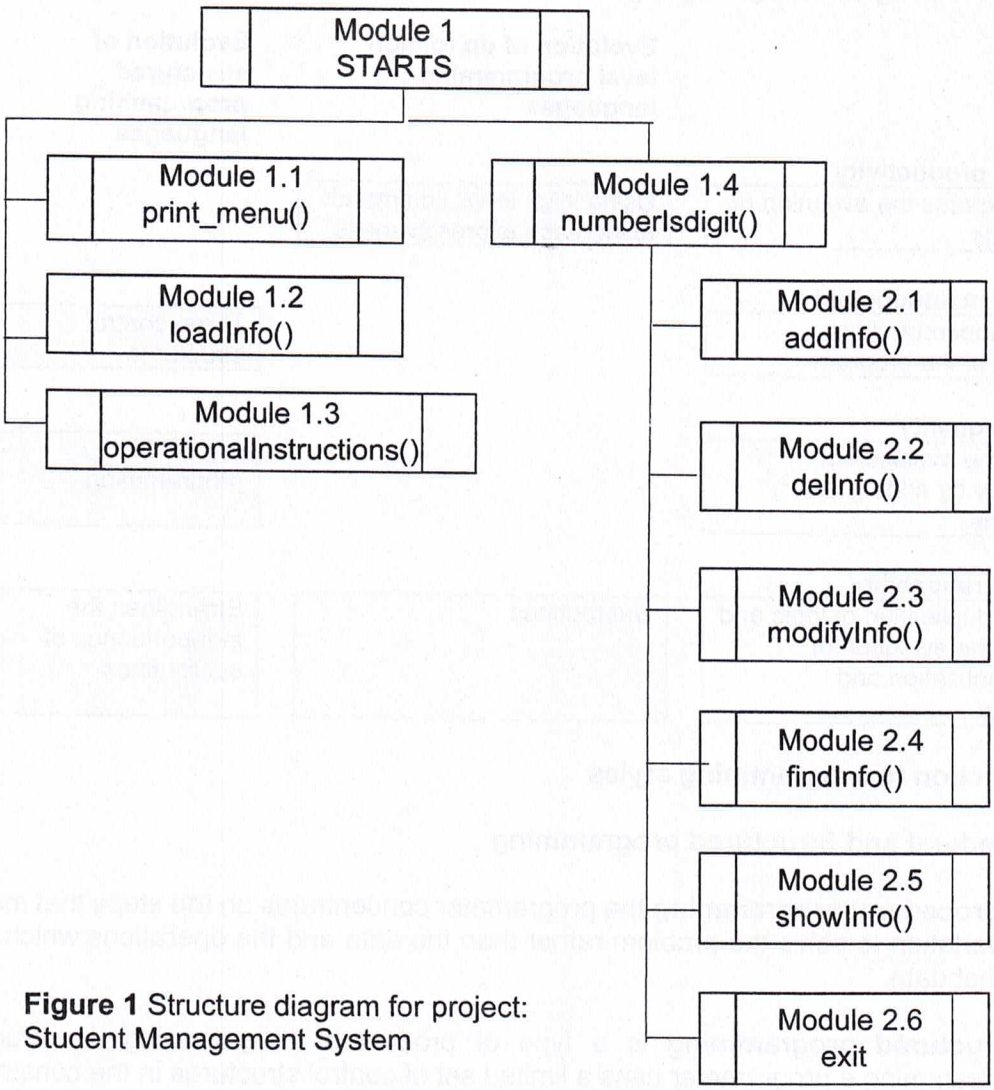


Figure 1 Structure diagram for project: Student Management System

Example 2 Python codes - Student Management System in Procedural and Structured programming

----- Python Project **-StuInfoSys.py**

```
""" Use python to create a student management system"""
""" Python functional programming realization ideas """
""" Use the dictionary to store the student's name, telephone numbers,
and email """
""" Use list to install student information """
""" Achieve adding, deleting, modifying, and checking dictionary data
in the list """
""" Version 2021. Procedural and Structured programming v1.1"""
""" Record:
{'name': 'Tan Soon Nee', 'phone': '90912234', 'email': 'tansn@hotmail.com'}
```

This is a top-down approach that breaks programs into modules, which in turn get broken down into subroutines and functions. Object-oriented programming can be thought of as an *extension* to this structured approach as it is entirely modular.

1.1.1.2.2 Problems in procedural and structured languages

Procedural and Structured programming can resolve the following

No GOTO programming	Accomplishing reusability by using subroutines

Problems in procedural and structured languages

Global variables	Poor reusability

1.1.1.2.3 Difference between procedural and object-oriented programming

The **key difference** is that in procedural programming, the lines of code and the data the code operates on are *stored separately*. An *object-oriented program* puts all the data and the processes that can be carried out on that data together in one place called an object and allows restrictions to be placed on how the data can be manipulated by the code.

Example 3 Global variables used in **Example 2**

```
students = []
info = "1:Added|2:Delete|3:Modified|4:Search|5:Display|6:Exit the system"
```

2 Object Oriented Programming / OOP

General Introduction

Though many computer scientists and programmers consider OOP to be a modern programming paradigm, the roots go back to 1960s. The first programming language to use objects was Simula 67. As the name implies, Simula 67 was introduced in the year 1967. A major breakthrough for object-oriented programming came with the programming language Smalltalk in the 1970s.

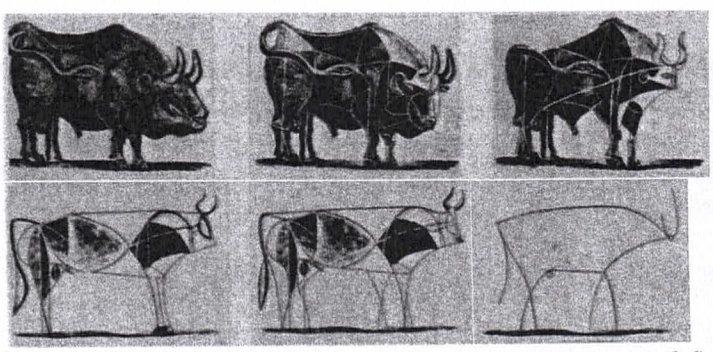
Four major principles of object-orientation deals with on object-oriented programming:

- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Question: State the four major principles in OOP.

2.1 Abstraction

- The concept of abstraction is to reduce problems to their essential features.



- Another way of explaining abstraction is that it is the process of finding similarities or common aspects about the problem, while ignoring differences.

Example 4: JC_Students

Similarities / common aspects:

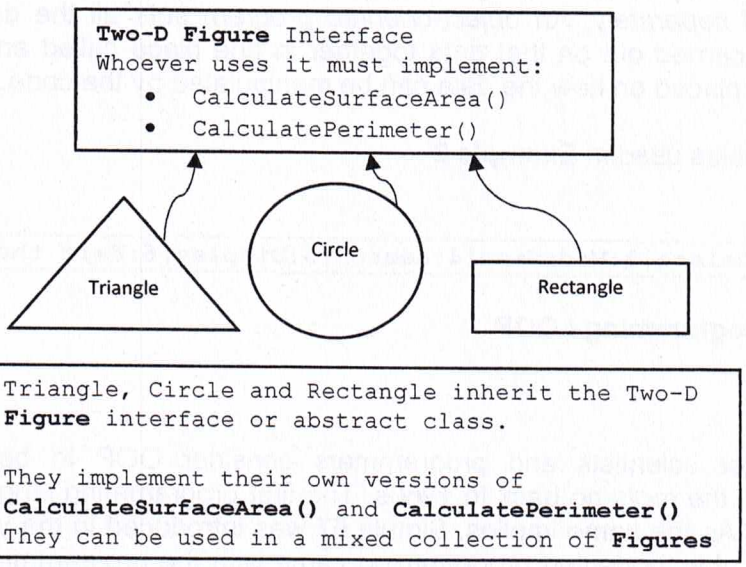
- L1R5 –
- Age range –
- ...

Example 5: Circle

Similarities / common aspects:

- Radius
- Area
- Circumference

Example 6: Regular 2-D shapes



- This is a useful concept for programmers as they can view the problem from a high level, concentrating on the key aspects of designing a solution whilst ignoring the detail, particularly during the initial design stages.
- Once a solution has been identified for the current problem, a feature of abstraction is that the abstraction from one problem can be applied to another similar problem, which shares the same common features.
- The concept of exposing only essential data and other details to the users of a class, whilst hiding unnecessary information that could lead to either confusion or increase the probability of making mistakes.

Question: What is data abstraction?

2.1.1 OOP

A **programming style** in which solutions take the form of a set of interrelated objects. Programmers identify objects (data and the associated operations on that data) in the domain their application is trying to model. Objects in this sense are collections of related data which represent the state of the object and methods which manipulate the data. They then consider the interactions between objects required to fulfil the requirements of the task.

Question: What is OOP?

2.1.2 There are three main ways in which object-oriented programming has been found to *improve* programmer productivity:

- By providing an environment which assists programmers to *improve the quality* of their code, e.g., **no need** to use global variables.
- Making it easier for programmers to *reuse* their own code.
- Providing simple mechanisms to make use of *existing code libraries.*

2.1.2.1 The three ways are:

- Class
- Inheritance and
- Polymorphism

Question: State the three main ways to *improve* programmer productivity in object-oriented programming.

2.1.2.2 Benefits

- Remove program redundancy
- Structure for finishing

Question: What are the benefits of OOP?

2.2 Implementation abstract data types with object-oriented languages

Modern object-oriented languages, such as Python and C++, support a form of abstract data types (ADT). When a **class** is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is usually an object of that class. The module's interface typically declares the constructors as ordinary procedures, and most of the other ADT operations as methods of that class.

2.2.1 Abstract data types/ADTs

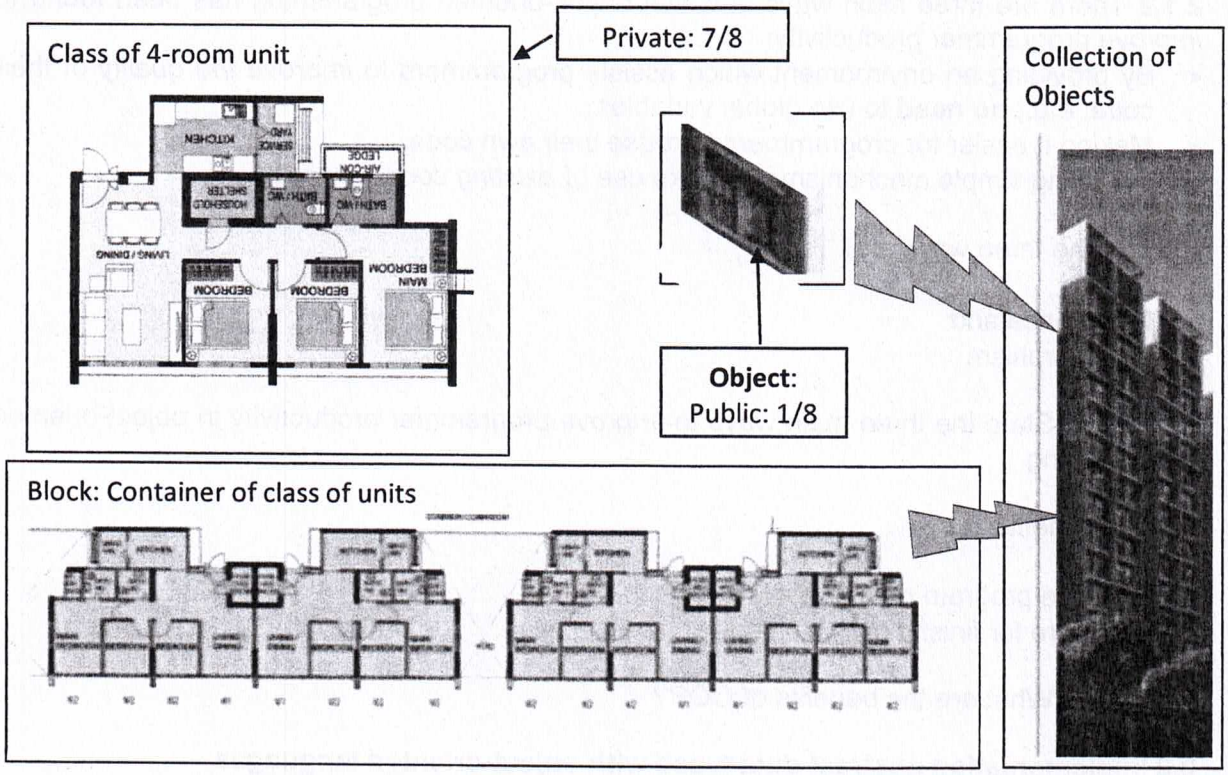
An **abstract data type** (ADT) is a mathematical model for data types where a data type is defined by its behaviour (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behaviour of these operations. An ADT is typically implemented as a class, and each instance of the ADT is usually an object of that class.

Question: What is ADT?

2.4 Relationship between the concept of **type** in a **typed programming language** and the concept of **class** in an **object-oriented programming language**

- Declaring that an object belongs to a class is analogous to typing a variable. Typing effectively says what operations are valid for a variable. If an object is identified with a class, then the class defines the operations which are valid for it. However, in OOP because of inheritance an object may also inherit valid operations from a superclass. Therefore, an object may conform to more than one class but have only a single type.

3 Class



3.1 Class diagram

Class diagram of "`Four_Room_Flat`".

Class diagram:

Class names →

Private attributes /Data members

Public methods /Member functions

Four_Room_Flat

Private:
`Master_Bed_Room`
`Bed_Room_1`
`Bed_Room_2`
`Living_Room`
`Kitchen`
`Toilet_1`
`Toilet_2`
:

Public:
`Area()`
`Layout()`
`Price()`
:

Notes:

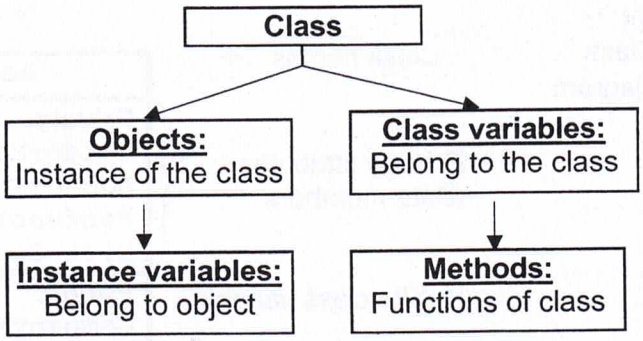
- In design stage, a **class** is a template for an object. It defines all the data items contained in the object and the methods that operate on that data
- At runtime a **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behaviour (member functions/ methods)

3.2 Attributes and methods

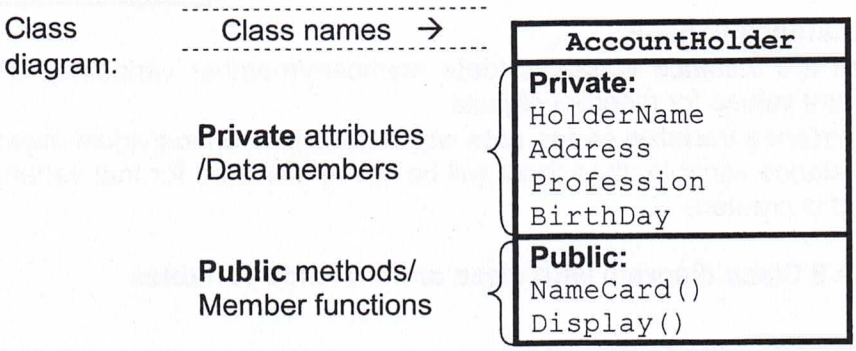
A class is the definition of all the **attributes** and **methods** which are the common aspects of all objects created from it.

3.2.1 Attributes in classes

- The **attributes** are the data items in a class.



Example 7 A "real-life" class diagram "`AccountHolder`".



- The attributes (data members) of the `AccountHolder` class consists for example of the `HolderName`, `Address`, `Profession`, and `BirthDay`.
  Methods (member functions) are `NameCard()` and `Display()`.
  This model is not complete, because need more data and above all more methods like e.g., setting and getting the birthday of an account holder.

Task_1 Creation of class `AccountHolder` **in Python**

3.2.1.1 Class variables

- There are *class variables*, which have the same value in all methods
- A class variable holds information which may be accessed by every object belonging to the class.
- There will only be one copy of a class variable in existence regardless of the number of objects created for that class
- e.g. `CompanyName` should consider as a class variable of class **`AccountHolder`**

Example 8

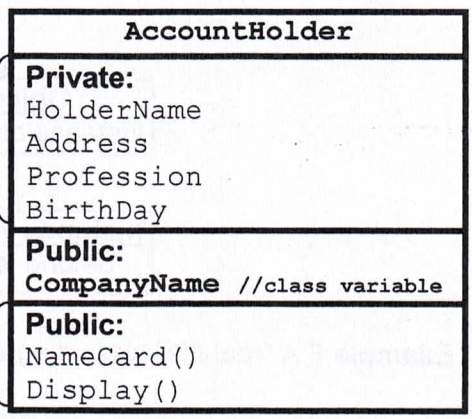
Class diagram:

Class names →

Private attributes /Data members

Public class variable

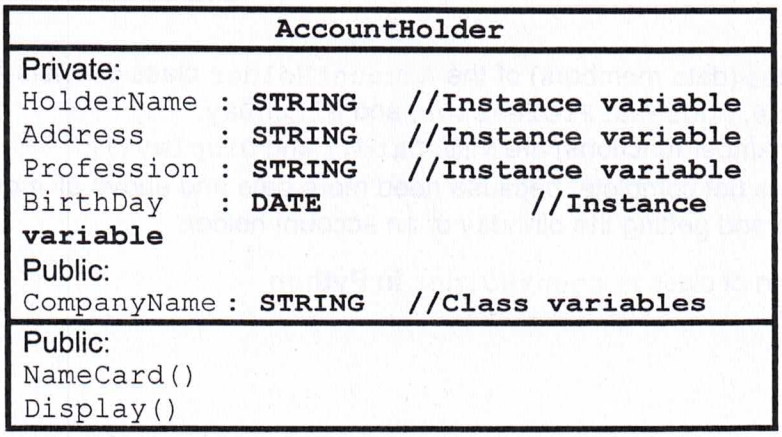
Public methods/ Member functions



3.2.1.2 Instance variable

- There are *instance variables* (data members/member variables), which have normally different values for different objects.
- An **instance variable** stores **data** which describes an individual object. If a class defines an instance variable, then there will be space allocated for that variable every time a new object is created.

Example 9 Class diagram with class and instance variables



3.2.1.3 Class member visibility

- Within a class instance variables and methods may be **public**, **private** or **protected**.
- **Public** variables and methods are visible in code external to the class.
- **Private** variables and methods are only visible in code within the class.
- **Protected** variables and methods are visible within the class in which they are declared and all subclasses of that class.

3.2.1.4 Class diagrams

Version 1

Class diagram with **data types**:

Class names →

Private attributes /Data members

Public methods/ Member functions

AccountHolder
Private: HolderName : STRING Address : STRING Profession : STRING BirthDay : DATE **Public:** CompanyName : **STRING**
Public: NameCard() Display()

Version 2

UML (Unified Modelling Language)

In UML, member access specifiers stipulate the visibility level of class members (attributes and behaviours) to promote an optimum level of encapsulation and information hiding. Symbols are **–** (**private**), **+** (**public**) and **#** (**protected**), which prefix each member in a class diagram to specify its access level.

UML Class diagram:

Class names →

Private attributes /Data members

Public attribute and methods/Member functions

AccountHolder
– HolderName : STRING – Address : STRING – Profession : STRING – BirthDay : DATE + CompanyName : STRING
+ NameCard() + Display()

Example 10 Draw 2 versions of class diagram for class "**Account** " with `accountNumber` and `balance` as attributes.

Version 1 **Version 2**

3.2.2 Methods used in classes

- Essentially, a **method** is a function, but it's a special kind of function which belongs to a class, i.e. it is defined within a class, and works on the instance and class data of this class. It implements the behaviour of a class as a code procedure and forms the interface by which an object communicates with the outside.
- A **method** is a programmed procedure (member functions) that is defined as part of a class.
- **Method signature:** A method is commonly identified by its unique **method signature**. This usually includes the **method name**, the **number** and **type of its parameters**, and its **return type**.

3.2.2.1 Instance method

- An object also has a number of member functions which specify the nature of the operations on the object.
- It is a **method** that is used to manipulate instance variables in an object
- Methods can only be called through instances of a class or a subclass, i.e. the instances name followed by a **dot(•)** and the method name.

Example 12

AccountHolder		
Private:		
HolderName	: STRING	//Instance variable
Address	: STRING	//Instance variable
Profession	: STRING	//Instance variable
BirthDay	: DATE	//Instance variable
Public:		
JobDescription	: STRING	//Class variables
CompanyName	: STRING	//Class variables
Public:		
NameCard()		**//Instance method**
Display()		**//Instance method**

3.2.2.1.1 Accessor methods are used to **retrieve** data member **values** from within an object, commonly known as getters. Getter methods do not change the values of attributes; they just return the values. **(Instance methods)**

Example 13

18/100 marks

AccountHolder		
Private:		
HolderName	: STRING	//Instance variable
Address	: STRING	//Instance variable
Profession	: STRING	//Instance variable
BirthDay	: DATE	//Instance variable
Public:		
JobDescription	:STRING	//Class variables
CompanyName	: STRING	//Class variables
Public:		
GetHolderName()	**: STRING**	**//Accessor [Instance method]**
GetAddress()	**: STRING**	**//Accessor [Instance method]**
GetProfession()	**: STRING**	**//Accessor [Instance method]**
GetBirthDay()	**: STRING**	**//Accessor [Instance method]**
NameCard()		//Instance method
Display()		//Instance method

3.2.2.1.2 Mutator methods are used to **change** the **value** of data member and are commonly known as setters. **(Instance methods).**

All private attributes must EACH have a GETTER and SETTER.

Example 14

AccountHolder		
Private:		
HolderName	: STRING	//Instance variable
Address	: STRING	//Instance variable
Profession	: STRING	//Instance variable
BirthDay	: DATE	//Instance variable
Public:		
JobDescription	: STRING	//Class variables
Public:		
GetHolderName()	: STRING	//Accessor [Instance method]
GetAddress()	: STRING	//Accessor [Instance method]
GetProfession()	: STRING	//Accessor [Instance method]
GetBirthDay()	: STRING	//Accessor [Instance method]
SetHolderName(newName : STRING)		**: BOOL //Mutator [Instance method]**
changeOfResidence (newAddress:STRING)		**:BOOL //Mutator [Instance method]**
changeOfProfession (newProfessional: STRING)		**:BOOL //Mutator [Instance method]**
SetBirthDay(newBDay : STRING)		**:BOOL //Mutator [Instance method]**
NameCard()		//Instance method
Display()		//Instance method

validation is needed.

Task_2 Adding methods to class `AccountHolder` **in Python**

3.2.2.1.3 Accessor vs Mutator

Mutator methods therefore accept arguments (reflecting the data member that they are destined to update), whereas **accessor** methods usually possess a return type reflecting the data member that they are to convey back to the caller. Both are typically designated as public to facilitate their invocation through an object.

Note: Do not write Python program in answering OOP questions in Theory paper.

3.2.2.2 Class method

- Class method, which can be executed without the creation of any instances, and may only manipulate class variables, not instance variables.
- Methods can only be called through instances of a class or a subclass, i.e. the class name followed by a **dot(•)** and the method name.

Example 15

AccountHolder
Private: HolderName : STRING //Instance variable Address : STRING //Instance variable Profession : STRING //Instance variable BirthDay : DATE //Instance variable **Public:** JobDescription: STRING **//Class variables** CompanyName: STRING //Class variables
Public: NameCard() Display() **ShowJobDescription() //Class method**

AccountHolder
– HolderName : STRING //Instance variable – Address : STRING //Instance variable – Profession : STRING //Instance variable – BirthDay : DATE //Instance variable + JobDescription: STRING **//Class variables** + CompanyName: STRING //Class variables
+ NameCard() + Display() + **ShowJobDescription() //Class method**

Task_3 Adding class methods to class `AccountHolder` **in Python**

3.2.2.3 Instance methods vs. Class methods

- **Instance methods** operate upon instances of a class (i.e., objects), and are typically concerned with the manipulation of instance variables – i.e., those data members that may differ from object to object.
- **Class methods** are typically concerned with the manipulation of class variables, and do not require the existence of any instance to operate.

3.2.2.4 Constructor and Destructor → 10marks

3.2.2.4.1 Instantiation is the **creation of a real instance** for particular realization of an abstraction or template such as a class of objects or a computer process. To instantiate is to create such an instance by, for example, defining one particular variation of object within a class, giving it a name, and locating it in some physical place.

Create an object → looks for class definition → activate

3.2.2.4.2 Constructor is the name given to the **procedure** by which objects are created from a class definition. Constructor functions undertake such tasks as initialising data members, making dynamic memory allocation requests, opening files or communication sockets.

- A **constructor** in a class is a special type of method called to **initialise** an object. It prepares the new object for use, often accepting arguments that the constructor uses to set the value of member variables
- first method of a class

Python: `__init__()`

Class: Constructor (HolderName, Address, Profession)

3.2.2.4.3 Destructor is a **method** which is invoked when the object is going to destroy. It can happen when its lifetime is bound to scope and the execution leaves the scope, when it is embedded into another object whose lifetime ends, or when it was allocated dynamically and is released explicitly. Its main purpose is to free the resources (memory allocations, open files or sockets, database connections, resource locks, etc.) which were acquired by the object along its life cycle and/or deregister from other entities which may keep references to it.

Example 16 Class diagram with constructor and destructor

AccountHolder
– HolderName : **STRING** – Address : **STRING** – Profession : **STRING** – BirthDay : **DATE**
– **Constructor(holderName , address , profession, birthday)** – **Destructor()** + NameCard() + Display()

or

AccountHolder
Private: HolderName : **STRING** Address : **STRING** Profession : **STRING** BirthDay : **DATE**
Private: **Constructor(holderName , address , profession, birthday)** **Destructor()** Public: NameCard() Display()

Task_4 Constructor and destructor methods of class `AccountHolder` **in Python**

3.2.2.4.4 Constructor vs. Destructor

Constructor methods are called when an object is created, and destructor method are called when an object goes out of scope. Destructors may not exist in some languages with garbage collection (e.g. Java). Constructors undertake initialisation duties such as setting data to default values, preparing files, sockets, or GUI components. **Destructors** are used to end the existence of an object gracefully, closing open files, sockets, or GUI components, and, in some languages, are used to de-allocate dynamically allocated memory to prevent memory leakage.

3.2.2.5 A **virtual method** is a concrete super-class method that can be replaced in a sub-class with a new implementation (viz., overriding).

3.2.2.6 A **pure virtual method**, by contrast, is a method signature in a super-class that has no corresponding implementation within that class (i.e., it is abstract), and therefore must be implemented in any sub-classes created that the programmer wishes to become concrete

Example 17 Class diagram for **Box**

Box
Private: Length : INTEGER Width : INTEGER Height : INTEGER
Public: Constructor(Length, Width, Height) Destructor() getLength() : INTEGER getWidth() : INTEGER getHeight() : INTEGER setLength(newLength) : BOOL setWidth(newWidth) : BOOL setHeight (newHeight) : BOOL

Constructor and Destructor are also private

Example 18 Class diagram for **Carpet** - Using Private Functions and Public Data
When creating a class, usually want to make data items private (to control how they are used) and to make functions public (to provide a way to access and manipulate the data). However, if there is a reason to do so, then free to make particular data items public, and particular functions private.

Carpet
Private: Length : INTEGER Width : INTEGER Price : REAL setPrice() : BOOL
Public: Constructor(Length, Width) Destructor() getLength() : INTEGER getWidth() : INTEGER getPrice() : REAL setLength(newLength) : BOOL setWidth(newWidth) : BOOL

or

Carpet
– Length : INTEGER – Width : INTEGER – Price : REAL – setPrice() : BOOL
– Constructor(Length, Width) – Destructor() + getLength() : INTEGER + getWidth() : INTEGER + getPrice() : REAL + setLength(newLength) : BOOL + setWidth(newWidth) : BOOL

Pseudocode for member function:

```
FUNCTION setLength(newLength)
      Length = newLength
      setPrice()
ENDFUNCTION
FUNCTION setWidth(newWidth)
      Width = newWidth
      setPrice()
ENDFUNCTION
```

Pseudocode for member function:

```
FUNCTION setPrice()
      CONST int SMALL = 12
      CONST int MED = 24
      CONST double PRICE1 = 29.99
      CONST double PRICE2 = 59.99
      CONST double PRICE3 = 89.99
      Area = Length * Width
      IF (Area <= SMALL) THEN
            Price = PRICE1
      ELSE
            IF (Area <= MED) THEN
                  Price = PRICE2
            ELSE
                  Price = PRICE3
            ENDIF
      ENDIF
ENDFUNCTION
```

4 Objects

- An **object** is a particular instance of a class which is a collection of data and operations that act on those data.
    - The data (variables, data structures) describes the state of the object.
    - It has memory allocated to it to record its state.
    - The operations (functions) are the methods defined in the class and those inherited from superclasses.
    - Methods describes the behaviour of the object.
    - The process of creating this object is known as instantiation.
    - An object has values for data.
    - It responds to messages passed to it.

4.1 Object diagrams

Example 19 Object for class `AccountHolder`

```
curHolder = AccountHolder('Alien Lee', '22 Bedok', 'Accountant', '04/04/2001')
```

curHolder : AccountHolder
Private: HolderName = 'Alien Lee' Address = '22 Bedok' Profession = 'Accountant' BirthDay = '04/04/2001'

Example 20 Object for class `Box`

matchBox : Box

Private:
Length = 5 //cm
Width = 3
Height = 1

Example 21 Object for class `Carpet`

LRCarpet : Carpet

Private:
Length = 3 //meter
Width = 2.5
Price = 29.99 //$

Question: Outline the features of an object.
Answer: An object is a combination of data and the operations that can be performed in association with that data. Each data part of an object is referred to as a data member while the operations can be referred to as methods. The current state of an object is stored in its data members and that state should only be changed or accessed through the methods. Common categories of operations include: the construction of objects; operations that either set (mutator methods) or return (accessor methods) the data members; operations unique to the data type; and operations used internally by the object.

5 Message

A **message** is the mechanism by which object communicate. Specifically, a message refers to an object by name, invokes a method, and provides input data in the form of method arguments, if required.

Example 22

```
class Player(object):
    def blast(self, enemy):
        print 'player'
        enemy.die()

class Alien(object):
    def die(self):
        print 'Alien'

hero = Player()
invader = Alien()
hero.blast(invader)
```

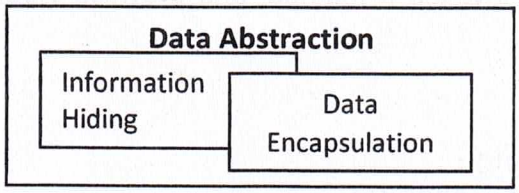
5.1 Differentiate between method and message

A method is a class behaviour – typically, a member function that performs an activity enabling objects of a class to be manipulated. Messages are communications between objects – for instance, where one object may call a public member function of another, and supply data as an argument to that function.

OOP Tutorial 1

6 Data Abstraction, Data Encapsulation, and Information Hiding

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things, so that the name captures the core of what a function or a whole program does.

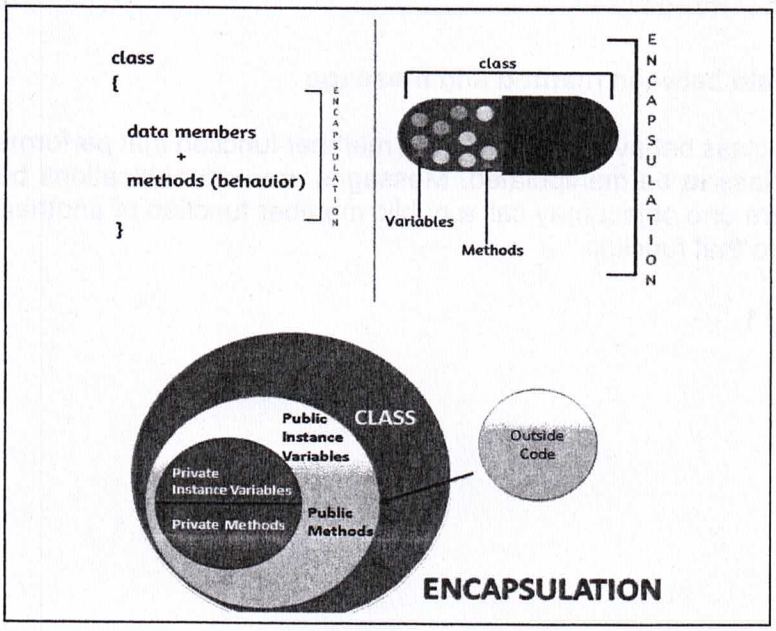


Definitions of Terms

Data Abstraction, Data Encapsulation and Information Hiding are often synonymously used in OOP. But there is a difference.

6.1 Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

6.2 Encapsulation In encapsulation, data and methods [operations] are combined together within a single unit [called an object] from being modified by accident. It is used to restrict access to data and methods.

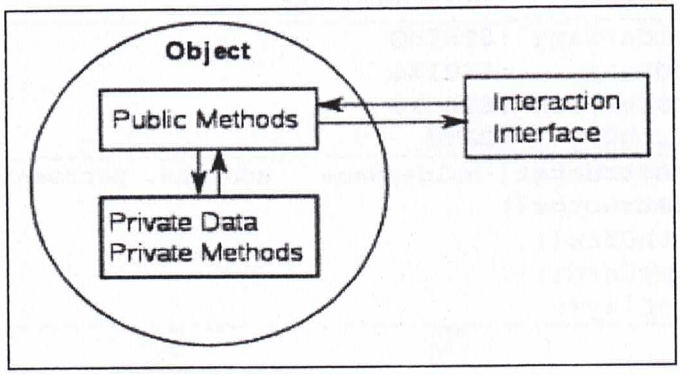


- Encapsulation is used as an information hiding mechanism, where the data structure of a class can be hidden, with a public interface provided by functions.
- Encapsulation prevents our data from unwanted accesses by binding code (methods) and data (attributes) together into a single unit called an object.
- Encapsulation supports the notion of gathering together related aspects of a design, grouping them and hiding the implementation. It therefore encourages high cohesion. The combination of information hiding with a well-defined set of interface methods also supports low coupling as it prevents a component of the design gaining access to the internals of an implementation.
- Encapsulation is often accomplished by providing two kinds of methods for attributes. (accessor and mutator)
- 'Encapsulation' and 'Data Hiding' are difference

Question: Explain the basic features and advantages of encapsulation.
Answer: Encapsulation is the combination of data and the operations that act on the data into a single "program unit" called an object. The advantages are that it allows for information and data hiding.

6.3 Information (Data) hiding

- **Data hiding** hides internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.
- **Data hiding** a mechanism for hiding the details of the implementation of an object from the code that uses it.
- **Data hiding** prevents external code accessing the internal member functions (methods) and data members of a class. It ensures that external code does not have access to partial information. It ensures that contracts between the provider of a class and the user of a class are fulfilled.



Question: Explain the basic features and advantages of information and data hiding.
Answer: Once encapsulated into an object both the data members and the details of the implementation of the member functions can be hidden. This allows the object to be used at an abstract level

6.4 Data encapsulation via methods doesn't necessarily mean that the data is hidden. You might be capable of accessing and seeing the data anyway, but using the methods is recommended. Finally, data abstraction is present, if both data hiding and data encapsulation is used. This means **data abstraction** is the broader term.

Data Abstraction = Data Encapsulation + Data Hiding

Question: Explain why data encapsulation in OOP, the data is not necessary be hidden.

7 Support vs. Service:

- **Support methods**, which are typically set with private or protected visibility, are used to assist other methods (functions) in performing internal tasks, but are not available to call through the object.
- **Service methods**, which are set with public visibility, provide services to the user of an object, and may be invoked through the public interface of an object.

8 Inheritance

Inheritance is when an object or class is based on another object (prototypal inheritance) or class (class-based inheritance), using the same implementation (inheriting from an object or class) specifying implementation to maintain the same behaviour (realizing an interface; inheriting behaviour).

Example 23

Base class

Class:	**AccountHolder**
Private:	HolderName :**STRING** Address :**STRING** Profession :**STRING** BirthDay :**DATE**
Private: Public:	**Constructor(holderName, address, profession, birthday)** **Destructor()** WithDraw() NameCard() Display()

Derived class

Class:	**Saving_Account**
Private:	balance
Private: Public:	Constructor() Destructor() SetBalance(NewBalance) GetBalance() WithDraw(Amount) Deposite(Amount) DisplayRecord()

Derived class

Class:	**Fixed_Deposite_Account**
Private:	Fixed_Amount
Private: Public:	Construct() Destructor() SetFixDep(NewFixAmt) GetFixDep() Renew(NoYear) WithDraw(Amount) DisplayRecord()

- **Inheritance:** Promotes the **reusability** of code; a **sub-class/derived class/child** acquires all the heritable features of its **superclass/base class/**parents, and may use them and extend upon them.
- **Single inheritance** is where subclasses inherit the features of **one** super class. A class acquires the properties of another class.
- **Use of inheritance** in the development of object-oriented programs.
    - **Inheritance** is a mechanism used to create new classes which are based on existing classes. The class created is called a **subclass (derived class)** whilst the class on which the new class is based is called a **superclass (base class)**. In inheritance, the subclass introduces new data above and beyond that held in the superclass and/or new methods in addition to those implemented in the superclass.
    - The major **use of inheritance** is to permit **incremental development**. In this type of development existing classes are **reused** and new functionality is provided by building classes which inherit from them.

Example 24 Protected visibility in Python

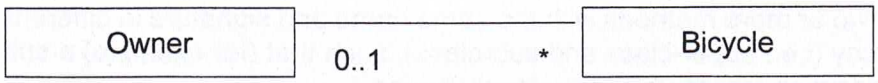
```
class MyClass():
    _a = 0              ## protected integer
    def increment():
    _a = _a + 1
class MySubclass(MyClass):
    def decrement():
    _a = _a - 1
```

The class MyClass has a method which increments the instance variable a. If we wish to create a class which will both increment and decrement the variable we do not need to copy the code which increments the variable. We simply introduce a class which inherits the functionality from the existing class and adds additional code to implement the new function.

Question: Explain the basic features and advantages of inheritance.
Answer: Inheritance allows one object to be derived from another. The derived object has all the data members and member functions of the original object and any additional data member or member functions that are defined within it. Even previously defined functionality may be redefined with the appropriate functionality applied to the particular object that invokes it. When functions (including constructors) are redefined in a derived object, they completely override the original function.

9 Association, Aggregation and Composition

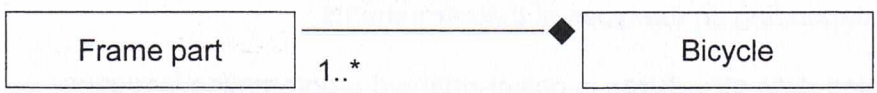
- **Association** defines **a relationship between classes of objects** that allows one object instance to cause another to perform an action on its behalf. Association refers to **"has a" relationship between two classes which use each other.**



- **Aggregation** refers to "has a" + relationship between two classes where one contains the collection of other class objects. For example, patient has-a doctor, but both patient and doctor may still sensibly exist without each other.

Car has an engine

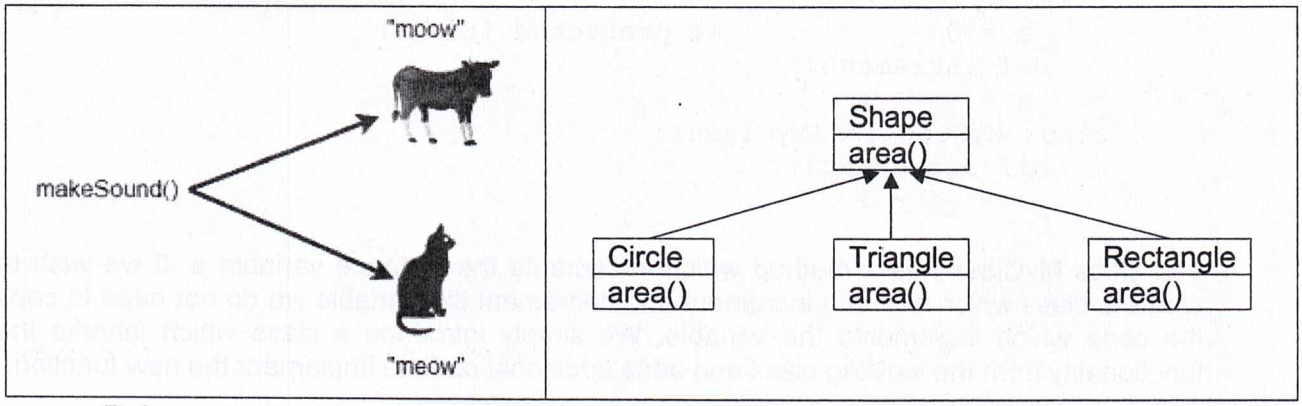
- **Composition** represents an "**owns-a**" relationship between classes. Unlike aggregation, the subservient class does not have an existence independently of the class that it is owned by. For example, a linked list node may be owned by a linked list, but may not exist without the linked list that it is contained within.



A book owns pages

10 Polymorphism

- **Polymorphism** is the ability of different objects to behave in different ways when the same message is applied to them.



- **Polymorphism** is the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations.
  In strongly typed languages, polymorphism usually means that type A somehow derives from type B, or type A implements an interface that represents type B.
- The primary usage of polymorphism is the creation of **objects** belonging to **different types** to **respond to method**, field, or property **calls of the same name**, each one **according** to an appropriate **type-specific behaviour**. The programmer (and the program) does not have to know the exact type of the object in advance, and so the exact behaviour is determined at run time.

11 Implementation of polymorphism - Method Overriding and Overloading

Reused of the same method name for objects of different classes

11.1 Method Overriding

Creating of two or more methods with the same name and signature in different classes in the class hierarchy (i.e., super-class and sub-class), such that (for example) a sub-class version can overshadow an inherited (super-class) version.

11.1.1 Method overriding allows a subclass to provide its own implementation of a method already provided by one of its superclass. A subclass can give its own definition of methods which also happen to have the same signature as the method in its superclass.

```
class AccountHolder():
      def WithDraw(self):
      .
      .
class Saving_Account (AccountHolder):
      def WithDraw(self):
      .
```

The method `WithDraw` is overridden.

11.1.2 Operator overloading is a specific case of polymorphism in which some or all of operators like +, = or == are treated as polymorphic functions and as such have different behaviours depending on the types of their arguments.

11.1.3 Defining **data structures** in object-oriented programming languages
A data structure is a specialized format for organizing and storing data. Data structures may be characterised by the operations that can be performed on them. For example, a **stack** is characterised by the operations *push* and *pop*. In object-oriented programming languages data structures are made available to users via classes which implement the methods which characterise the data structure. The classes describe complex objects whose constituent parts may themselves be complex objects.

11.2 Overloading [not in syllabus]

- **Method Overloading**: Having several versions of the same method name, but with different signatures. The appropriate version is executed, depending upon the arguments passed when the method is invoked.
- **Method overloading** is a feature found in various programming languages that allows the creation of several functions with the same name which differ from each other in terms of the type of the input and the type of the output of the function. In overloading the name of the function is the same but some other part of the signature is different.
- **Overloaded methods** are those that have the same name, but that have different formal parameter types (or result value type, if the language supports overloading on result type).

12 Benefit of modularity in OOP

12.1 Because object-orientation provides modularity, it also provides the generally assumed benefits of modularity, namely:

- **Reusability** Programs can be assembled from pre-written software components that can be used in many different applications.
- **Extensibility** New software components can be written or developed from existing ones without affecting the origin components

12.2 Differentiates a 'module' from an 'object'

A module is a 'functionally decomposed' piece of code, which performs a particular process, or set of processes. An object is a unit of software comprising both data and process, which provides some meaningful behaviour

13 Code re-use in OOP

The modular unit (class definition) in object-orientation (the 'abstract data type', which provides the blueprint for objects of a specific type) encapsulates both data and process. Because abstract data types hide their internal information behind a consistent 'public' interface, they can be easily ported from one application to another. These units therefore provide the reusability absent from other methods in that not only can they be used in many applications without modification, but they can also be extended, without corrupting what already exists.

13.1 The various aspect (Encapsulation, inheritance, Polymorphism) of object-orientation contributes to reuse

There are various contexts in which reuse are possible:

- Encapsulation allows us to reuse an abstract data type to create multiple objects, and
- inheritance allows classes to be reused in the definition of other classes.
- Polymorphism means the reuse of symbols (operators and names) to apply to different object behaviours, while aggregation reuses existing classes to provide components for larger objects.

13.2 Software reuse

Contribution that abstraction, encapsulation and data hiding make to the potential of a language to encourage software reuse.

By identifying the most general cases and hiding the details of specific implementations from the code using it, it is possible to develop set of classes which operate on all objects rather than just objects belonging to specific classes. Such classes are easily reusable.

14 Problems Associated with object-orientation

Specific problems encountered when applying the object-oriented approach to a software problem include:

1. Resource Demands
   Since an object-oriented program can require a much greater processing overhead than one written using traditional methods, it may work more slowly.
2. Object Persistence
   An object's natural environment is in RAM as a dynamic entity. This is in contrast to traditional data storage in files or databases where the natural environment of the data is on external storage. This causes problems when we want objects to persist between runs of a program, even more so between different applications.
3. Reusability
   It is not easy to produce reusable objects between applications when inheritance is used, since it makes their class closely coupled to the rest of the hierarchy. With inheritance, objects can become too application specific to reuse. It is extremely difficult to link together different hierarchies, making it difficult to coordinate very large systems.
4. Complexity
   The message passing between many objects in a complex application can be difficult to trace and debug.

15 Testing

15.1 Testing object-oriented software

Testing would initially centre on individual classes. Each public method of class would require testing. The testing methodology could consist of both white and black box testing. In white box testing we test the paths through the code and provide data which exercises every possible path through the code. In black box testing, test data is developed by considering the specification of the methods. Using the specification, a number of equivalence partitions are developed, and data is selected from each partition and from the boundaries of the partitions. Once classes have been tested it is not necessary to retest them if their methods are inherited. It is, however, necessary to test inherited and overridden methods in the context of the new class.

After **individual classes** have been tested using one or both of the techniques referred to in part a) they are often combined and **tested as a group**:

Integration testing is a technique for testing assemblages of classes which will previously been tested individually.

In big bang integration testing, the majority of the components of the system are put together. This is a very quick way of building the system but requires careful design of the test data.

In bottom up testing the components at the lowest levels of the hierarchy are combined and tested first. The software is then put together by including successfully higher level components. In contrast in **top down testing** the higher levels are integrated first and tested. Initially lower levels will be stubs but as testing continues the stubs are replaced by actual classes.

OOP Tutorial 2

TEMASEK
JUNIOR COLLEGE

Temasek Junior College
H2 Computing
OOP Concepts Tutorial 1

***Q1** The librarian of a school library wishes to keep records of the books held in the library. Each book has the following data recorded:

- **CatalogueNo** is used to identify a particular book and is seven digits. The first four digits represent the year that the book was placed in the library and the last three digits are used to make the CatalogueNo unique e.g., 2007103.
- **Title** represents the title of the book and is at most 30 characters.
- **Author** represents the author(s) of the book and is at most 30 characters.
- **Format** is a single character and is used to indicate whether the book is of standard size (S), large print (L), oversize (O) or paperback (P).

Draw a diagram that shows suitable class include appropriate attributes and methods in the class. **[4]**

***Q2**

a) Briefly describe the following in the context of object-oriented programming:
   i) Class.
   ii) Object. **[4]**

b) Describe two types of class members used in object-oriented programming languages. **[6]**

c) Explain how three types of class member visibility are used in object-oriented programming. **[6]**

***Q3** A class has a single constructor which is only visible within the class. What purpose might this class serve? **[5]**

Q4 Some object-oriented languages allow the programmer to write methods known as destructors. Explain the term destructor, the purpose of these methods, and at what point they are invoked. **[6]**

Q5 Define *class variable* and *instance variable* and provide a code fragment that implements these concepts in an object-oriented programming language of your choice and demonstrates how they may be used. **[10]**

Q6 What is abstraction in OOP? **[2]**

Q7 Explain what is meant by the term class member visibility. **[6]**

Q8 Describe three modifiers that can be used to define the visibility of class members (fields/data members and methods/member functions). **[6]**

***Q9** Explain the difference between a class and an object. **[2]**

Q10

a) Describe the features that differentiate object-oriented programming languages from structured programming languages which do not support objects. **[10]**
b) What is the difference between an object in an object-oriented language and a variable in a structured programming language?

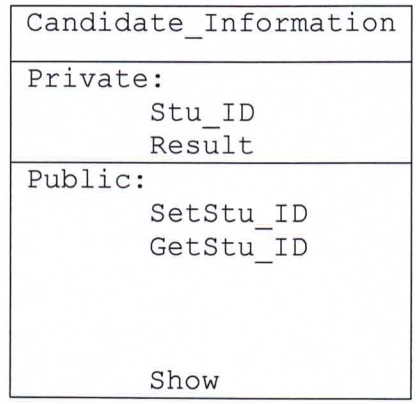
***Q11** Distinguish between the following pairs of methods.

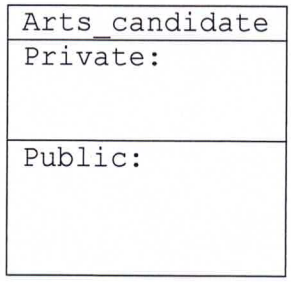
a) instance and class
b) accessor and mutator
c) constructor and destructor

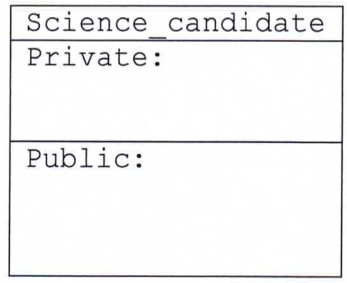
11.2 Overloading [not in syllabus]

- **Method Overloading**: Having several versions of the same method name, but with different signatures. The appropriate version is executed, depending upon the arguments passed when the method is invoked.
- **Method overloading** is a feature found in various programming languages that allows the creation of several functions with the same name which differ from each other in terms of the type of the input and the type of the output of the function. In overloading the name of the function is the same but some other part of the signature is different.
- **Overloaded methods** are those that have the same name, but that have different formal parameter types (or result value type, if the language supports overloading on result type).

Example 25
Candidates are categorised as Arts and Science streams.
Arts stream candidate takes examination in History and Geography.
Science stream candidate takes examination in Combined Science.
Copy and **complete** the following class diagram: **[7]**







Use this example to explain the following terms:

(a) encapsulation **[3]**
(b) inheritance **[3]**
(c) polymorphism **[3]**

OOP Tutorial 2