Procedures and functions are the building blocks of the application in the procedural approach. However, in the OO approach, it is the objects which are the building blocks.

**Advantages of OOP:**

**Simplicity:** Software objects model the real world objects. Hence the complexity is reduced and the

program structure is very clear.

**Modularity:** Each object forms a “separate entity” whose internal workings are decoupled from other parts of the system.

**Modifiability**: It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only “public interface” that the external world has to a class is through the use of “methods”.

**Extensibility:** Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

**Maintainability:** Objects can be separately maintained, thus making locating and fixing problems easier.

**Re-usability:** Objects can be reused in different programs.

**What is an Object? What comprises an object?**

An object in the real-world can be physical, conceptual, or a software entity.

**Each object is characterized by identity, state, and behavior.**

**Identity:** It distinguishes one object from another. Two books of same title are still two different books - they are two instances of a “book” which happen to have similar properties, just as there will be two copies if they existed in the library. The identity of one is to be distinguished from the other.

**What is meant by state of an object?**

**State:** It comprises set of properties of an object along with its value. It is one of the possible conditions that an object may be in. It is indicated by the set of values that each of its attributes possesses. For example: An account object may be in an active or suspended state depending on the balance that it possesses.

The current state of an object is defined by the set of values of its attributes and the links that the object has with other objects. The current state of an object is said to have changed, if one or more attribute values change. The object remains in control of how the outside world is allowed to use it:

**-**by assigning a state (e.g., account number, Account type, balance) to itself, and

**-**by providing methods for changing that state

**What is meant by behavior of an object?**

**Behavior:** It is the manner in which an object acts or reacts to request received from other objects. It is what an object does when it receives instructions. For example: Deposit or withdrawal that occurs against an account object.

**What is a Class?**

Classes describe objects that share characteristics, methods, relationships, and semantics. Each class has a:

* **Class name :** Customer
* **Class attributes :** Name, Address, Email-Id (its values determine state of an object)
* **Class operations:** displayCustomerDetails(), changeContactDetails() (which provides the behavior for the object).

**What is the relationship between objects and classes?**

What exist in real world are objects. When we classify these objects on the basis of commonality of structure and behavior, what we get are classes. Classes are “logical” they don’t really exist in real world. While writing software programs, it is the classes that get defined first. These classes serve as a blueprint from which objects are created.

**Class Attribute and Operation:**

A class has named properties, which are attributes of the class. An attribute would be of a specific type. At runtime, an object will have associated values for each of its attributes. A class can have several operations. An operation is an implementation of a service that can be requested from an

Object. When an operation of an object has to be invoked by another object, it passes a “message” to the object. Messages would correspond to the operation name.

**Different access specifiers?**

**Access Modifiers:**

Access Modifiers specify how members of a class will be accessible. OOP supports the following three types of access modifiers:

**Public:** means accessible to all. An object can access the public variable outside its own class.

**Private**: means accessible only to the own class. An object can access the private variable only in its own class.

**Protected**: means accessible to own class and its subclass. An object can access the protected variable only in its own class or its subclass.

Information Hiding means hiding all the details of an object that do not contribute to its essential characteristics. To this end, access modifiers help to restrict the accessibility to attributes and operations.

**Access an attribute or an operation:**

To access members, a dot operator is used against the corresponding Account object. For example: If MyAccount is an object of Account type, to display the details of this object, we would specify MyAccount.displayAccountDetails(). From where an attribute or operation can be accessed depends on the access modifiers. For e.g. if the Account Class had a private operation for calculating interest, this operation can only be invoked from within another operation of the same class. Both attributes of the Account Class are private here, which means they are not accessible from outside this class.

**What is a Constructor and when it gets called? What is a Destructor and when it gets called?**

**Constructors and Destructors:**

Objects are created using Constructors. The Constructors are special member functions of the class that share the same name as that of the class. When objects are created, memory is set aside for them. The attribute values of objects may be initialized, if needed. A class may have multiple constructors since we may want objects to be created and attributes initiated in various ways. The constructor that takes no parameters is a default constructor, while constructors that take 1 or more parameters are the parameterized constructors.

When objects are no longer needed, Destructors come into play. The most common use of destructors is to de-allocate memory that was allocated for the object by the Constructor. There can

be only 1 destructor for a class. The destructor name is same as class name, and preceded with tilde (~) sign. Eg. ~Account().

**Attribute Types:**

**What are instance and static variables? Understand their difference**

**Local variables:** These are variables declared within a function, or method, or any block of code, and the arguments passed to a function. They are not accessible outside the block in which they

are defined.

**Instance variables:** These are declared as a part of class. We have seen examples of such variables earlier – AccountNumber, Balance, etc. These are accessible by all the functions written

within the class.

**Why static variables are used?**

**Class variables:** They are also called Static variables. Such variables are useful when all the

objects of the same class need to share a common item of information.

**Object-Oriented Principles:**

Object-Oriented technology is built upon a sound engineering foundation, whose elements are collectively called the “object model”. This encompasses the following principles –

* Abstraction
* Encapsulation
* Modularity
* Hierarchy

**What is the meaning of abstraction?**

Abstraction is determining the essential qualities. By emphasizing on the important characteristics and ignoring the non-important ones, one can reduce and factor out those details that are not essential, resulting in less complex view of the system. Abstraction means that we look at the external behavior without bothering about internal details. Abstraction is domain and perspective specific.

**What is the meaning of encapsulation?**

Every object is encapsulated in such a way, that its data and implementations of behaviors are not visible to another object. Encapsulation allows restriction of access of internal data. Encapsulation is often referred to as information hiding. However, although the two terms are often used interchangeably, information hiding is really the result of encapsulation, not a synonym for it.

**Encapsulation versus Abstraction:**

The concepts of Abstraction and Encapsulation are closely related. In fact, they can be considered like two sides of a coin. However, both need to go hand in hand. If we consider the boundary of a class interface, abstraction can be considered as the User’s perspective, while encapsulation as the Implementer’s perspective. Abstraction focuses on the outside view of an object (i.e., the interface).

Encapsulation (information hiding) prevents clients from seeing its inside view, where the behavior of the abstraction is implemented. The overall benefit of Abstraction and Encapsulation is “Know only that, what is totally mandatory for you to Know”. Having simplified views help in having less complex views, and therefore a better understanding of system.

Increased Flexibility and Maintainability comes from keeping the separation of “interface” and “implementation”.

When we define a blueprint in terms of a class, we abstract the commonality that we see in objects sharing similar structure and behavior. Abstraction in terms of a class thus provides the “outside” or

the user view. The implementation details in terms of code written within the operations need not be known to the users of the operations. This is again therefore abstracted for the users. The implementation details are completely encapsulated within the class.

The data members and member functions which are defined as private are “encapsulated” and users of the class would not be able to access them.

**What is meant by Modularity?**

Modularity is obtained through decomposition, i.e., breaking up complex entities into manageable pieces. An essential characteristic is that the decomposition should result in modules which can be independent of each other. As modules are groups of related classes, it is possible to have parallel

development of modules. Changes in one may not affect the other modules. Modularity is an essential characteristic of all complex systems. Well-designed modules can be reused in similar situations in other designs.

Modularity in OO systems is implemented using Components. A component is a set of logically related classes. For e.g. several classes may need to be used together for an application to retrieve data from the underlying databases. So this collection of logically related set of classes for retrieving data can be bundled together as a component for Data Access. A user of a component need not know about the internals of a component. Modularity thus helps in simplifying the complexity.

**What is meant by hierarchy?**

Hierarchy is the systematic organization of objects or classes in a specific sequence in accordance to their complexity and responsibility. In a class hierarchy, as we go up in the hierarchy, the abstraction increases. So all generic attributes and operations pertaining to the Account are in the Account superclass. Specific properties and methods pertaining to specific accounts like current and savings account is part of the corresponding sub class. Is A relationship holds true – Current

Account is an account; Savings Account is an Account. In object hierarchy, it is the containership property, where one object is contained within another object. So Window contains a Form, a Form

contains textboxes and buttons, and so on. Here we have “Has A” relationship – Form has a textbox.

**Why Inheritance Hierarchy?**

Inheritance is the process of creating new classes, called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base class, but can add some specificity of its own. The base class is unchanged by this process. Once the base class is written and debugged, it need not be touched again, but can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases the program reliability.

**How many types of hierarchies are possible? Name them**

**Single-level inheritance**: It is when a sub-class is derived simply from its parent class.

**Multilevel Inheritance:** It is when a sub-class is derived from a derived class. Here a class inherits from more than one immediate super-class.

**Multilevel inheritance:** can go up to any number of levels.

**Multiple Inheritance:** It refers to a feature of some OOP languages in which a class can inherit behaviors and features from more than one super-class.

**Hybrid inheritance:** which is essentially combination of the various types of inheritance mentioned above.

In an inheritance hierarchy, the super class is the more generic class, and subclasses extend from the generic class to add their specific structure and behavior. The relationship amongst these classes is a generalization relationship. OO Languages provide specific syntaxes to implement inheritance or the generalization relationship. Has A or Containment is further of two forms depending on how tight is the binding between the container (“Whole”) and its constituents (“Part”). In the whole-part relationship, if the binding is loose i.e. the contained object can have an independent existence, the objects are said to be in an aggregation relationship. On the other hand, if the constituent and the container are tightly bound (E.g. Body & parts like Heart, Brain.), the objects are said to be in a composition relationship.

The relationship we are most likely to see amongst classes is the Association Relationship. When two classes have an association relationship between them, it would mean that an object of one class can access the public members of the other class with which it is associated. For e.g. if a “Sportsman” class is associated with “Charity” class, it means that a Sportsman object can access features such as “View upcoming Charity Events” or “Donate Funds” which are defined within the

“Charity” class.

**Polymorphism:**

The word Polymorphism is derived from the Greek word “Polymorphous”, which literally means “having many forms”. Polymorphism allows different objects to respond to the same message

in different ways!

There are two types of polymorphism, namely:

**-**Static (or compile time) polymorphism, and

**-**Dynamic (or run time) polymorphism

Overloading is when functions having same name but different parameters (types or number of parameters) are written in the code. When Multiple Sort operations are written, each having different

parameter types, the right function is called based on the parameter type used to invoke the operation in the code. This can be resolved at compile time itself since the type of parameter is known.

On the other hand, the function calculateInterest can be coded across different account classes. At runtime, based on which type of Account object (i.e., object of Current or Savings Account) is invoking the operation, the right operation will be referenced. Overriding is when functions with same signature provide for different implementations across a hierarchy of classes. As seen in the example here, inheritance hierarchy is required for these objects to exhibit polymorphic behaviour. The classes here are related since they are different types of Accounts, so it is possible to put them

together in an inheritance hierarchy. Does that mean that polymorphism is possible only with related classes in an inheritance hierarchy? The answer is No! We can have unrelated classes participating in polymorphic behavior with the help of the “Interface” concept, which we shall study in a

subsequent section.

**Static Members:**

Static member functions are operations defined within the scope of a class. However, they can be invoked without using an instance. This means that a static function is not invoked on an “object”. However, it is instead invoked on a “class”. A static member function has access to the private data of a class. It can access “static member variables”, or given a pointer or reference to an object of the class as an argument it can access the “private instance variables” of any object passed as an argument.

**What is abstract class?**

Abstract classes are special type of base classes. In addition to normal class members, they have abstract class members. These abstract class members are methods that are declared without an implementation. All classes derived directly from abstract classes must implement all of these abstract methods. Abstract classes can never be instantiated, since the members have no implementations. For example: One does not really have just an Account, but one has an account of specific type, say an account of savings account type or current account type.

Abstract classes sit toward the top of a class hierarchy. They establish structure and meaning to code. They make frameworks easier to build. This is possible because abstract classes have information and behavior common to all derived classes in a framework. The aspects that are specific to different derived classes can be left open for implementation in the derived classes.

**What is interface?**

Interfaces only specify operations, not implementation of the operations. In that sense, they are like a contract specification, and to fulfill the contract, the corresponding class needs to provide implementations for all operations specified within interface.

Polymorphism and interfaces go hand-in-hand. Interfaces formalize polymorphism. If two objects use same methods, to achieve different, but more or less similar results, then they are polymorphic in nature.

Plug and Play architecture – allowing for easy maintainability and extensibility – are benefits of using interfaces.

**Interface versus Abstract Class:**

Abstract classes allow you to partially implement your class, whereas Interfaces contain no implementation for any members. Interfaces are used to define the peripheral abilities of a class.

An Abstract Class defines the core identity of a class and there it is used for objects of the same type. If we add a new method to an Interface, then we have to track down all the implementations of the interface and define implementation for the new method. If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.

**Scope of UML:**

The figure shown in the slide illustrates what is within the scope of UML, and what is outside of the UML scope. While it provides adequate notation and semantics to address contemporary modeling issues.Some of the things that are outside the scope of UML are:

**Programming Language:** UML is a “visual modeling language”. It is not intended to be a visual

programming language. It is a language for visualizing, specifying, constructing, and documenting

the artifacts of a software intensive system. It does have a close mapping with OO languages.

**Tools:** Language standards form the foundation for tools and process. UML defines a semantic metamodel, and not a tool interface, storage, or run-time model.

**Process:** Software development process will use UML as a common language for its project artifacts, but will use the same type of UML diagram in the context of different processes. UML is process independent.

**And What UML is NOT …**

UML is not meant to be a programming language, rather it is a language meant for modeling. By using UML, one can convey a concept or a specification but not a solution (which a program

does). UML comprises model elements, each with its own associated notation and semantics.

UML is not meant to specify a tool or repository in terms of interfaces, storage, or run time behavior.

Similarly, UML is not a process. A process will provide guidance regarding order of activities, and spell out the work products that have to be developed. They are usually domain specific. UML does not require a process. However, it enables and promotes Object-Oriented and component-based processes.

**UML Building Blocks – Views and Diagrams:**

Various views are:

**Structural view -** Structure diagrams define the static architecture of a model. They are used to model the “things” that make up a model. They are used to model the relationships and dependencies. Analysts and Designers can get the view of the structural aspects of the system through the Structural view.

**Behavioral view -** It can give important inputs in terms of performance, scalability, and throughput, which can be used by the system integrator.

**Implementation view -** This view is helpful for programmers.

**Environment view -** This view can convey decisions relating to system topology, delivery mode,

installation, and communication.

**UML Building Blocks – Mechanisms:**

There are some mechanisms available to add on to the expressive power of UML. They are broadly categorized as General mechanisms and Extension mechanisms. (**Pg-26**)