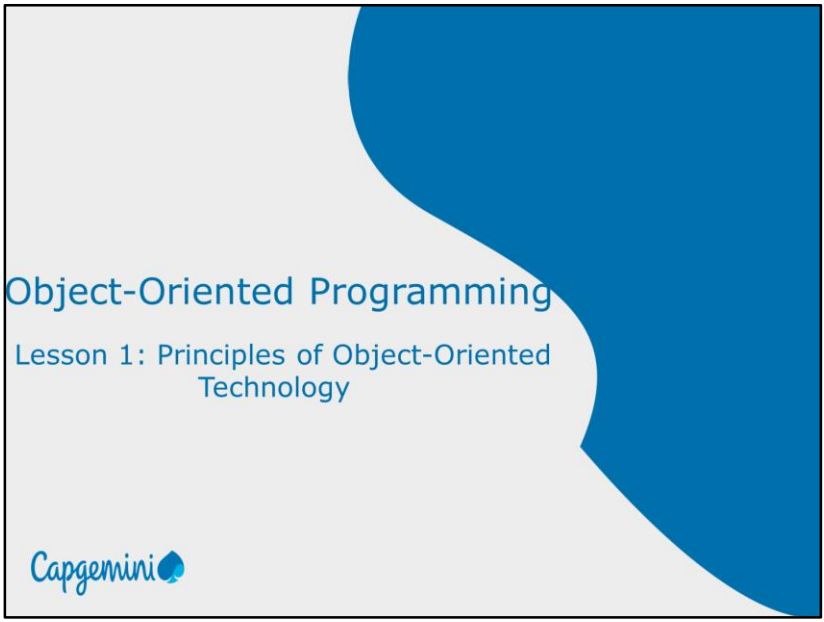

Object-Oriented Programming
Lesson 1: Principles of Object-Oriented Technology
Capgemini

Lesson Objectives

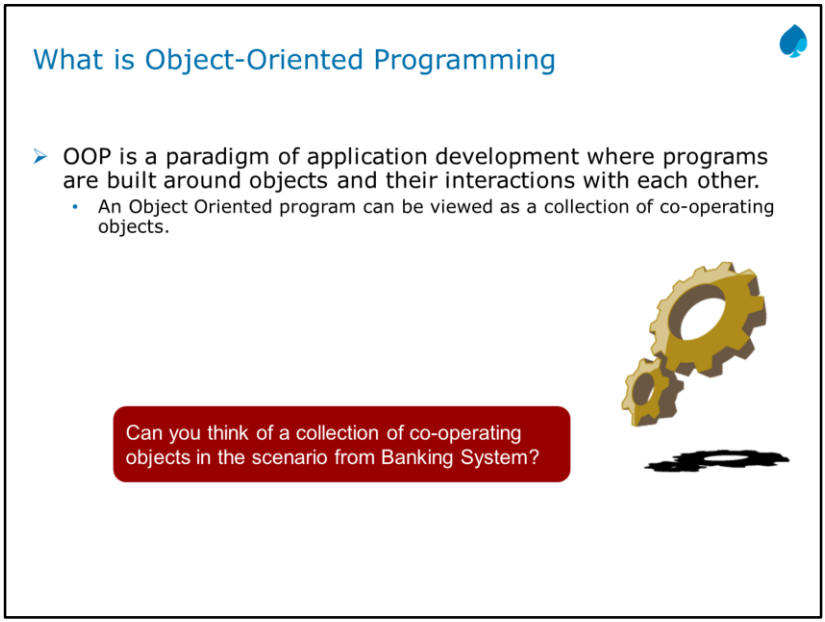
In this lesson, you will understand the basic principles of Object-Oriented technology, namely:

- What is Object-Oriented Programming?
- Why Object-Oriented Programming?
- Abstraction
- Encapsulation
- Modularity
- Hierarchy and its types
- Polymorphism and Types of Polymorphism
- What is Class
- What is Object

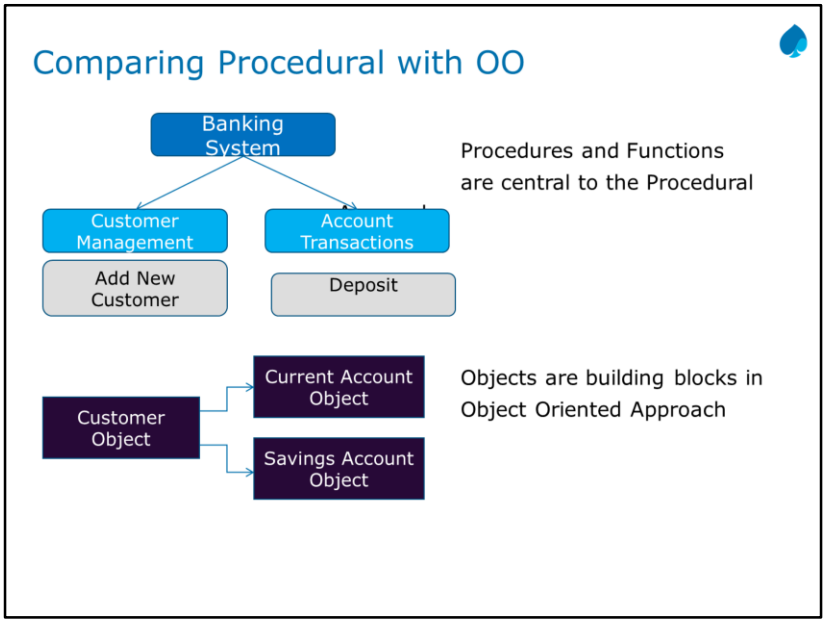
Example: Scenario from Banking System

- Geetha and Mahesh hold accounts in Bank XYZ Ltd. Geetha has a savings as well as a current account with the bank. Mahesh only has a current account. As customers of the bank, Geetha and Mahesh can deposit or withdraw money from their accounts as per the norms and policies defined by the bank on savings and current accounts.
- Bank XYZ Ltd. continuously adds new customers to its existing customer base. Of course, some its customers may also want to close their accounts due to changing needs of the customer.

If you consider this scenario for a Banking System, what services and features do you expect the bank to offer? Who are the customers mentioned for this bank? What operations can they perform on their accounts?



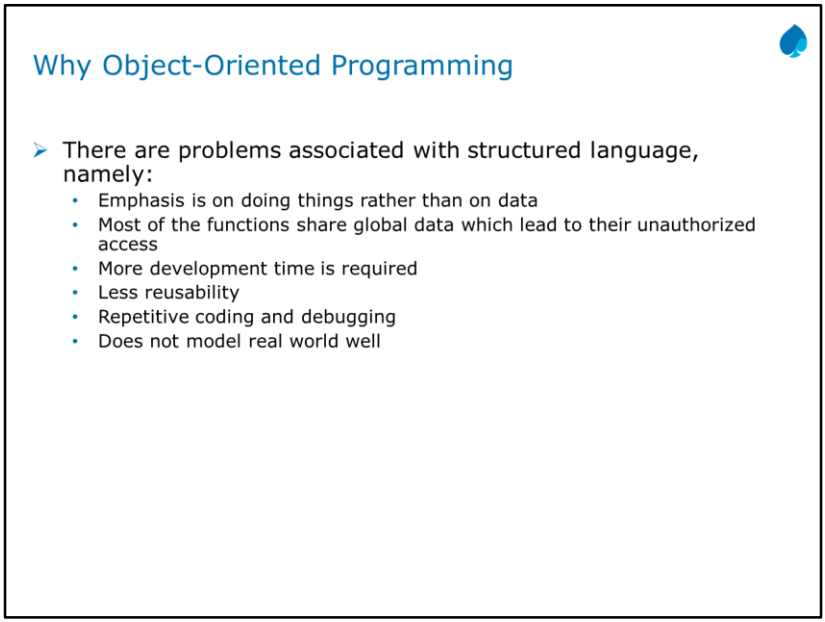
What is Object-Oriented Programming?
The object oriented approach is a fundamental shift from the procedural approach. Instead of functions and procedures being central to the program, in the OO world, we have objects that are the building blocks. An OO program is made up of several objects that interact with each other to make up the application.
For example: In a Banking System, there would be Customer objects pertaining to each customer. Each customer object would own its set of Account Objects, pertaining to the set of Savings and Current Accounts that the customer holds in the bank.
Today, most programming languages are object oriented.
For example: Java, C++, C#
Why do you think most of today's programming languages are object oriented? Are there any advantages of OO languages?



Consider the Banking System. In the procedural approach, we would try to find the top level modules of the application. Eg. A Module to maintain customers, another to maintain accounts and so on. In each of these modules, we would have procedures and functions to take care of different features. Eg. Procedure/Function to add customer or delete customer in the module for maintaining customer. Or procedures/ functions to deposit and withdraw in the module for maintaining accounts. So in the procedural approach, we identify modules, then identify procedures/functions – this is like a top down approach to system development.

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. If we reconsider the same system, we would have objects for customer “Geetha” and customer “Mahesh”; we would also have objects for their respective savings and current accounts. These objects would interact with each other for us to achieve the desired features of the application.



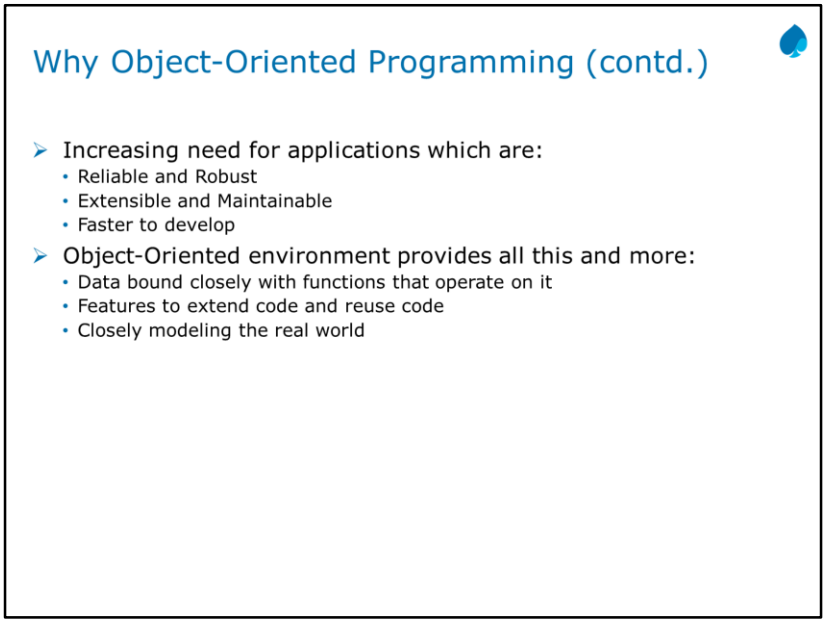
Why Object-Oriented Programming?
Before the advent of Object-Oriented technology, the primary software engineering methodology was structured or procedural programming. Some drawbacks of this approach are as follows:

Structured programming is based around data structures and subroutines. Data structures are simply containers for the information needed by subroutines. Thus, emphasis is almost entirely on algorithm required to solve a problem.

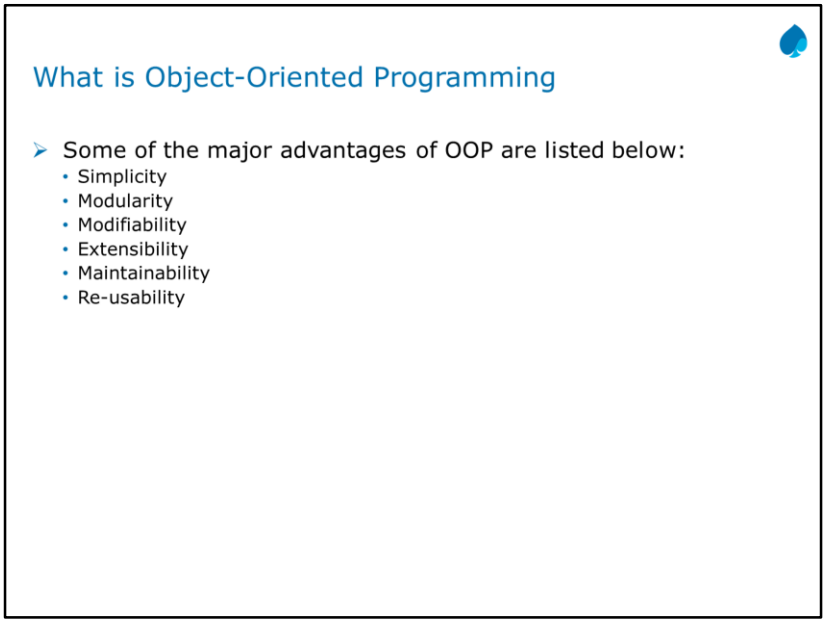
Data is openly accessible to other parts of the program, which is risky.

Structured programming tends to produce a design that is unique to that problem (thus non-reusable). Reusing code from another project usually involves a lot of effort and time. Moreover, since the emphasis is on functionality, functionality change might force entire code to be modified, thus increasing development time.

In structured programming, while analysis starts with a consideration of real-world problems, the real-world focus is lost as requirements are transformed into a series of data flow diagrams.



Why Object-Oriented Programming? (contd.)
With increasing complexity of software applications, some of the “must have” features are reliability, robustness, and maintainability. With increasing competition, high productivity which aids faster turn around times for application development and deployment is key.
Object-Oriented programs offer features which allow meeting the above goals. Binding of data and functions together means that data cannot be accessed unless designed for it. There is no possibility of mistakenly corrupting data. Decomposition in terms of objects allow for easily building new programs using existing objects and adding features to existing objects.
Moreover, the Object-Oriented world very closely models the real world, making it much more intuitive and faster to develop. The objects themselves often correspond to phenomena in the real world that the system is going to handle.
For example: An object can be an invoice in a business system or an employee in a payroll system. Thus, OO is natural way programming, i.e., “real life” objects are mapped as is in “programming” as classes / objects.



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.

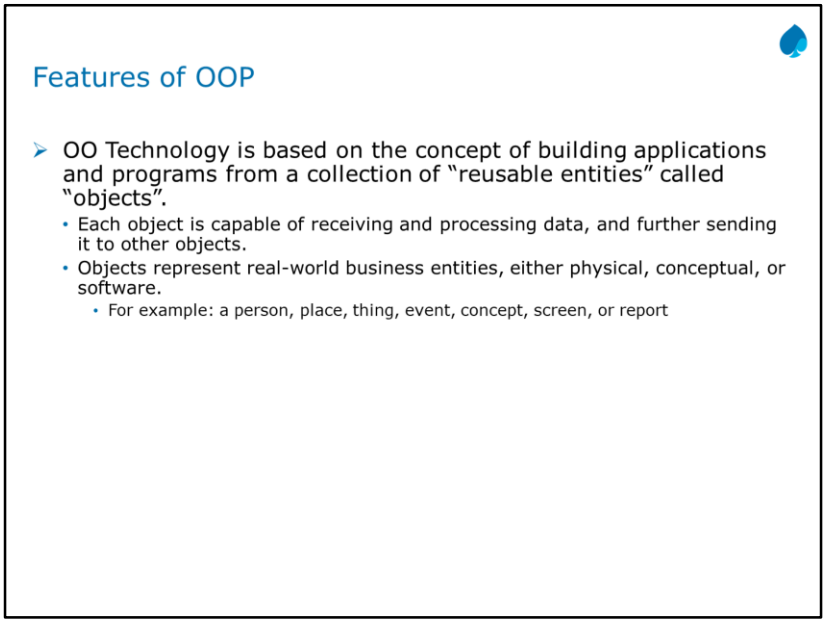
Modifiability: It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only “public interface” that the external world has to a class is through the use of “methods”.

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.

OOP is more than just learning a new language. It requires “a new way of thinking”. The idea is primarily not to concentrate on the cornerstones of procedural languages - data structures and algorithms, instead think in terms of “objects”.



Features of OOP:
Models built by using Object-Oriented technology can be smoothly implemented in any software, by using "Object-Oriented modeling language". These models also easily adjust to changing requirements. It is based on best practices. As a result, the systems developed by using Object-Oriented technology are stable with a baselined architecture. The systems are more reliable, scalable, and succinct. They are more easily maintained and adaptable to change.

What is a Class?

- A Class characterizes common structure and behavior of a set of objects.
- It constitutes of Attributes and Operations.
- It serves as a template from which objects are created in an application.

Class name	Customer
Class attributes	Name, Address, Email-ID, TelNumber
Class operations	displayCustomerDetails() changeContactDetails()

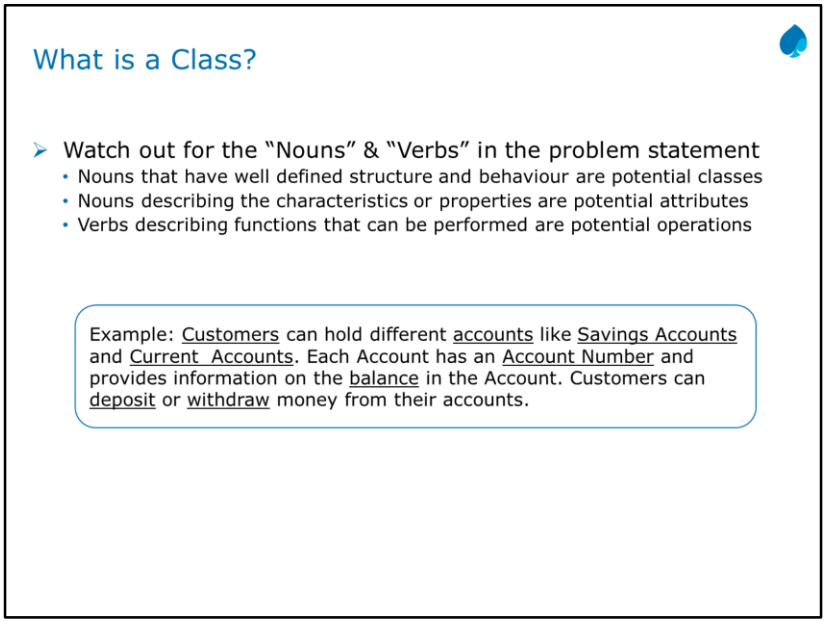
What is a Class?

Classes describe objects that share characteristics, methods, relationships, and semantics. Each class has a name, attributes (its values determine state of an object), and operations (which provides the behavior for the object).

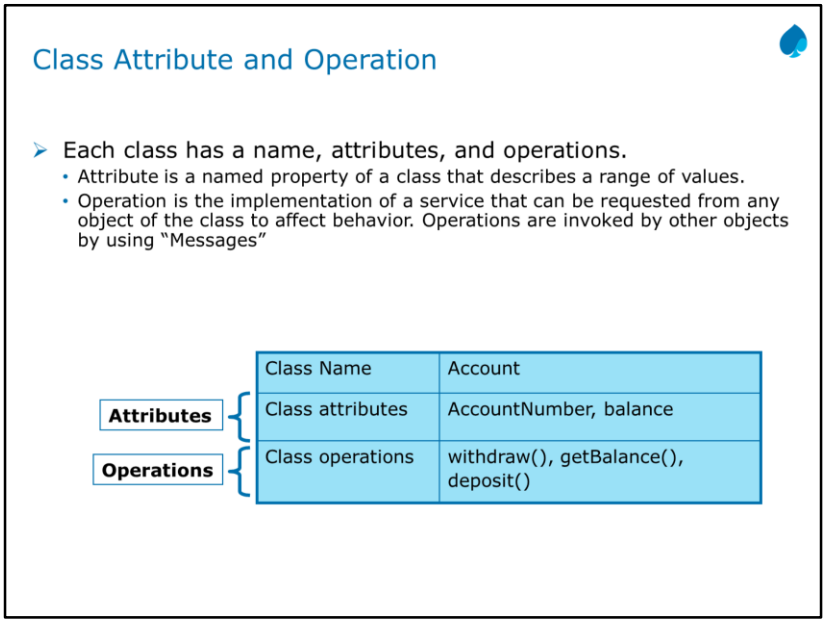
What is the relationship between objects and classes?

What exists in real world is 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. For example: In the example shown in the slide, there may be thousands of bank customers all having same set of attributes (i.e., Name, Address, Email-ID, TelNumber). Each customer is created from the same set of blueprints, and therefore contains the same attributes. Similarly, there can be thousands of Bank Accounts instantiated from the same “Account” class!

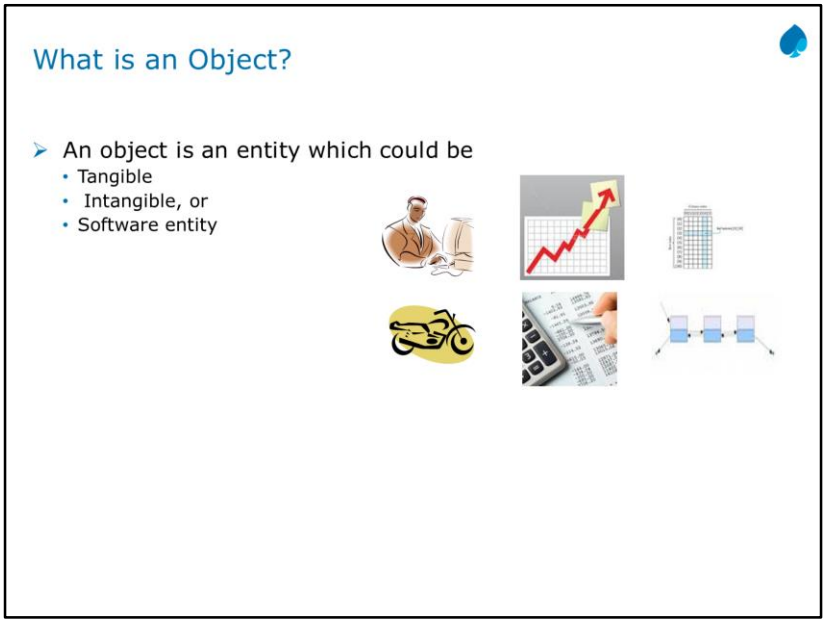
In terms of Object-Oriented technology, we say that these customers are all “instances” of the “class of objects” known as Customer. A “class” is the blueprint from which individual “objects” are created.



To help identify potential classes, their attributes and their operations, watch out for the nouns and verbs of the problem statement. A Noun having a well defined structure and behaviour, which can be a standalone entity, is a potential class. Nouns which cannot be a stand alone entity but point to properties or characteristics of something could be potential attributes. And finally, verbs describing what could be "done" are potential operations of the class.
In the example here, note that all nouns and verbs are underlined. Potential Classes could be Customer, Account, Savings Account, Current Account. Potential Attributes (of Account Class) could be Account Number and Account Balance. Potential operations (of Account Class) could be deposit and withdraw.



Getting into Details – 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.

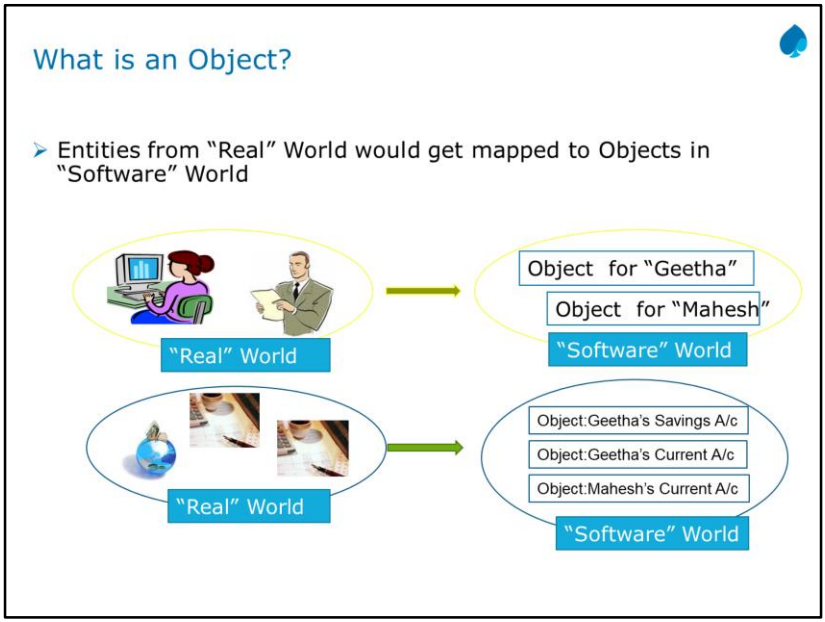


What is an Object?
An object in the real-world, can be physical, conceptual, or a software entity. We come across so many objects in the real world. In fact, everything can be considered as an object - a person, a pen, a vehicle, a book, etc. Essentially these are all tangible things that exist, can be felt, or can be destroyed.
However, there could be other entities which may not be considered as objects in the real world, like an account or a contract, or a set of business charts, or a linked list since they are “intangible” or “conceptual”. Nevertheless, they also have a well defined structure and behavior, and hence are treated as objects in the software domain.

Examples of tangible entities
Person, Pen, Vehicle

Examples of intangible entities
Account, Contract, Business Charts

Examples of software entities
Database Management System



Remember the scenario from Banking System? "Geetha" and "Mahesh" from our example of Banking System are entities in the Real World. These would get mapped into corresponding objects in the software world. So in our OO application, we will have an object corresponding to "Geetha" and another object corresponding to "Mahesh". Similarly, we would have objects corresponding to Geetha's Savings Account, Geetha's Current Account as well as Mahesh's Current Account.

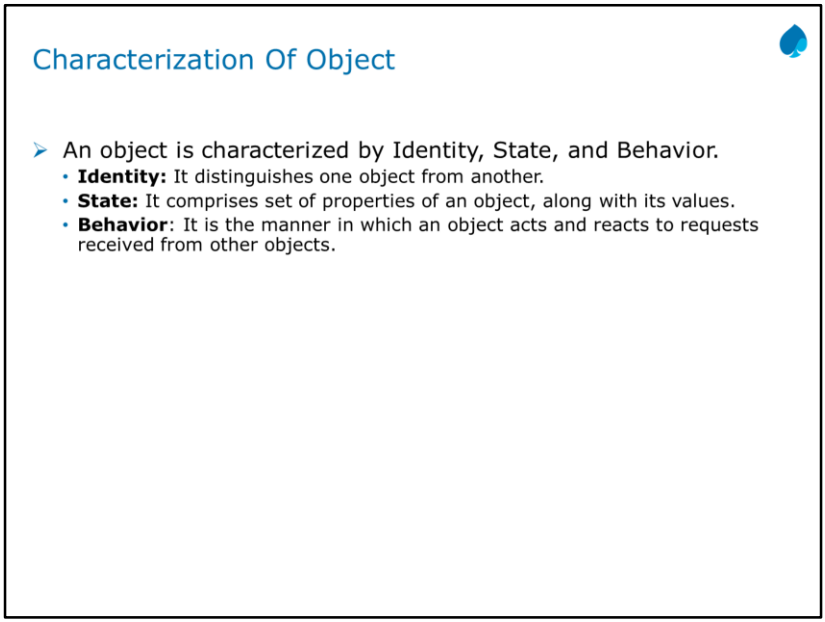
Typically objects could correspond to following

Roles played by people interacting with system (Like Geetha and Mahesh who are "Customer" objects)

Structures used for storing and processing data (Like Account objects for Geetha and Mahesh)

Other systems or devices interacting with system (Like Utility Payment System that can be accessed from Bank System)

Events and entities of the system (Like a transaction or a account status report)



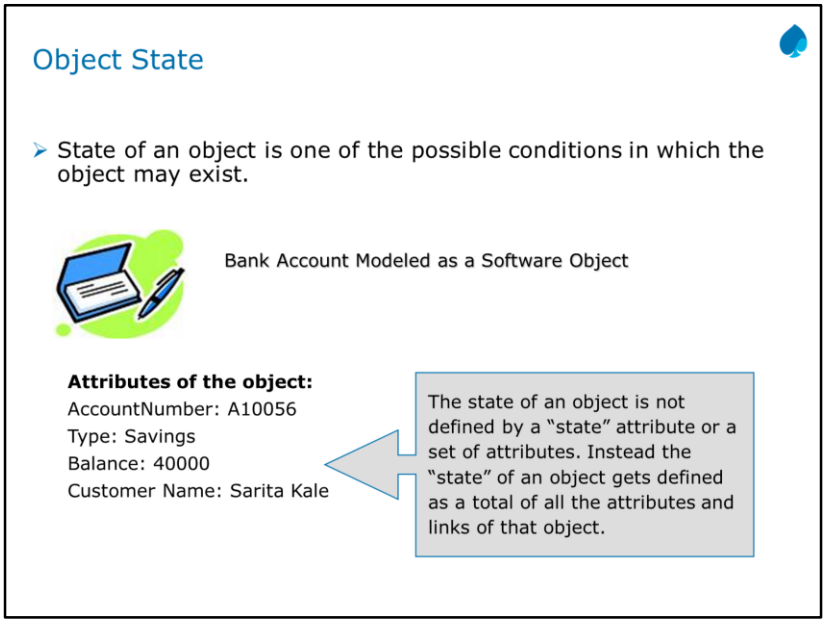
Each object is characterized by identity, state, and behavior.

Identity: 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.

State: 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.

Behavior: It is what an object does when it receives instructions. For example: Deposit or withdrawal that occurs against an account object.



What is an Object? – Object State:
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

Object State

- Behavior of the Object depends on the State of the Object

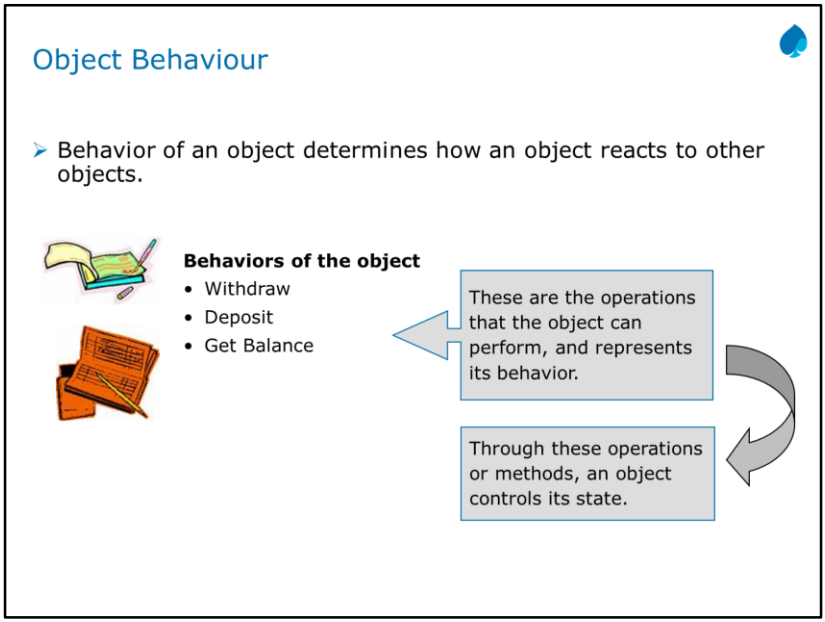
Withdrawal depends on the State of the Account Object

State where withdrawal is permitted	State where withdrawal is NOT permitted
Attributes of the object: AccountNumber: A10056 Type: Savings Balance: **40000** Customer Name: Sarita Kale	**Attributes of the object:** AccountNumber: A10056 Type: Savings Balance: **500** Customer Name: Sarita Kale

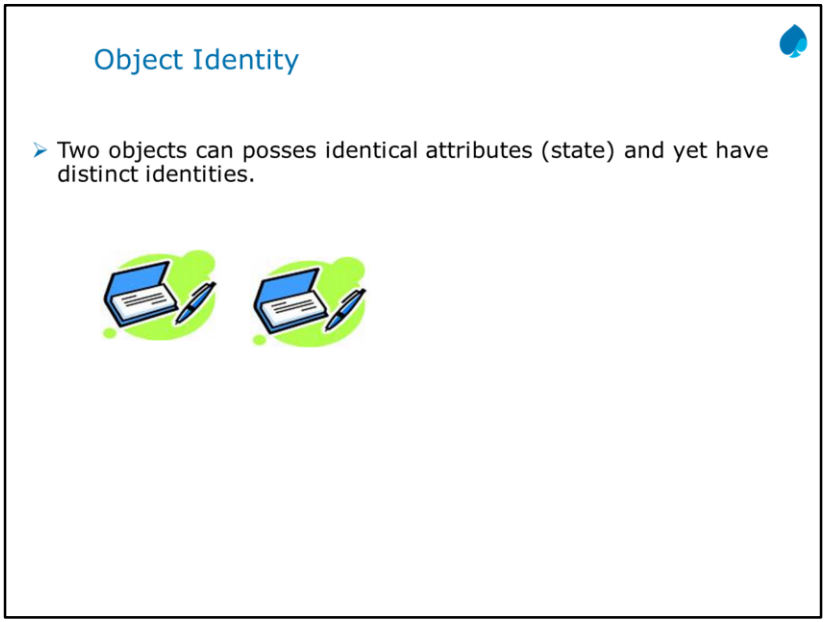
How many States of an Object should we consider??

The behaviour of an object in terms of how an object responds, depends on state of object. If the bank has a business rule on the minimum account balance, then an operation such as withdrawal will not be permitted if the balance is less than what is permitted.

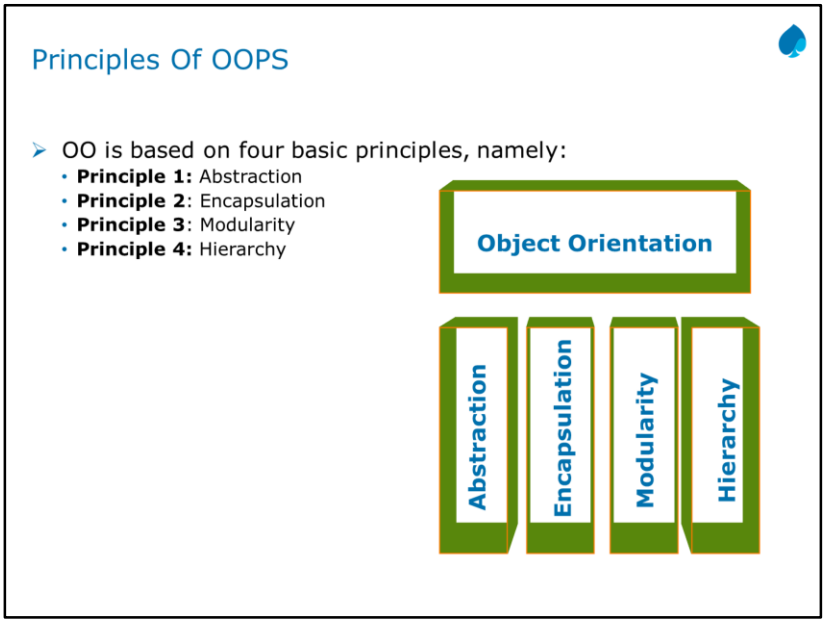
At any point, an object will be in a single state. As such, any new combination of attribute values would imply a new state for an object. That means there could be infinitely many states for each object. Should we consider each of these states?? Well No! We only need to consider the object states which will have an impact on the behaviour of the object.



What is an Object? – Object Behavior:
Behavior is what an object does on receiving a set of instructions. Object behavior is represented by the operations that the object can perform.
For example: A Bank ATM object will have operations such as withdraw, print transactions, swipe card, and so on. A bicycle object may have operations such as change gear, change speed, and so on.



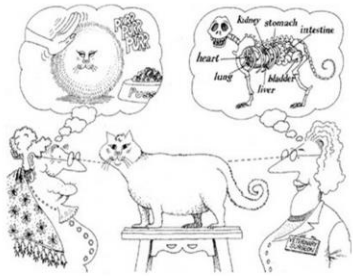
What is an Object? – Object Identity:
Two accounts may possess same attributes like type, balance, etc. Yet these two accounts have separate distinct identities. One account could be mine, and the other could be yours.
Although objects may share the same state (attributes and relationships), they are separate and independent objects with their own "unique identity".



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, and Hierarchy
Each of these principles has been discussed in detail in the subsequent slides.

Concept of Abstraction

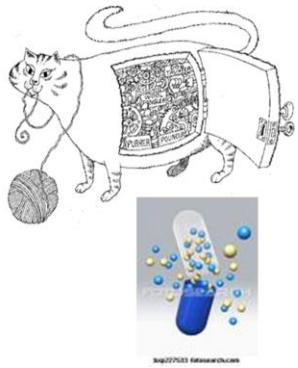
- Focus only on the essentials, and only on those aspects needed in the given context.
  - **For example:** Customer needs to know what is the interest he is earning; and may not need to know how the bank is calculating this interest
  - **For example:** Customer Height / Weight not needed for Banking System!



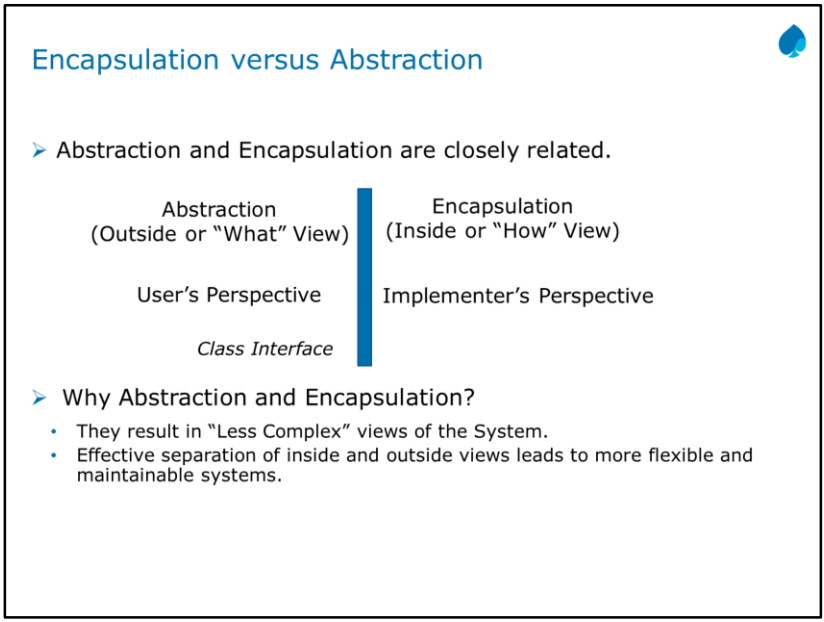
Abstraction:
Abstraction is determining the essential qualities. By emphasizing on the important characteristics and ignoring the non-important ones, one can reduce and factor out those details that are not essential, resulting in less complex view of the system. Abstraction means that we look at the external behavior without bothering about internal details. We do not need to become car mechanics to drive a car!
Abstraction is domain and perspective specific. Characteristics that appear essential from one perspective may not appear so from another. Let us try to abstract “Person” as an object. A person has many attributes including height, weight, color of hair or eyes, etc. Now if the system under consideration is a Banking System where the person is a customer, we may not need these details. However, we may need these details for a system that deals with Identification of People.

Concept of Encapsulation

- "To Hide" details of structure and implementation
  - **For example:** It does not matter what algorithm is implemented internally so that the customer gets to view Account status in Sorted Order of Account Number.



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". Developers can change implementation details without affecting the user's perspective.

Examples: Abstraction and Encapsulation

- Class is an abstraction for a set of objects sharing same structure and behavior

Customer Class

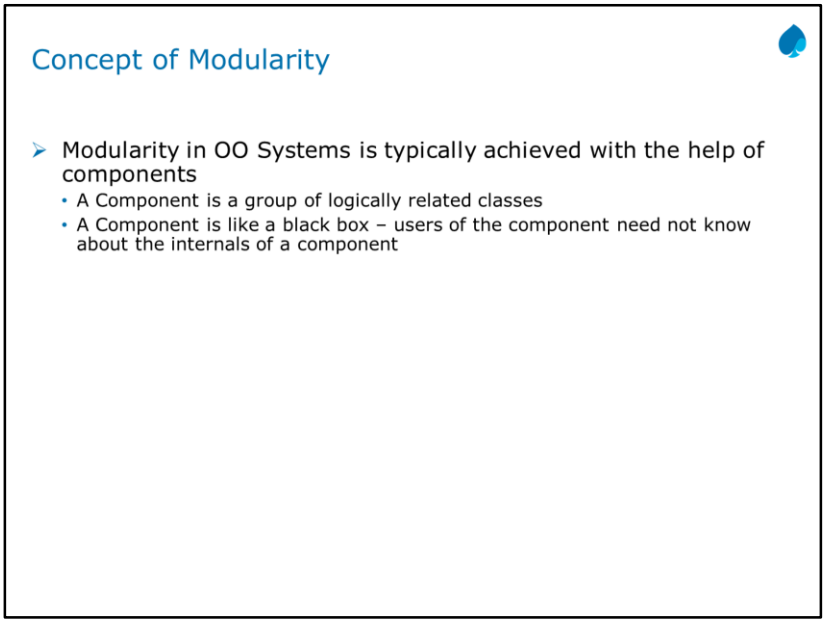
- "Private" Access Modifier ensures encapsulation of data and implementation

Abstraction and Encapsulation:
When we define a blueprint in terms of a class, we abstract the commonality that we see in objects sharing similar structure and behaviour. 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.

Concept of Modularity

- Decomposing a system into smaller, more manageable parts
  - Example: Banking System can have different modules to take care of Customer Management, Account Transactions, and so on.
- Why Modularity?
  - Divide and Rule! Easier to understand and manage complex systems.
  - Allows independent design and development, as well as reuse of modules.

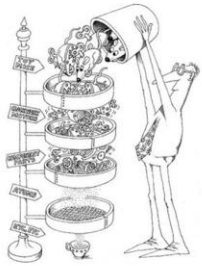
Modularity:
Modularity is obtained through decomposition, i.e., breaking up complex entities into manageable pieces. An essential characteristic is that the decomposition should result in modules which can be independent of each other.
As modules are groups of related classes, it is possible to have parallel developments 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:
Modularity is one of the corner stones of structured or procedural approach, where functions or procedures are the smallest unit of the application, and they help in achieving the modularity required in the system. In contrast, it is the class which is the smallest unit in OO Systems.
Modularity in OO systems is implemented using Components. A component is a set of logically related classes. For eg. 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.

Concept of Hierarchy

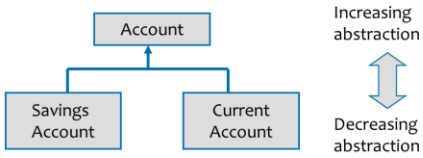
- A ranking or ordering of abstractions on the basis of their complexity and responsibility
- It is of two types:
  - Class Hierarchy: Hierarchy of classes, Is A Relationship.
    - Example: Accounts Hierarchy
  - Object Hierarchy: Containment amongst Objects, Has A Relationship.
    - Example: Window has a Form seeking customer information, which has text boxes and various buttons.



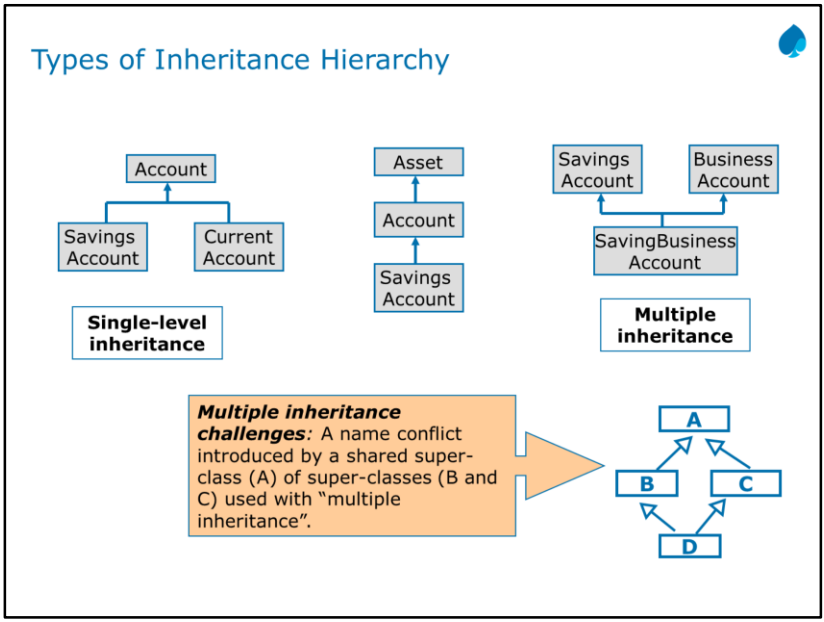
Hierarchy:
Hierarchy is the systematic organization of objects or classes in a specific sequence in accordance to their complexity and responsibility. In a class hierarchy, as we go up in the hierarchy, the abstraction increases. So all generic attributes and operations pertaining to an Account are in 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

- Why Inheritance Hierarchy?
  - It is a powerful technique that enables code reuse resulting in increased productivity, and reduced development time.
  - It allows for designing extensible software.



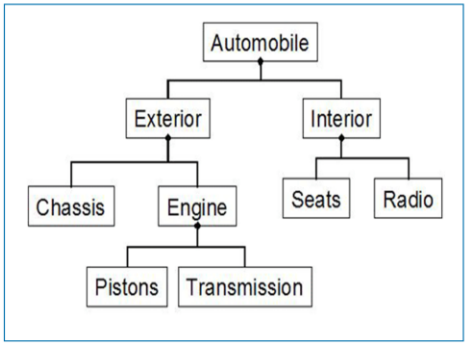
Why Inheritance Hierarchy?
Inheritance is the process of creating new classes, called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base class, but can add some specificity of its own. The base class is unchanged by this process.
Once the base 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.



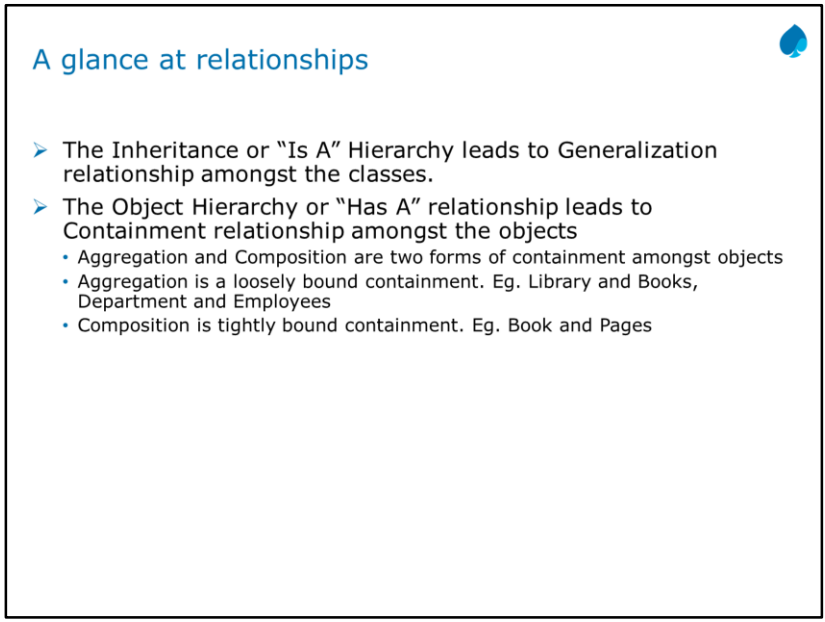
Types of Inheritance Hierarchy:
Single-level inheritance: It is when a sub-class is derived simply from its parent class.
Multilevel Inheritance: It is when a sub-class is derived from a derived class. Here a class inherits from more than one immediate super-class. Multilevel inheritance can go up to 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.
Finally, we could have Hybrid inheritance, which is essentially combination of the various types of inheritance mentioned above.

Object Hierarchy

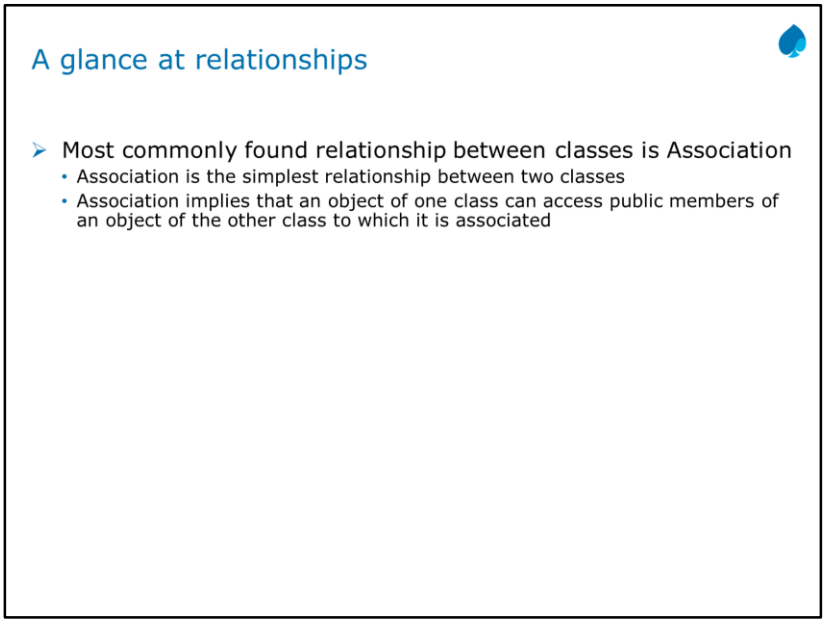
- "Has-a" hierarchy is a relationship where one object "belongs" to (is a part or member of) another object, and behaves according to the rules of ownership.



Note: The container hierarchy (has-a hierarchy) is in contrast to the inheritance hierarchy, i.e., the "generic-specific" levels do not come in here.



As seen earlier, 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 behaviour. 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 (Eg. Body & parts like Heart, Brain..), the objects are said to be in a composition relationship.



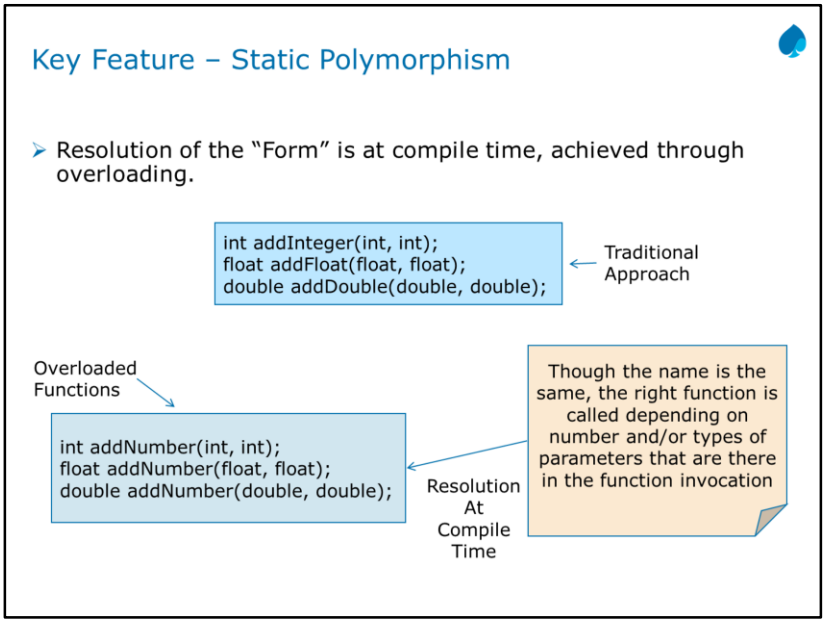
The relationship we are most likely to see amongst classes is the Association Relationship. When two classes have an association relationship between them, it would mean that an object of one class can access the public members of the other class with which it is associated. For eg. if a “Sportsman” 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.

Key Feature – Polymorphism

- It implies One Name, Many Forms.
- It is the ability to hide multiple implementations behind a single interface.
- There are two types of Polymorphism, namely:
  - Static Polymorphism
  - Dynamic Polymorphism

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:

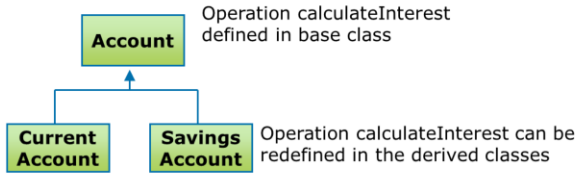
Static (or compile time) polymorphism, and
Dynamic (or run time) polymorphism



Polymorphism (contd.):
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.

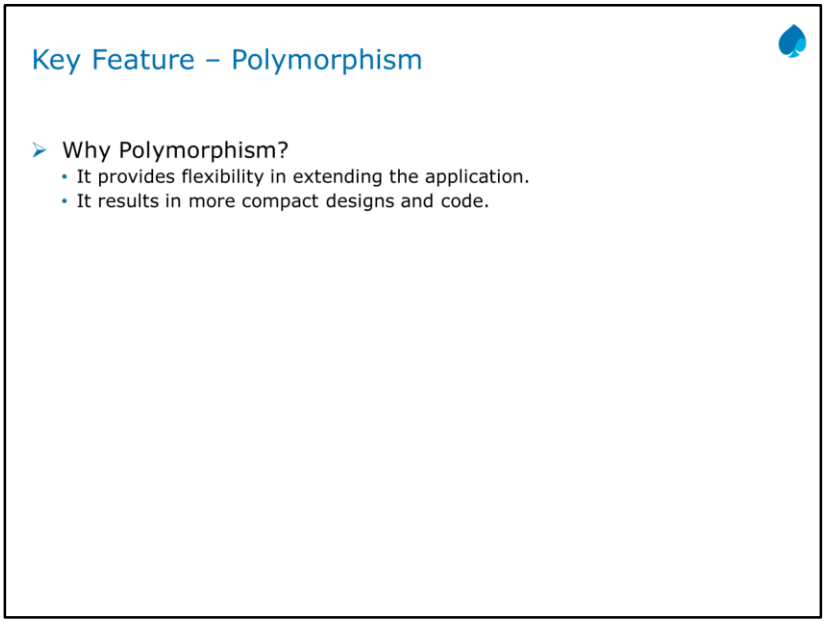
Key Feature – Dynamic Polymorphism

- Resolution of the "Form" is at run time, achieved through overriding.



The right operation defined in one of these classes is invoked at Run Time depending on which object is invoking the operation.

Polymorphism (contd.):
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.



Polymorphism (contd.):
If the banking system needs a new kind of Account, extending without rewriting the original code, then it is possible with the help of polymorphism.
In a Non OO system, we would write code that may look something like this:

```
IF Account is of CurrentAccountType THEN
        calculateInterest_CurrentAccount()
IF Account is of SavingsAccountType THEN
        calculateInterest_SavingAccount()
```

The same in a OO system would be myAccount.calculateInterest(). With object technology, each Account is represented by a class, and each class will know how to calculate interest for its type. The requesting object simply needs to ask the specific object (For example: SavingsAccount) to calculate interest. The requesting object does not need to keep track of three different operation signatures.

Lab

- Consolidated Exercise
  - Lab 1,2,3,4,5

Summary

- In this lesson, you have learnt that:
  - Class, Object, Principals Of OOAS etc.

Answers to Review Questions

Question 1: Behaviour

Question 2: True

Question 3: Static

Review Question

- Question 1: ___ determines how an object reacts to other objects.
  - Option 1: State
  - Option 2: Behavior
  - Option 3: Identity
  - Option 4: Attribute
- Question 2: A class can have zero or more number of operations.
  - True / False
- Question 3: ___ variables are accessible by all the functions written within the class.

Answers to Review Question

Question 4: Abstraction,Encapsulation, modularity,hierarchy

Question5: True

Question 6:Has-a

Review Question

- Question 4: The 4 basic principles of Object Model are ___, ___, ___ and ___.
- Question 5: Function Overriding is kind of polymorphism.
  - True / False
- Question 6: ___ hierarchy is a relationship where one object behaves according to the rules of ownership.

Answers to Review Question

Question 4: Option2

Question 5. Interfaces

Review Question

- Question 10: Abstraction focuses on:
  - Option 1: implementation
  - Option 2: observable behavior
  - Option 3: object interface

- Question 11: Polymorphism can be achieved by:
  - Option 1: Hierarchy of Classes providing polymorphic behavior
  - Option 2: Interfaces
  - Option 3: Containment of Objects