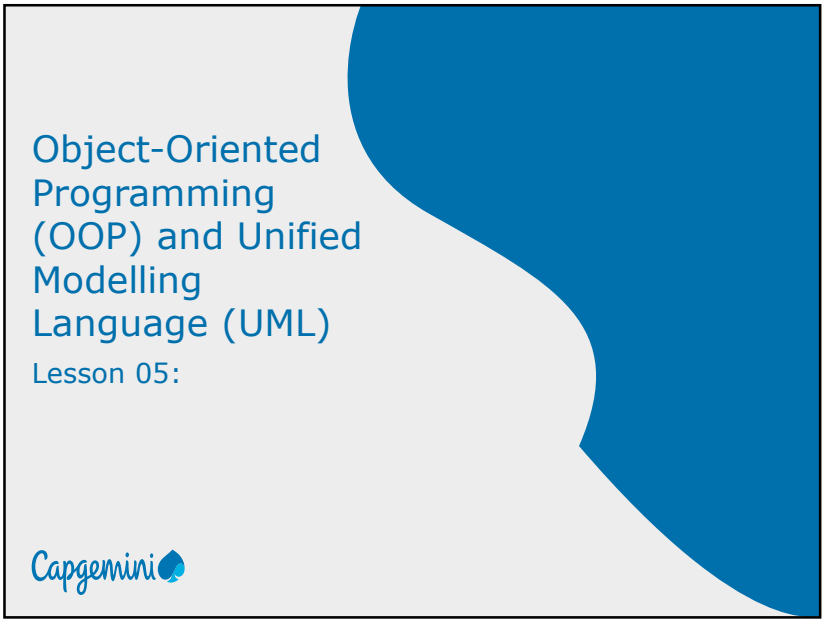
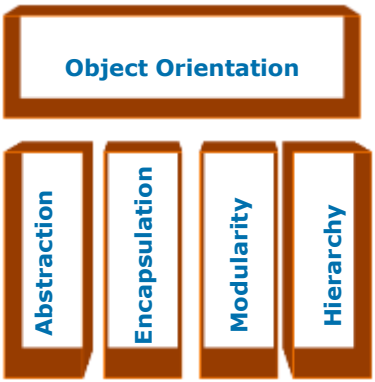


Lesson Objectives

- To understand the following concepts
  - Principles in Object-Oriented technology
    - Abstraction
    - Encapsulation
    - Modularity
    - Hierarchy
  - Polymorphism
  - UML
    - Use Case Diagram
    - Sequence Diagram
    - Class Diagram
  - Demo
  - Case study

Introduction

- OO is based on four basic principles, namely:
  - **Principle 1:** Abstraction
  - **Principle 2**: Encapsulation
  - **Principle 3**: Modularity
  - **Principle 4:** Hierarchy



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.

5.1.1: Object-Oriented Principles

Concept of Abstraction

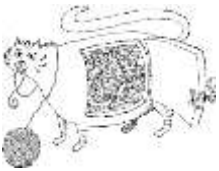
- Focus only on the essentials, and only on those aspects needed in the given context.
  - **For example:** Customer needs to know what is the interest he is earning; and may not need to know how the bank is calculating this interest
  - **For example:** Customer Height / Weight not needed for Banking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

5.1.2: Object-Oriented Principles

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.

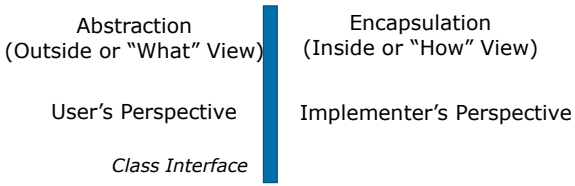


Encapsulation:
Every object is encapsulated in such a way, that its data and implementations of behaviors are not visible to another object. Encapsulation allows restriction of access of internal data. Encapsulation is often referred to as information hiding. However, although the two terms are often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

5.1: Object-Oriented Principles

Encapsulation versus Abstraction

- Abstraction and Encapsulation are closely related.



- Why Abstraction and Encapsulation?
  - They result in "Less Complex" views of the System.
  - Effective separation of inside and outside views leads to more flexible and maintainable systems.

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.

5.1: Object-Oriented Principles

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.

5.1.3: Object-Oriented Principles

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.

5.1.3: Object-Oriented Principles

Concept of Modularity

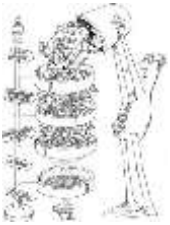
- Modularity in OO Systems is typically achieved with the help of components
  - A Component is a group of logically related classes
  - A Component is like a black box – users of the component need not know about the internals of a component

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 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.

5.1.4: Object-Oriented Principles

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.

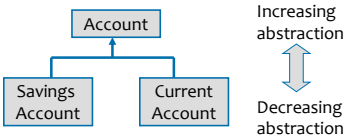


Hierarchy:
Hierarchy is the systematic organization of objects or classes in a specific sequence in accordance to their complexity and responsibility. In a class hierarchy, as we go up in the hierarchy, the abstraction increases. So all generic attributes and operations pertaining to an Account are in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

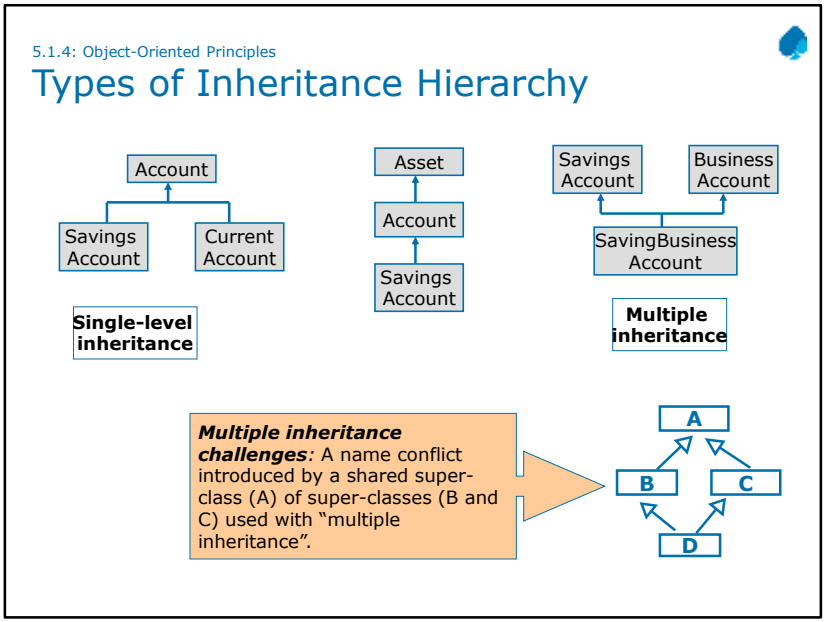
5.1.4: Object-Oriented Principles

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.

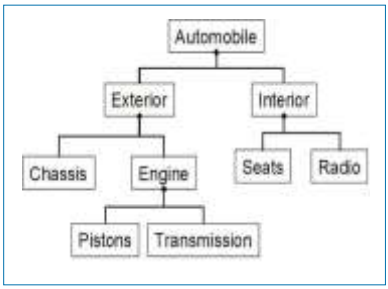


Types of Inheritance Hierarchy:
Single-level inheritance: It is when a sub-class is derived simply from its parent class.
Multilevel Inheritance: It is when a sub-class is derived from a derived class. Here a class inherits from more than one immediate super-class. Multilevel inheritance can go up to 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.

5.1.4: Object-Oriented Principles

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.

5.1.4: Object-Oriented Principles

A glance at relationships

- The Inheritance or "Is A" Hierarchy leads to Generalization relationship amongst the classes.
- The Object Hierarchy or "Has A" relationship leads to Containment relationship amongst the objects
  - Aggregation and Composition are two forms of containment amongst objects
  - Aggregation is a loosely bound containment. E.g. Library and Books, Department and Employees
  - Composition is tightly bound containment. E.g. Book and Pages

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 (E.g. Body & parts like Heart, Brain..), the objects are said to be in a composition relationship.

5.1.4: Object-Oriented Principles

A glance at relationships

- Most commonly found relationship between classes is Association
  - Association is the simplest relationship between two classes
  - Association implies that an object of one class can access public members of an object of the other class to which it is associated

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.

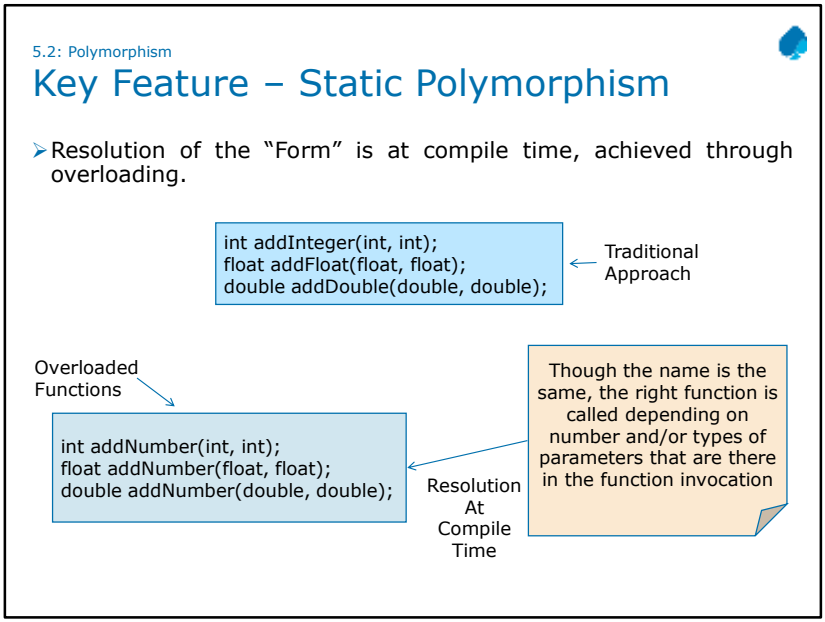
5.2: Polymorphism

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

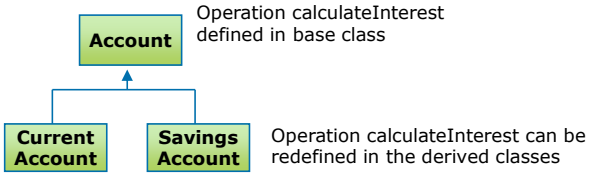


Polymorphism (contd.):
Overloading is when functions having same name but different parameters (types or number of parameters) are written in the code. When Multiple Sort operations are written, each having different parameter types, the right function is called based on the parameter type used to invoke the operation in the code. This can be resolved at compile time itself since the type of parameter is known.

5.2: Polymorphism

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.

5.2: Polymorphism

Key Feature – Polymorphism

- Why Polymorphism?
  - It provides flexibility in extending the application.
  - It results in more compact designs and code.

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:

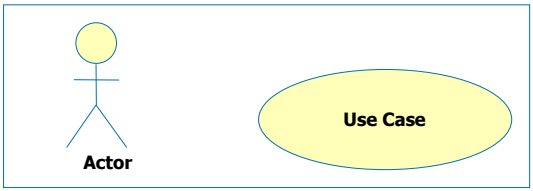
```
IF Account is of CurrentAccountType THEN
        calculateInterest_CurrentAccount()
IF Account is of SavingsAccountType THEN
        calculateInterest_SavingAccount()
```

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.

5.3.1: Use Case Diagrams

Use Case Diagrams - Features

- Use Case Diagrams model the functionality of a system by
- using Actors and Use Cases:
  - Actor is a user of the system.
  - Use cases are services or functions provided by the system to its users.



Use Case Diagrams:
Use Case is a description of a system's behavior from a user's point of view. It is a set of scenarios that describe an interaction between a user and a system. It also displays the relationship among Actors and Use Cases.
Two main components of Use Case diagram are Use Cases and Actors.

Use case diagrams, which render the User View of the system, describe the functionality (Use Cases) provided by the system to its users (Actors).
An Actor represents a user or another system that will interact with the system you are modeling.

An Use Case is an external view of a system that represents some action that the user might perform in order to complete a task.

5.3.1: Use Case Diagrams

Definition of Actor

- Actor:
  - An Actor can be defined as follows:
    - Actor is any entity that is external to the system and directly interacts with the system, thus deriving some benefit from the interaction.
    - Actor can be a human being, a machine, or a software.
    - Actor is a role that a particular user plays while interacting with the system.
    - Examples of Actors are End-user (roles), External systems, and External passive objects (entities).

Actor:
Actors are people, organizations, systems, or devices which use or interact with our system. The system exists to support that interaction. Therefore, the important part of the project is to identify the Actors and find out what they want from the system. Actors are characterized by their external view rather than their internal structures. It is a role that the user plays to get something from the system.
Role and organization Actors only require logical interactions with the system. Ask who wants what from our system, rather than who operates the system.

For example: ABC and XYZ are users who wish to buy from an online store. For the online stores system, they play the role of a customer, and hence customer is the Actor for the system. The database for this system may already be existing, and hence this may be another Actor (note that user in this case is not a human).
The Actors will finally be used to describe classes, which will interact with other classes of the system.

5.3.1: Use Case Diagrams

Definition of Use Cases

- Use Case:
  - An Use Case can be defined as a set of activities performed within a system by a User.
  - Each Use Case:
    - describes one logical interaction between the Actor and the system.
    - defines what has changed by the interaction.

Use Cases:
The Use Cases define “units of functionality” provided by system. They model “work units” that the system provides to its outside world.
A Use Case is one usage of the system. It is a generic description of a use of the system. It allows interactions in a specific sequence.
At the lowest level, they are nothing but methods which need to be implemented by various classes in the system.
Use Cases determines everything that the Actor wants to do with the system.
A Use Case performs the following functions:

Defines main tasks of the system
Reads, writes, and changes system information
Informs the system of real world changes

A Use Case needs to be updated / informed about system changes.

5.3.1: Use Case Diagrams

Drawing the Use Case Diagram

- A Use Case diagram has the following elements:
  - **Stick figure**: It represents an Actor.
  - **Oval**: It represents a Use Case.
  - **Association lines**: It represents communication between Actors and Use Cases.

Drawing the Use Case Diagram:
The Use Case Diagram has the following elements:

A stick figure, which represents Actors (sometimes stereotyped classes, as explained later, are also used to represent Actors). They differ from tool to tool.
Ovals or ellipses, which represent Use Cases
Association lines, which indicate interactions between Actors and Use Cases.

Use Cases will have description of what the Use Case is supposed to do when it is used.
An example of use case description is given.

5.3.1: Use Case Diagrams

Use Case Relationships - Overview

- Types of relationships between Use Cases are:
  - Include
  - Extend

Use Case Relationships:
Relationships help us connect the model elements.
After finding out the primary Use Cases, one can start looking "into" the system to see if there are any relationships between the Use Cases.
The following types of relationships can exist between the Use Cases:

include
extend

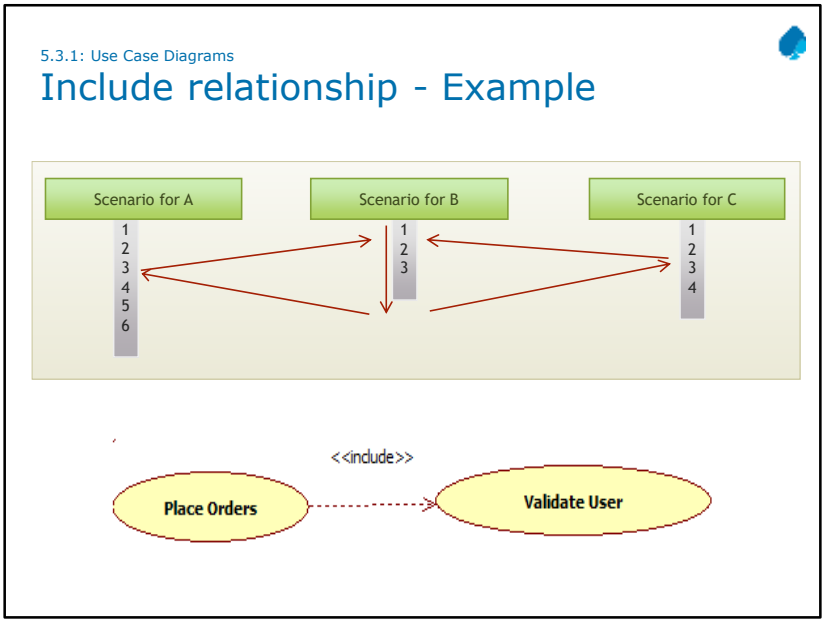
5.3.1: Use Case Diagrams

Include relationship - Characteristics

- Include relationship:
  - «include» stereotype indicates that one use case "includes" the contents of another use case.
  - Include relationship enables factoring out frequent, common behavior.

- Use case "A" includes use case "B", if:
  - B describes scenario which is part of scenario of A, and
  - B describes scenario common for a set of Use Cases including A.

Use Case Relationship – Include:
In an Include relationship, one Use Case includes behavior specified by another Use Case. If there are common steps in the scenarios of many Use Cases, they can be factored out into a separate Use Case. This Use Case can then be included as part of the "Primary Use Case".
The above arrangement helps us segregate and organize common sub-tasks. An "Included Use case" is not a complete process. Extra behavior is added to the base Use Case.
This may also be used in case of complex Use Cases (where there is too much functionality in one Use Case). In such cases, the primary functionality can be distributed across Use Cases, and a primary Use Case can then include the remaining secondary Use Cases.



Use Case Relationships – Include (contd.):
As illustrated in the figure shown in the slide, scenario of Use Case B is required by Use Case A and Use Case C. Hence both Use Cases A and C can include Use Case B.
After completing the scenario of Included Use Case B, the Use Cases A and C will continue with their respective scenarios.

5.3.1 : Use Case Diagrams

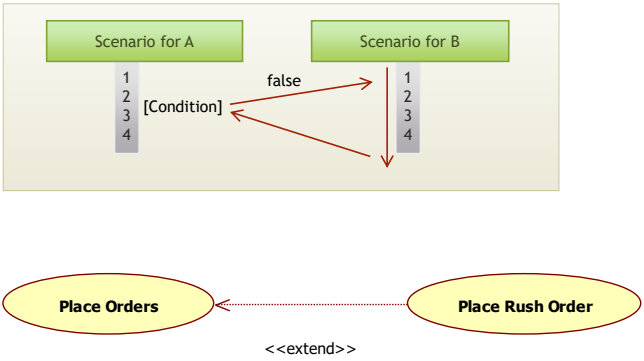
Extend relationship - Characteristics

- Extend relationship:
  - «extend» stereotype indicates that one Use Case is "extended" by another Use Case.
  - Extend relationship enables factoring out infrequent behavior or error conditions.
  - Extend relationship represents optional behavior for a Use Case which will be required only under certain conditions.

Use Case Relationships - Extend :
In an Extend relationship, an Use Case may be required by another use case based on "some condition", or due to "some exceptional situation".
Again, upon completion of extension activity sequence, the original Use Case will continue.

5.3.1 : Use Case Diagrams

Extend relationship - Example

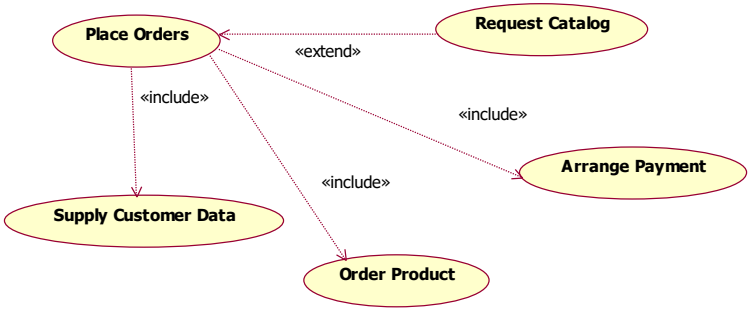


Use Case Relationships – Exclude (contd.):
As illustrated in the figure shown in the slide, in Use Case A when the condition becomes false, the scenario of Use Case B is invoked.
Note that Use Case B is said to extend Use Case A. The stereotyped generalization arrow with keyword “extend” depicts the extend relationship between the Use Cases.

5.3.1 : Use Case Diagrams

Examples of Use Case Relationships

- Example 1:



Example of Use Case Diagram:
The slide shows an example where we are looking at the Use Case relationships (though the Actors and system boundary has not been shown here, let us assume that they exist. They have been left out so as to focus on the relationships).
The Request for Catalog may not always happen when an Order needs to be placed. Hence, Extend is the relationship used between the two Use Cases of “Place Order” and “Request Catalog”.
“Place Order” being a complex Use Case, it is broken down into secondary use cases of:

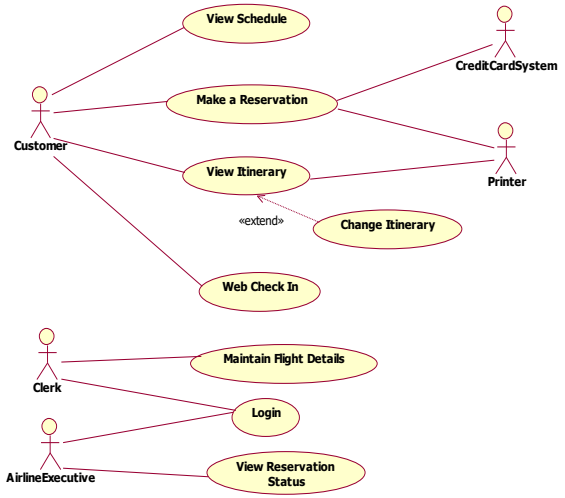
- Supply Customer data
- Order Product
- Arrange Payment

It is important to note that the diagram by itself does not indicate any kind of ordering (i.e. first invoke order product, and then arrange payment, etc.). The ordering has to be taken care of as part of the Use Case scenario.

5.3.1 : Use Case Diagrams

Examples of Use Case Relationships (Cont.…)

Example 2:



How do you interpret this diagram?

5.3.2 : Sequence Diagrams

Features

- Sequence Diagram:
  - A Sequence diagram describes interactions among classes in terms of an "exchange of messages over time".
  - Some of the facts related to Sequence Diagrams are:
    - Sequence Diagrams are used to depict the time sequence of messages exchanged between objects.
    - Messages can correspond to operation on class or a event trigger.

Sequence Diagrams: Features:
Starting from the scenarios of the Use Case, interactions between objects in the scenario can be depicted in the Sequence Diagram. These interactions are in the "sequence of time", and finally correspond to "operations" or "event triggers".
Only the sequences of messages, with respect to time, are important. There is no particular span of time that is considered.
Sequence Diagrams show objects, and not classes. This is different from Class Diagrams, which contains classes that signify the existence of objects.
Sequence Diagrams provide a better correlation between the "dynamic view" of the system and the "static view" held in the Class Diagram.

5.3.2 : Sequence Diagrams

Notations

- Notations of a Sequence Diagram include:
  - **LifeLine**: It is a vertical dashed line that represents the "lifetime" of an object.
  - **Arrows**: They indicate flow of messages between objects.
  - **Activation**: It is a thin rectangle showing period of time, during which an object is performing an action.

Sequence Diagrams: Notations:

In the Sequence Diagram, different objects / class roles are laid out horizontally at the top. There is no significance to the ordering of these objects.

The Lifeline, denoted as dashed line beneath the class role, is used to model "existence of entities over time".

Activation, shown as thin rectangles along the lifeline, model the time during which entities are active.

Activation may not always be depicted.

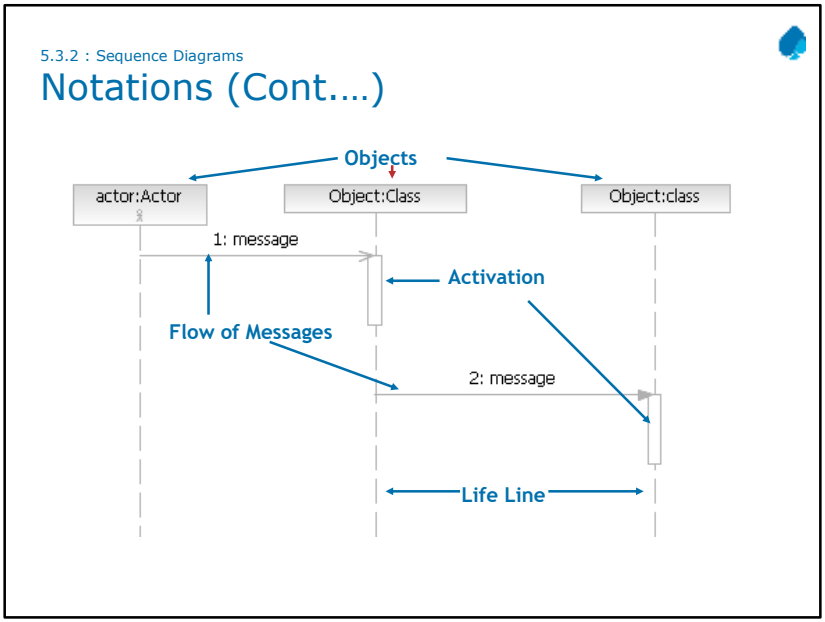
Arrows indicate flow of messages between objects.

Messages are denoted as horizontal arrows between lifelines.

Messages may be:

as simple as a "string" conveying an operation, or

as detailed as a "method", which includes parameters passed and values returned.

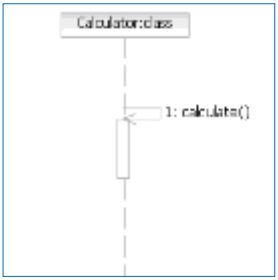


The slide shows the notations used in a Sequence Diagram.

5.3.2 : Sequence Diagrams

Direction of Arrows

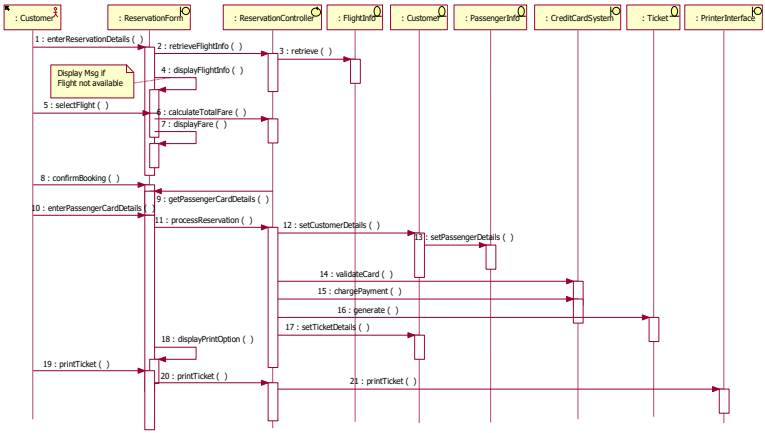
- Direction of Arrows:
  - Direction indicates which object's method is being called by whom.
  - A circulating arrow on the Object Lifeline is for a self method - called within the object by itself.



Sequence Diagram: Notations (Directions and Branches):
Direction of arrows:
Every message has a "sender" and "receiver" and this is depicted by the direction of the arrow head. Control gets passed from the sender to receiver.
"Implicit returns" are assumed in case there are no return messages to the sender.
It is possible that there is a "call to object's own method". A "circulating arrow" is used to depict such a case. In the example shown on the slide, Calculator object calls its own method calculate().

5.3.2 : Sequence Diagrams

Example of Sequence Diagrams



How do you interpret this diagram?

5.3.3: Class Diagrams

Features

- Class Diagrams:
  - Class Diagrams define the basic building blocks of a model, namely:
    - types
    - classes, and
    - general material used to construct the full model

Class Diagrams: Features:
Class Diagrams can be used to model classes, and the relationships between classes.
When drawn during the analysis stage, only the names of the classes maybe represented.
During further refinements in the detailed analysis or design stage, details like “attributes” and “services” get added to each class, and are depicted in the Class Diagram.

5.3.3 : Class Diagrams

Functions

- Class Diagrams have the following functions:
  - They describe the static structure of a system.
  - They show the existence of classes and their relationships.
    - Classes represent an abstraction of entities with common characteristics.
    - Relationships may be:
      - Generalization
      - Association
      - Aggregation
      - Composition, or
      - Dependency

5.3.3 : Class Diagrams

Uses

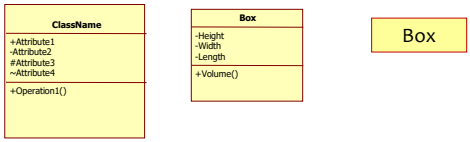
- Typical uses of Class Diagrams are:
  - To model vocabulary of the system, in terms of system's abstractions
  - To model collaborations between classes
  - To model logical database schema (blueprint for conceptual design of database)

Uses of Class Diagram:
The importance of the Class diagram is that it gives a view of all the classes that are required to make up the system. It also conveys the collaborations that exist between classes to give the system behavior.

5.3.3 : Class Diagrams

Notations for Class

- Class may be represented in any of the following ways
  - Only Class Name is mandatory



Notations for Class:
Classes are denoted as rectangles, with compartments for name, attributes, and operations. There is optionally a last compartment that can be used for specifying responsibilities, variations, business rules, etc.
The name is the mandatory part. Other compartments may be included based on the amount of details required to be communicated. The representations of classes that do not have all compartments are known as “elided notations for class”.

5.3.3 : Class Diagrams

Notations for Class (Cont....)

- Class Visibility signifies how information within class can be accessed.

Symbol	Meaning
+	Public
_	Private
#	Protected

Notations for Class: Class Visibility:
Information about visibility of attributes and operations can sometimes be represented by using symbols like + for public, - for private, or # for protected.
These symbols may vary from tool to tool.

5.3.3 : Class Diagrams

Association Relationship - Features

- In Association:
  - Name indicates relationship between classes.
  - Role represents the way classes see each other.

Relationships: Association:

Associations may be characterized by the following:

Name: The name signifies purpose of association, and is written along with the line indicating association, role and direction of association.

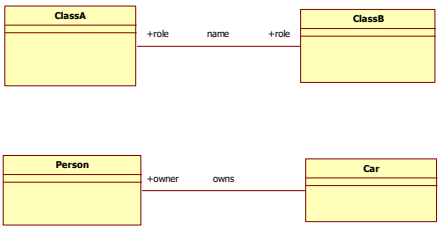
Role: In case there are specific roles played by classes in the association, then it is indicated by the role name, which is written near the class.

Arrow: Arrows may be used to indicate whether the association is uni-directional or bi-directional. Absence of arrows implies that no inferences can be drawn about the navigability.

The example in the slide shows an association relationship between a class Person and a class Car. The class Person plays the role of an owner.

5.3.3 : Class Diagrams

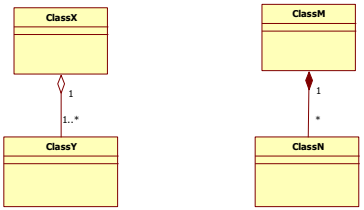
Association Relationship - Example



5.3.3 : Class Diagrams

Relationships - Features

- Aggregation and Composition:
  - The following Class Diagram, possessing Composition and Aggregation, displays:
    - Aggregation as indicated by a hollow diamond.
    - Composition as indicated by a filled diamond.
    - Diamond as pointing towards the "whole" class or the aggregate.

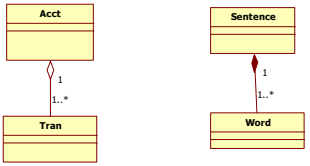


Relationships: Aggregation and Composition:
Composition and Aggregation are modeled with filled diamond and hollow diamond, respectively, on the "Whole" part.
Roles and multiplicity, if required, can be mentioned here, as well. Typically they are done for the "Part" part of the "Whole" part.

5.3.3 : Class Diagrams

Relationships - Examples

Aggregation **Composition**



Examples of Aggregation and Composition:
In the examples shown in the slide,

the relationship between a Sentence and a Word is represented as a “Composition” (Word is a part of a sentence).

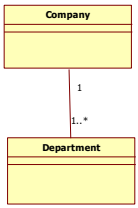
the relationship between Account and Transaction is represented as an “Aggregation”.

5.3.3 Class Diagrams

Definition of Multiplicity

- Multiplicity:
  - Multiplicity indicates the "number of instances" of one class linked to "one instance" of another class.

Symbol	Meaning
1	Exactly one
0..1	Zero or one
*	Many
0..*	Zero to many
1..*	One to many



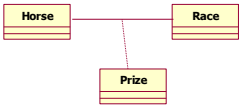
Multiplicity:
Multiplicity attached to a class denotes the possible cardinalities of objects of the association.

For example: The above figure depicts that "One company has one or more departments, and a department is associated with one company".
Multiplicity values can be indicated in association, aggregation, and composition relationships.

5.3.3 : Class Diagrams

Association Class Relationship - Features

- Association Class:
  - An Association Class is a class that has properties of both an "association" and a "class".
  - It is required when properties result from unique combination of two classes.
  - For example:



Association class:
An Association class is a class required as the result of association between two classes.
For example:

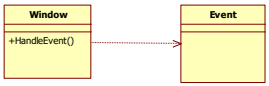
The Prize class is a result of association between the Horse class and the Race class.
For each Horse placed in a Race there is a prize.
The amount of prize depends on the race.
The Prize class could not be associated with the Horse class alone because a Horse might have many Prizes, and the relationship between the Prize and Race would be lost.
Similarly, Prize class cannot be associated with Race class alone because a Race has many Prizes, and the relationship between the Prize and the Horse would be lost.

Similarly, result of a student (in terms of marks in assignments, test, and grade) in a course is a unique combination of an individual student, and a particular course. So we can have an association between Student and Course Classes, with Result being an Association Class.

5.3.3 : Class Diagrams

Dependency - Features

- Dependency:
  - Dependency is a "using" relationship within which the change in the specification of one class may affect another class that uses it.
  - For example:

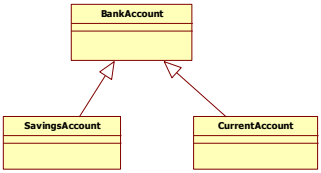


Dependency:
The dependencies are denoted as dashed arrows with arrow head pointing to the independent element.
In the example shown in the slide, the structure and behavior of the Window Class is dependent on the structure and behavior of the Event Class.

5.3.3 : Class Diagrams

Generalization - Features

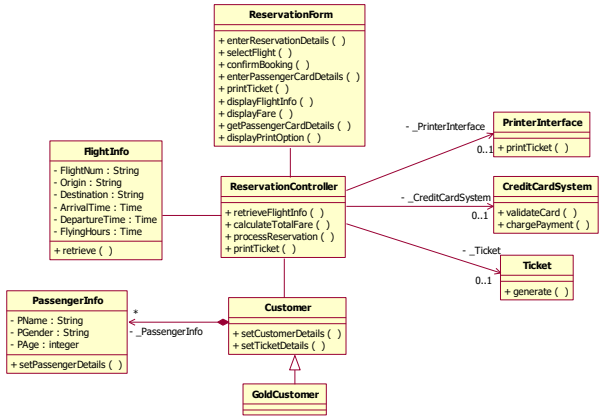
- Generalization:
  - Generalization indicates relationships between super-class and sub-class.
  - For example:



Relationships: Generalization:
Generalizations are denoted as “paths” from specific elements to generic elements, with a hollow triangle pointing to the more general elements.

5.3.3 : Class Diagrams

Example of Class Diagrams



How do you interpret this diagram?

5.3.4 Demos

Demo

- Demo on
  - Use case diagram
  - Sequence diagram
  - Class diagram

5.3.5 Case Study

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.

5.3.5 :Case Study

Scenario from Banking System

- Identify classes, attributes and methods
- Draw an use case diagram to identify the actors and their functionalities they perform.
- Draw the class diagram and identify correct relationship between the classes

Summary

- In this lesson, you have learnt about:
  - Principles in Object-Oriented technology
    - Abstraction
    - Encapsulation
    - Modularity
    - Hierarchy
    - Polymorphism
  - UML diagram
    - Use Case Diagram
    - Sequence Diagram
    - Class Diagram
  - Demo
  - Case study

Answers to Review Question

Question 1: Abstraction,Encapsulation,modularity,hierarchy

Question 2: True

Question 3:Has- a

Review Question

- Question 1: The 4 basic principles of Object Model are ___, ___, ___ and ___.
- Question 2: Function Overriding is kind of polymorphism.
  - True / False
- Question 3: ___ 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 4: Abstraction focuses on:
  - Option 1: implementation
  - Option 2: observable behavior
  - Option 3: object interface

- Question 5: Polymorphism can be achieved by:
  - Option 1: Hierarchy of Classes providing polymorphic behavior
  - Option 2: Interfaces
  - Option 3: Containment of Objects

  - Question 6: A message in a Sequence Diagram will help identifying ___.

Review Question

- Question 7: Use Case Diagrams represent the functionality needed in a system.
  - True / False
- Question 8: A Class Diagram gives information about:
  - A. Attributed defined for a class
  - B. Operations defined for a class
  - C. Logic to be used for an operation of a class
- Question 9: Relationships that you may find on a Class Diagram are ___, ___, ___, ___ and ___.