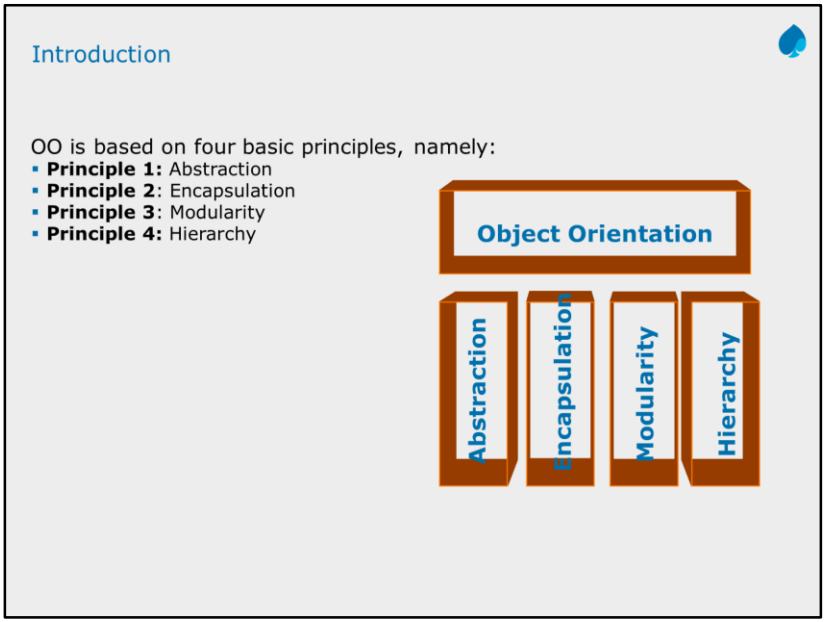

Object-Oriented
Programming
Lesson 3: Principles in
Object-Oriented
Technology
Capgemini

Lesson Objectives

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

- Abstraction
- Encapsulation
- Modularity
- Hierarchy and its types
  - Types of Inheritance Hierarchy
- Polymorphism and Types of Polymorphism



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.

3.1: 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.

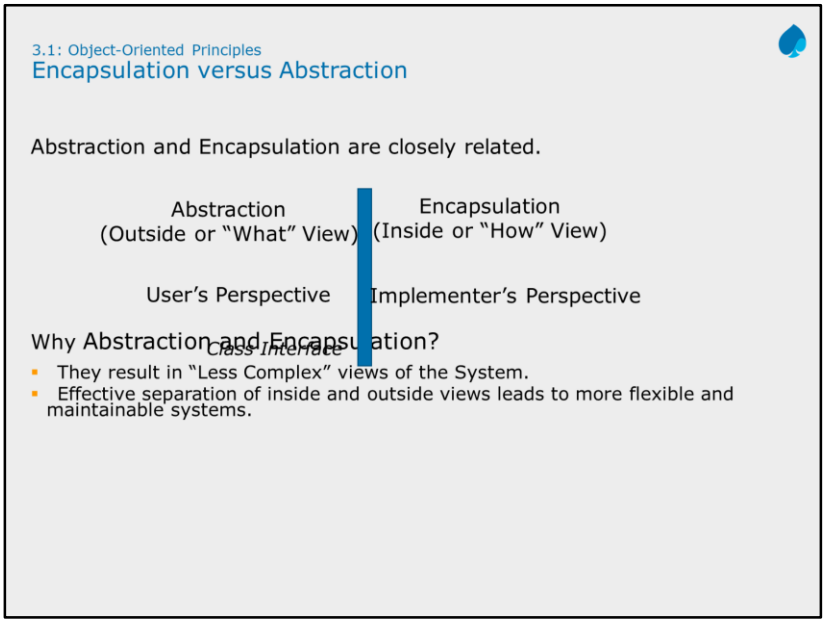
3.1: Object-Oriented Principles

Concept of Encapsulation

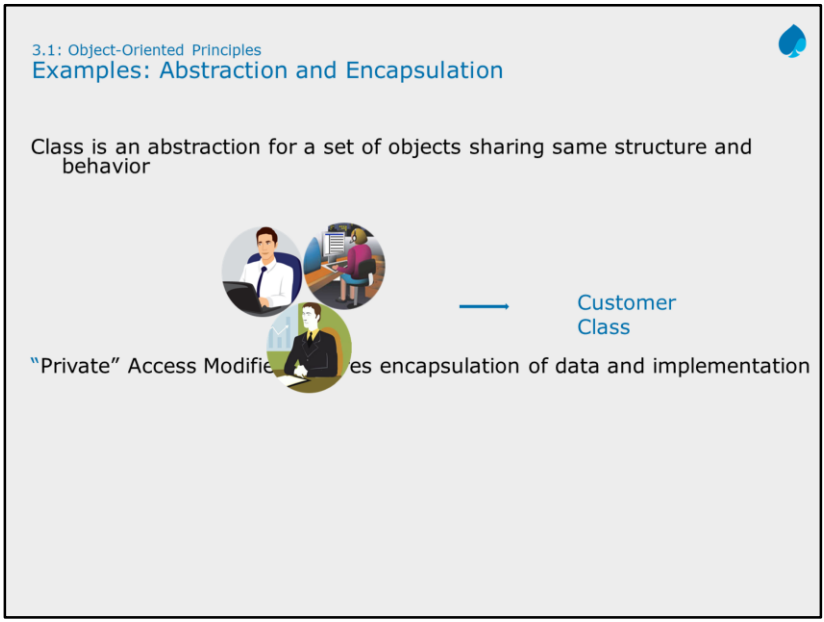
"To Hide" details of structure and implementation

- **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.



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.



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.

3.1: Object-Oriented Principles

Concept of Modularity

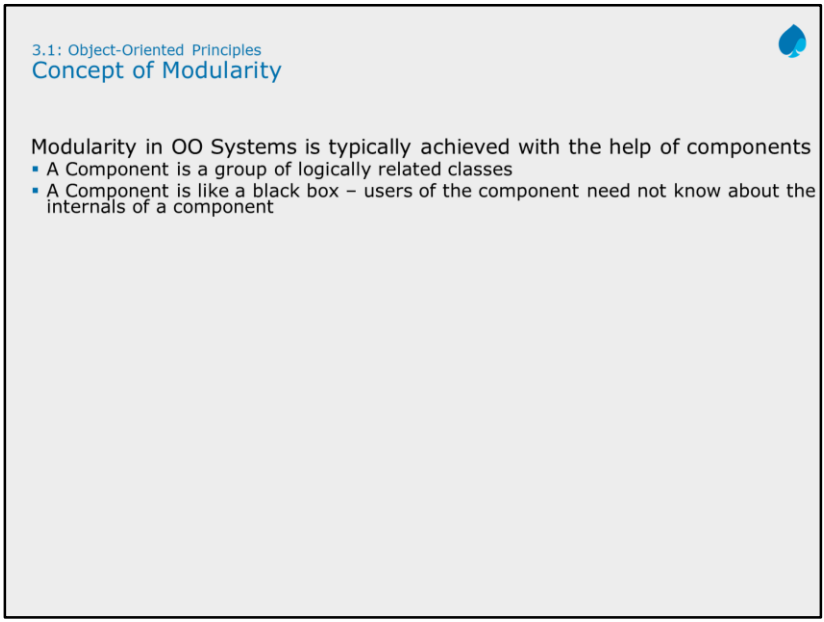
Decomposing a system into smaller, more manageable parts

- 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.



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.

3.1: Object-Oriented Principles

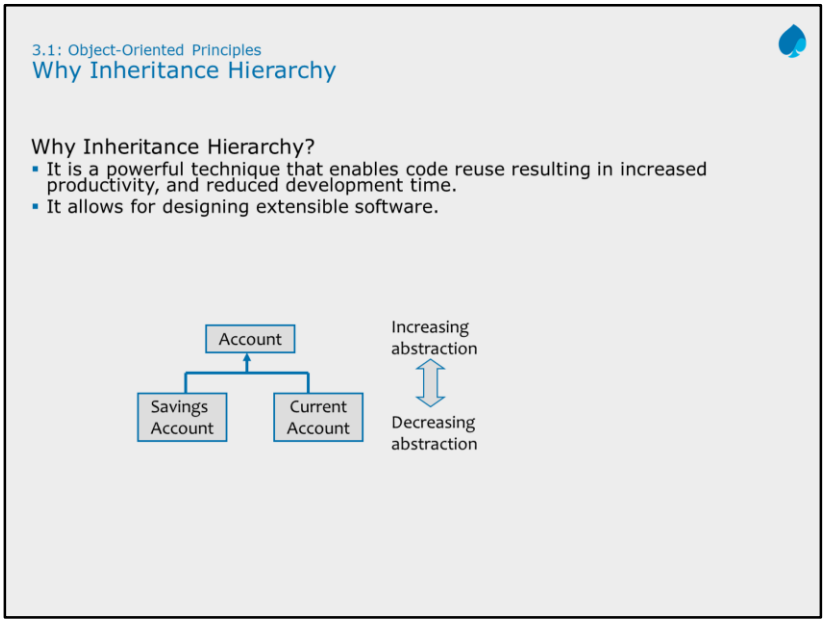
Concept of Hierarchy

A ranking or ordering of abstractions on the basis of their complexity and responsibility

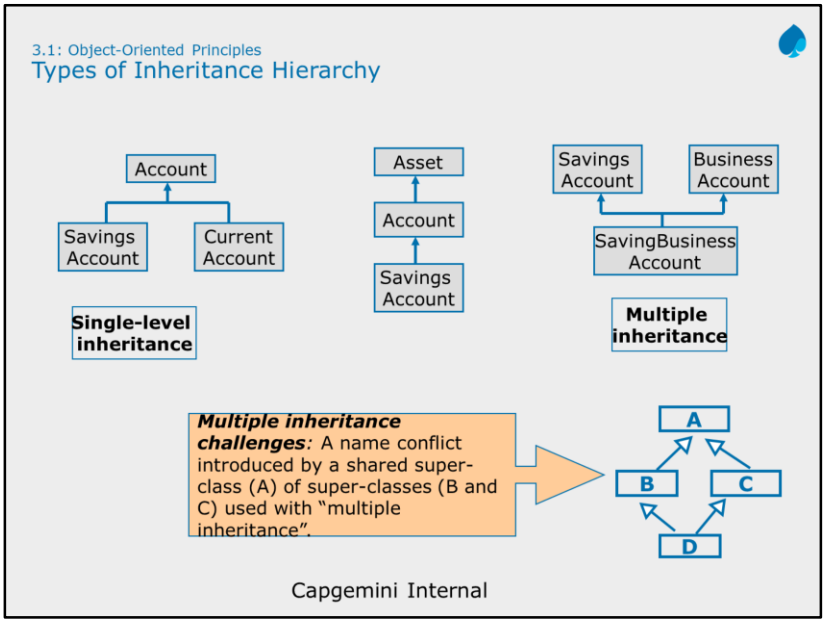
It is of two types:

- 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.



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.

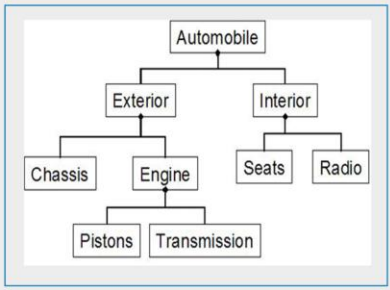


Types of Inheritance Hierarchy:
Single-level inheritance: It is when a sub-class is derived simply from its parent class.
Multilevel Inheritance: It is when a sub-class is derived from a derived class. Here a class inherits from more than one immediate super-class. Multilevel inheritance can go up to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

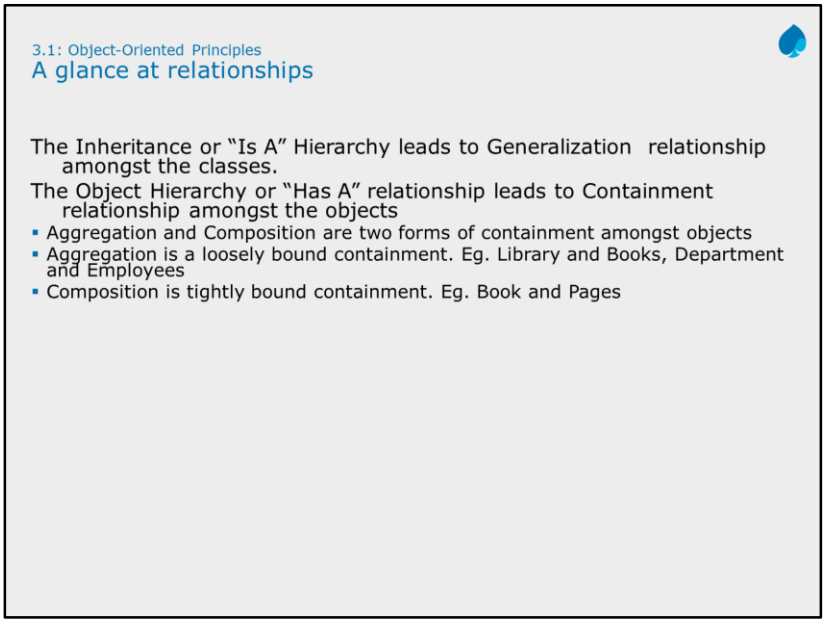
3.1: Object-Oriented Principles

Object Hierarchy

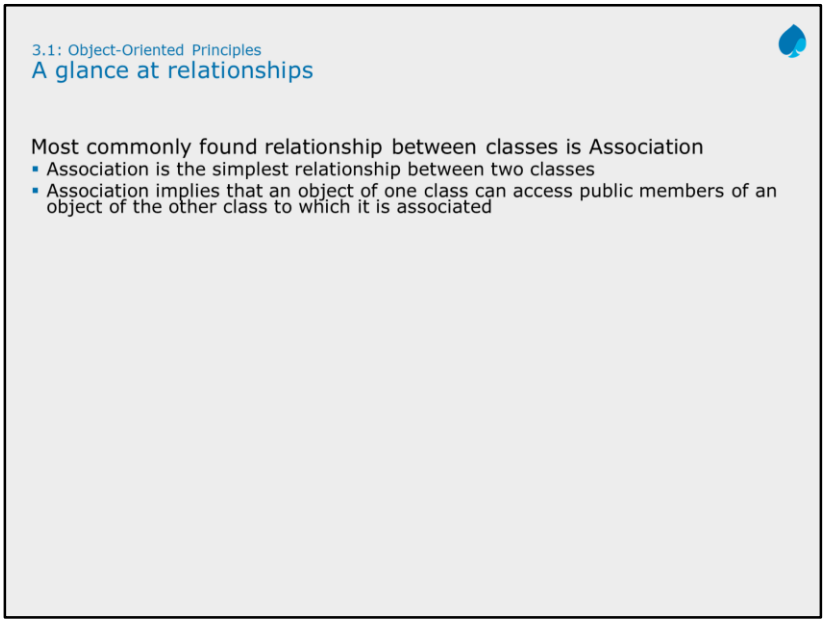
"Has-a" hierarchy is a relationship where one object "belongs" to (is a part or member of) another object, and behaves according to the rules of ownership.



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.



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.

3.2: Polymorphism

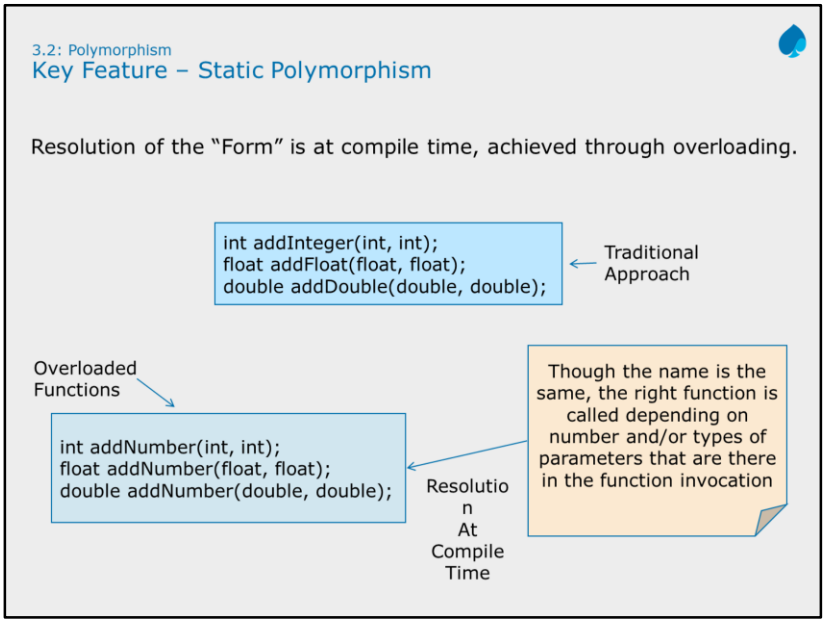
Key Feature – Polymorphism

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:

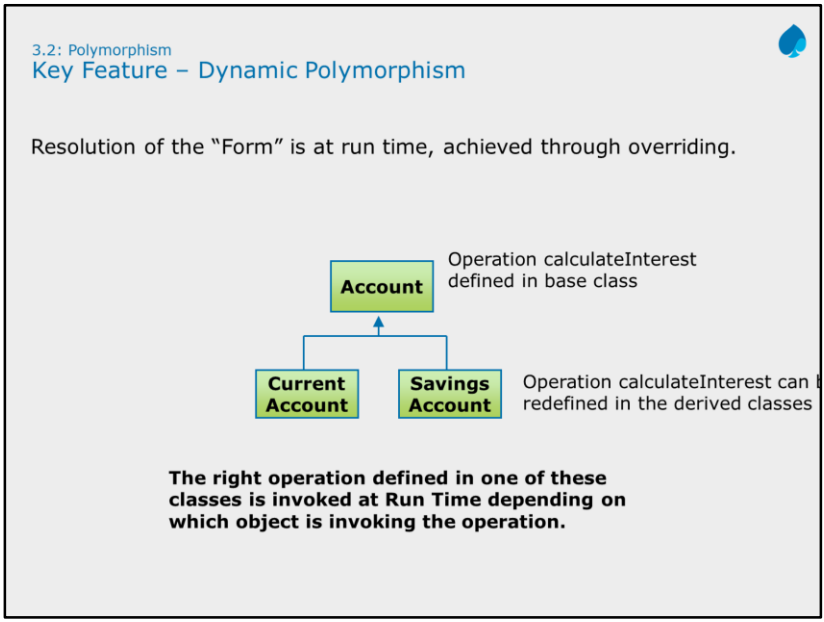
- Static Polymorphism
- Dynamic Polymorphism

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:

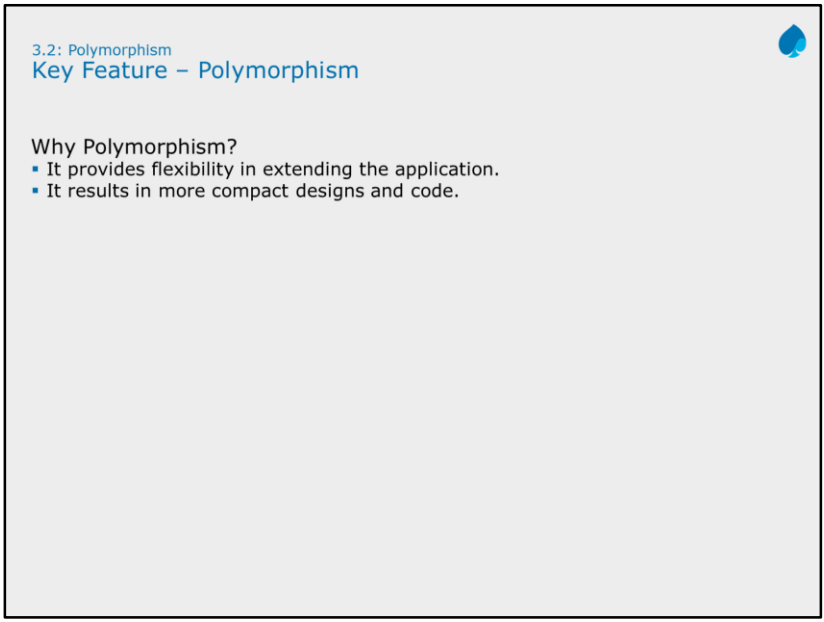
Static (or compile time) polymorphism, and
Dynamic (or run time) polymorphism



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.



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.

Summary

In this lesson, you have learnt that:

- Object-Orientated Programming is guided by four basic principles, namely:
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
- Polymorphism allows multiple implementations to be hidden behind a single interface.

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

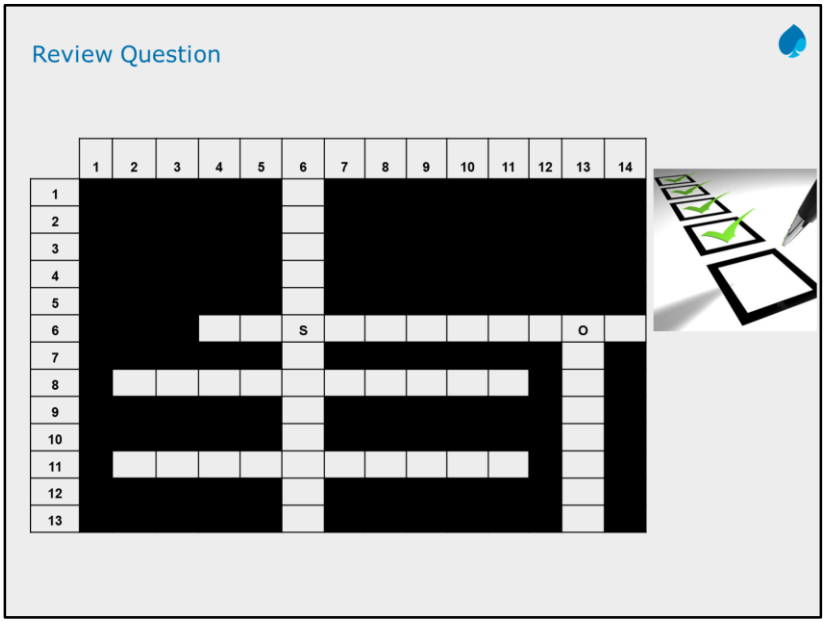
Question 4: Option2

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



Clues. 13 rows (1-10) , 14 Cols (1-14), (Row,Col) Combination
Across:
6-4 : Determining the essential qualities of an object (11)
8-2 : One of the basic advantages of OO (10)
11-2 : Type of inheritance where sub-class is derived from a derived class (10)

Below:
1-6 : Helps to restrict access to its internal data (13)
6-13 : Sub-class extends the existing version of a base class method (8)