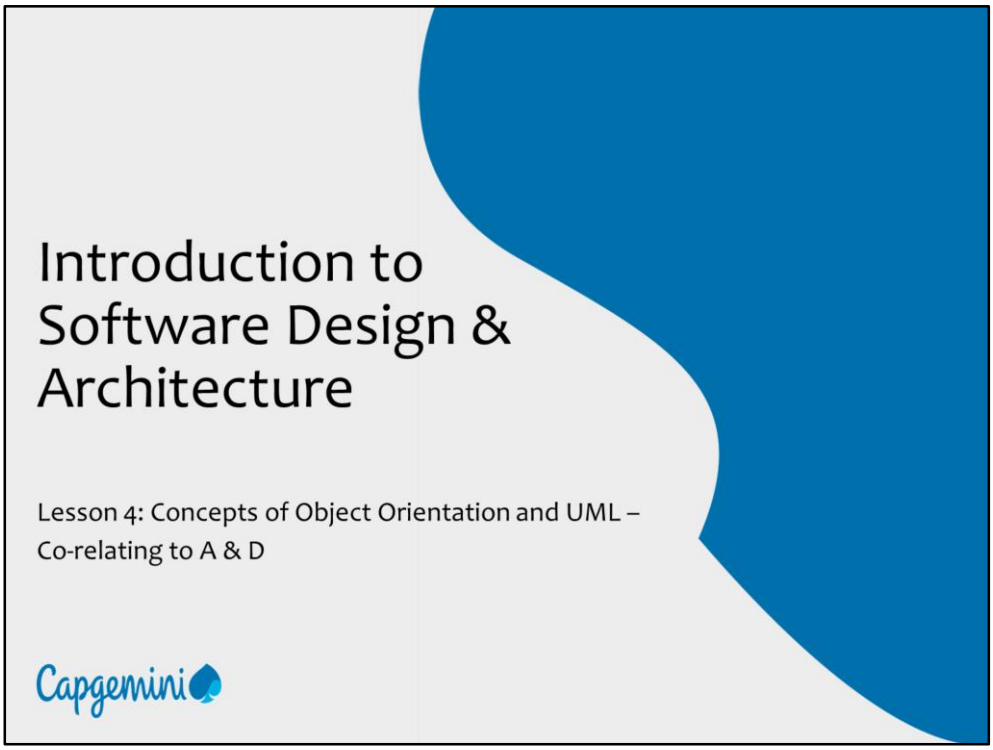
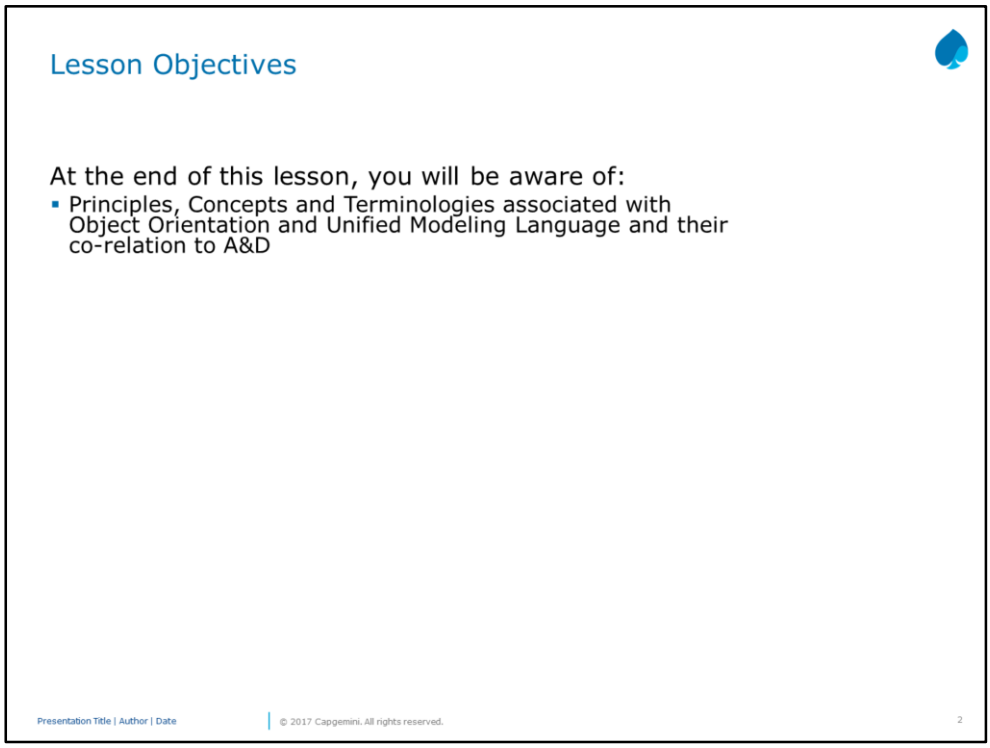
Instructor Notes:



Instructor Notes:

This lesson is to review concepts of OOP and UML, and while reviewing, co-relating it to A&D. This will be the foundation of everything else we do in this course, so it is essential that the participants are very clear about these concepts.

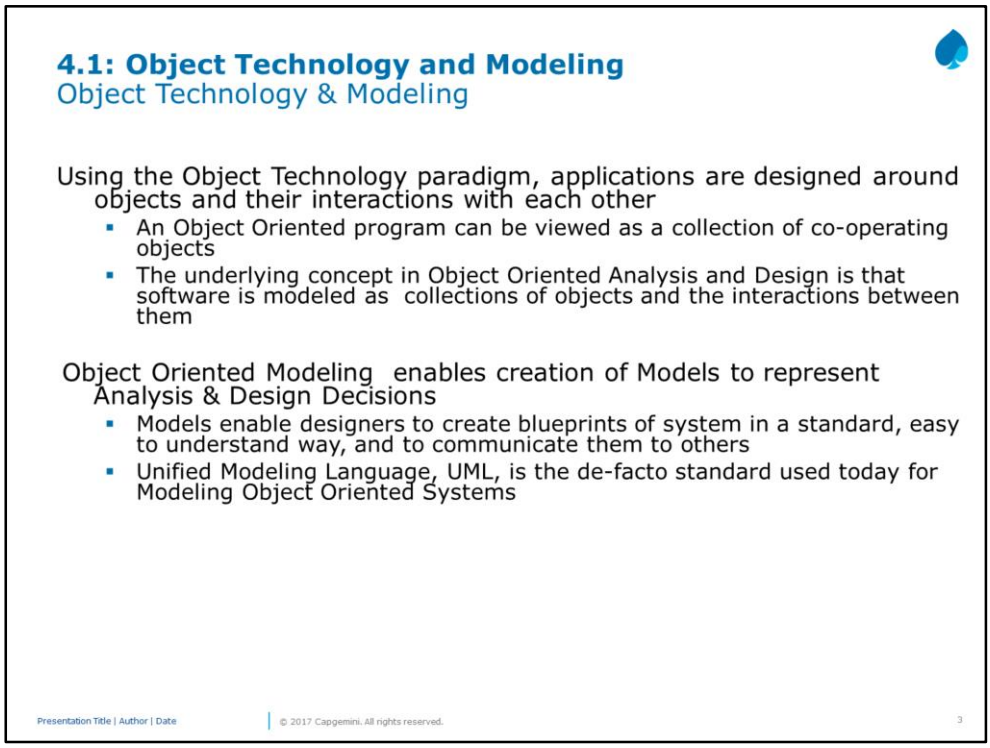


Lesson Objectives:

- In this lesson, we will review the principles, concepts, and terminologies associated with **Object-Oriented technology** and **Unified Modeling Language** and see how they co-relate to A&D.

Instructor Notes:

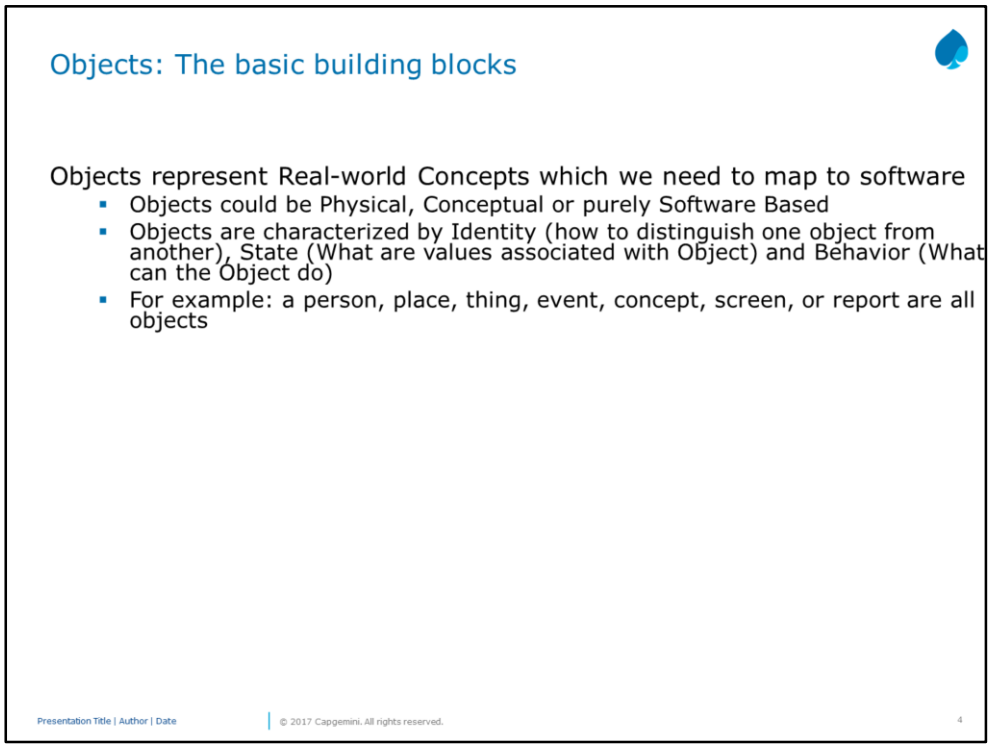
Get started with what it would mean when we talk about OOAD & UML: Objects as building blocks, Modeling as a way of capturing the blueprints



- In the OO world, we have objects that are the basic building blocks of an application. An OO program is made up of several objects that interact with each other to make up the application. OOAD constitutes of modeling the software as a collection of objects and their interactions.
- This modeling is done using UML, which is today an industry standard for modeling OO systems.

Instructor Notes:

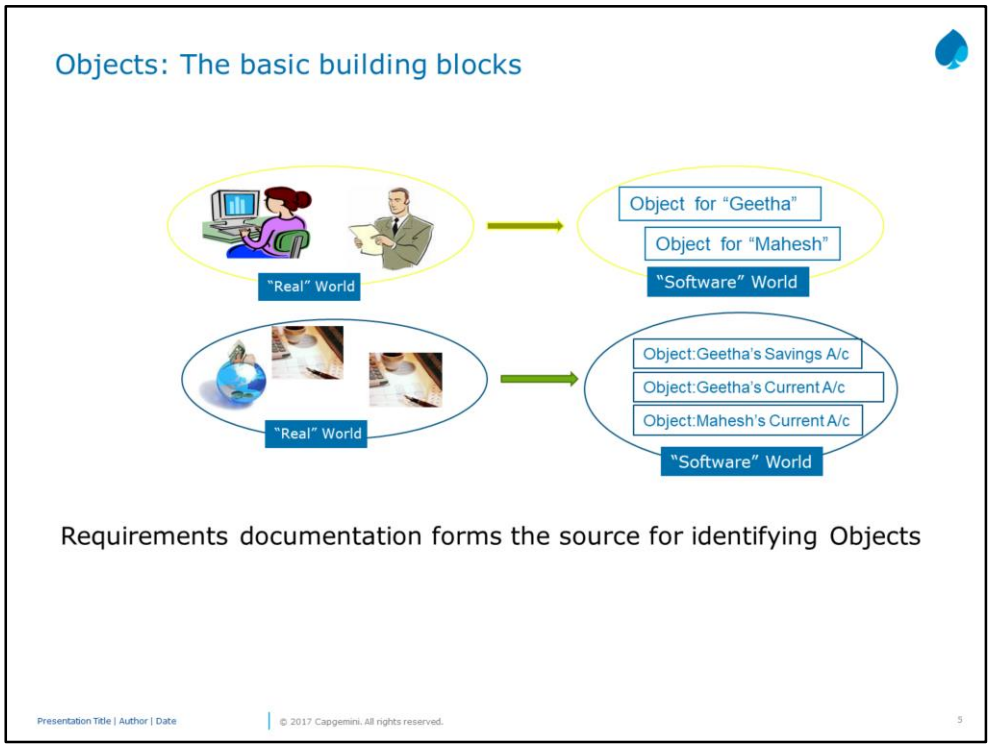
When we move from real to software world, we identify “objects” that exist in the real world to map them to software



- **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.

Instructor Notes:

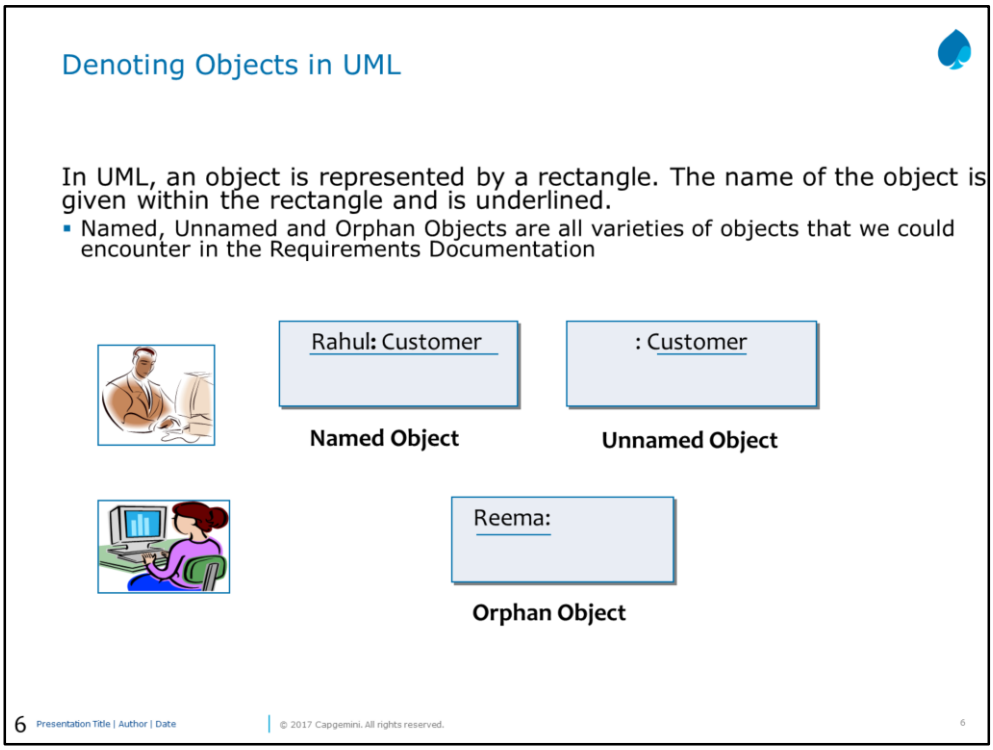
When we move from real to software world, we identify “objects” that exist in the real world to map them to software



- **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.

Instructor Notes:

We would typically use the Unnamed Object notation, unless we need to explicitly provide an object name.

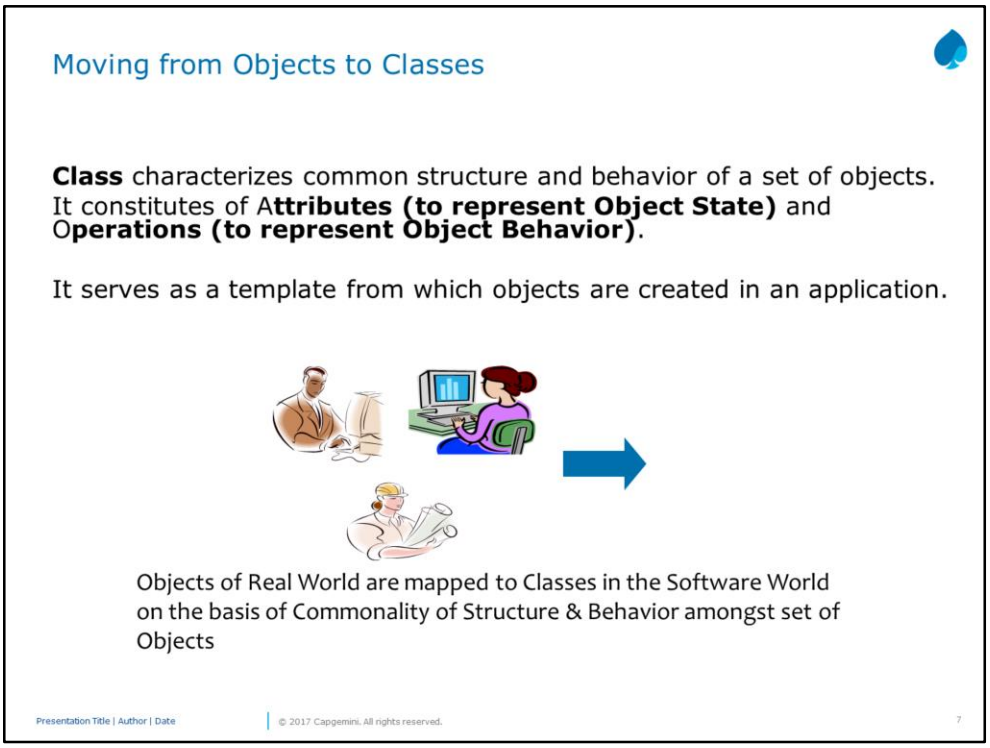


Objects and Classes: Denoting Objects in UML:

- To represent an object in the UML diagram:
  - Always represent objects within a rectangle.
  - Specify the name of the object within the rectangle and underline it.
  - Ensure that name of the object is followed by a colon and name of the class.
  - Omit the name before the colon and the class while creating an unnamed object.
- You can also have orphan objects. This is useful at an initial stage of analysis, when objects are found but we have not yet reached a stage where corresponding classes are determined.

Instructor Notes:

Important to emphasize how one moves from objects to classes while defining classes; similarly from classes to objects in the software world, when writing programs

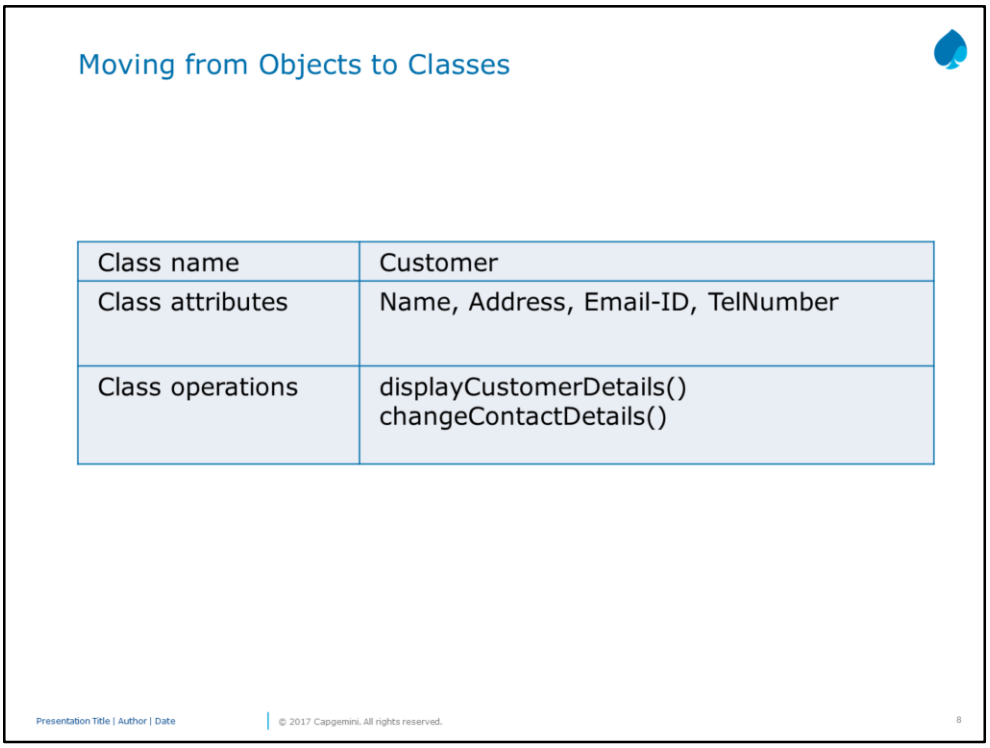


Object and Classes: What is a Class?

- 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? The entities that exists in real world are **objects**. When we classify these objects on the basis of commonality of structure and behavior, the result that we get are the **classes**.
- Classes are “logical”. They do not really exist in real world. The classes get defined first when writing software programs. These **classes** serve as a blueprint from which **objects** are created.

Instructor Notes:

Important to emphasize how one moves from objects to classes while defining classes; similarly from classes to objects in the software world, when writing programs



Object and Classes: What is a Class?

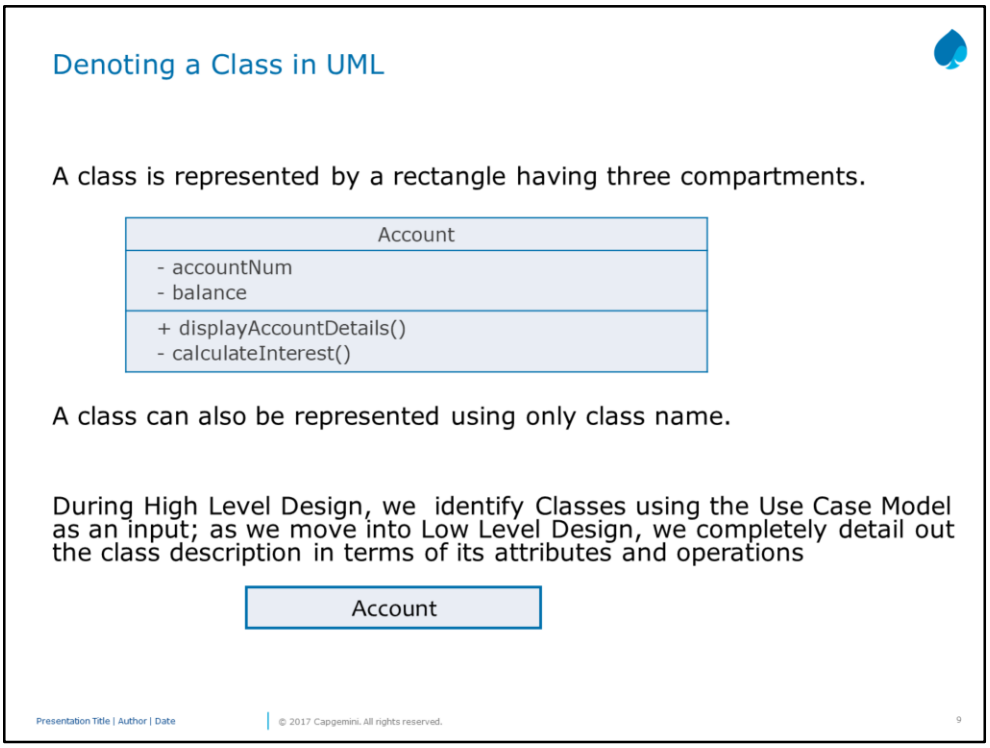
- Classes describe objects that share characteristics, methods, relationships, and semantics. Each class has a **name**, **attributes** (its values determine state of an object), and **operations** (which provides the behavior for the object).
- What is the relationship between objects and classes? The entities that exists in real world are **objects**. When we classify these objects on the basis of commonality of structure and behavior, the result that we get are the **classes**.
- Classes are “logical”. They do not really exist in real world. The classes get defined first when writing software programs. These **classes** serve as a blueprint from which **objects** are created.

Instructor Notes:

Discuss various ways in which classes can be represented.

UML notation allows for a 4th compartment for class responsibilities. Generally when outlining classes, one would begin with class name, and then move on to class responsibilities (4th compartment). Each responsibility maps to one or more operations (3rd compartment). Properties needed for these responsibilities/ operations are captured as class attributes (2nd compartment).

Even if all 3 compartments are not used, important to maintain the position as per UML notation.



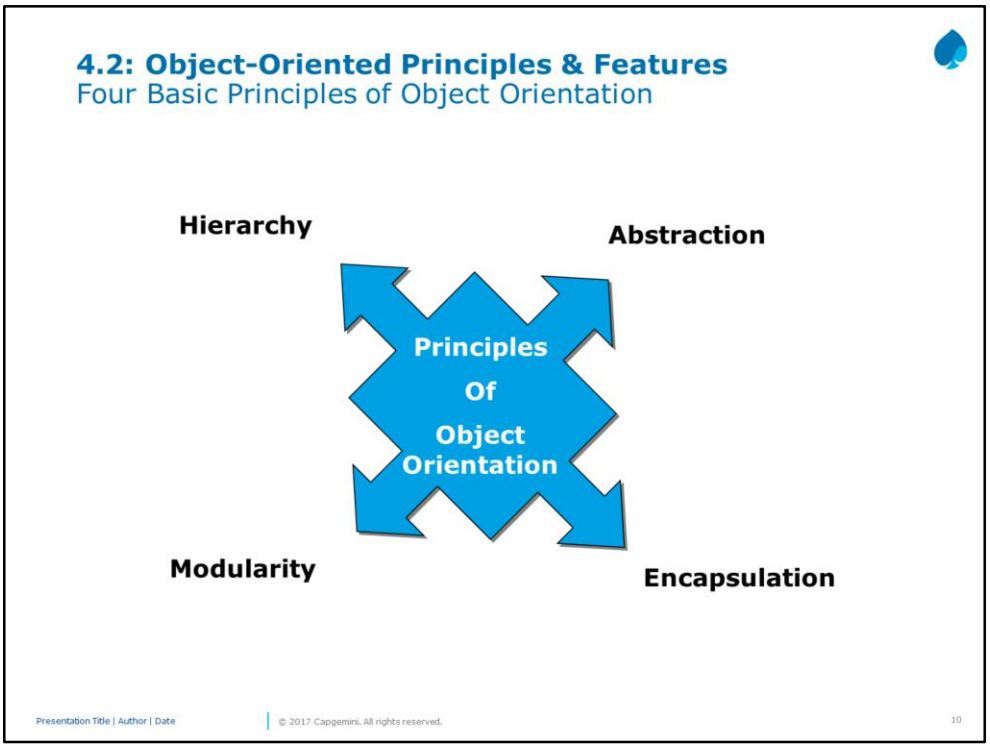
Objects and Classes: Denoting a Class in UML:

- A class is represented as a rectangle with three compartments.
    1. The top compartment lists the name of the class.
    2. The middle compartment lists attributes of the class.
    3. The third compartment lists the operations (behaviors) of the class.
- Further details can be provided for the class. For example, the Access Modifiers that specify how members of a class will be accessible. The UML notations are: **+** for Public, **-** for Private**,** and # for Protected.
- Other details that you can include are as follows:
    - The data types and default values, if any, for attributes
    - The complete signature for operations
- We shall see these notations in detail later on.
- One can also represent a class by a single rectangle indicating the class name.

Instructor Notes:

There is a tendency to include Inheritance, but it should technically be Hierarchy since Inheritance is restricted to classes while Hierarchy applies to both classes and objects.

Some mention Polymorphism too, but that would not be a basic principle. It is infact derived from these basic principles.

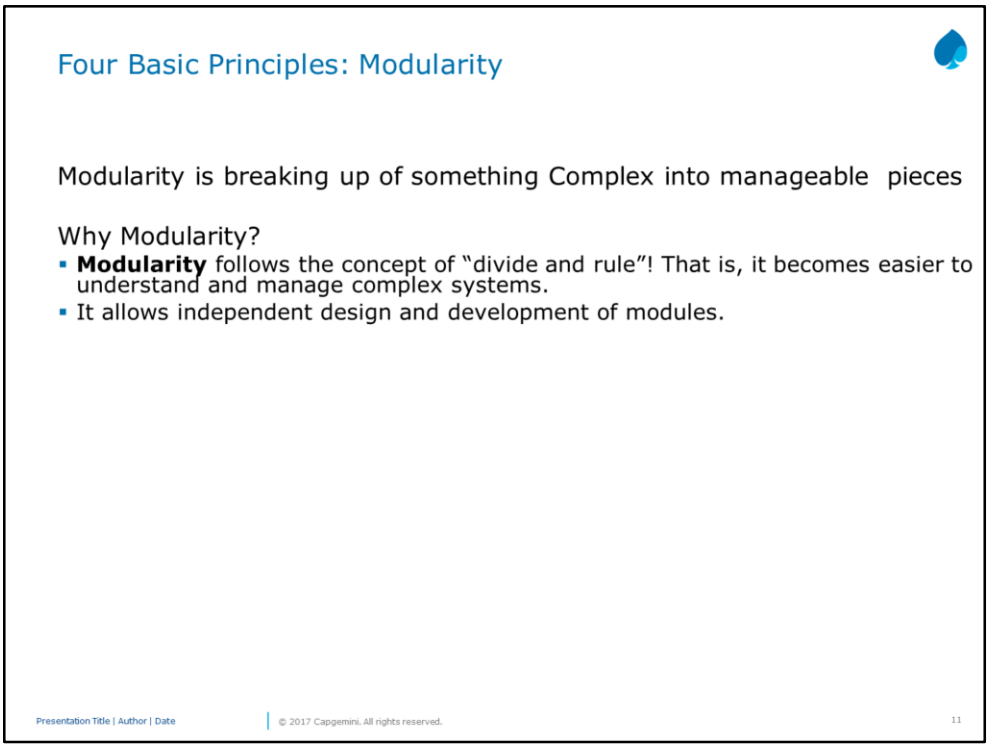


Object-Oriented Principles:

- The four basic principles guiding object orientation are as follows:
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
- Each of these are discussed in the following slides.

Instructor Notes:

Talk about modularity being achieved by the use of Layers/ Packages / subsystems in the OO World

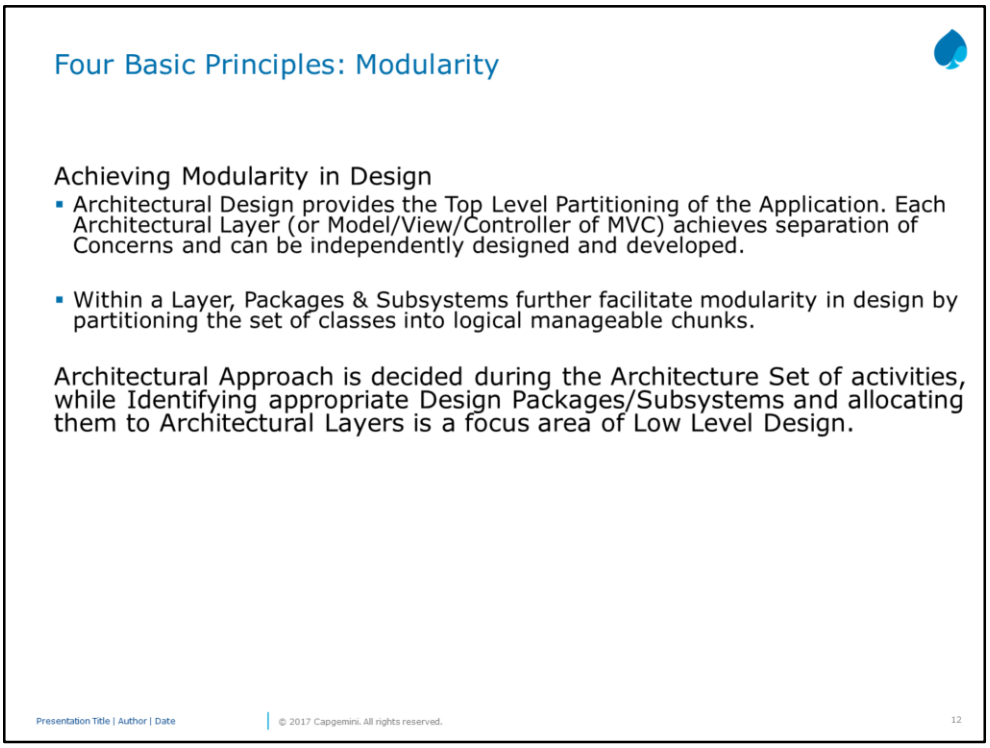


Object-Oriented Principles: Modularity:

- Modern day applications are quite complex, and modularity gives a mechanism to handle that complexity by looking at the application in smaller units. **Modularity** is obtained through decomposition, that is, breaking up complex entities into manageable pieces. Each of these units can then be independently designed and developed.
- Architecture provides the application foundation in the form of high level organization of the system – through partitioning the system on the basis of separating out the various concerns of the application.
- Each layer would have several classes and interfaces. Managing these classes is a challenging issue. Maintenance becomes easier by placing these model elements (classes and the interfaces) into **logical groups** called **packages**.
- A **subsystem** is a model element that has the semantics of both the following:
  - A **package**, such that it can contain other model elements
  - A **class**, such that it has behavior (The behavior of the subsystem is provided by classes or other subsystems it contains.)
- A **subsystem** realizes one or more interfaces, which define the behavior it can perform.

Instructor Notes:

Talk about modularity being achieved by the use of Layers/ Packages / subsystems in the OO World



Object-Oriented Principles: Modularity:

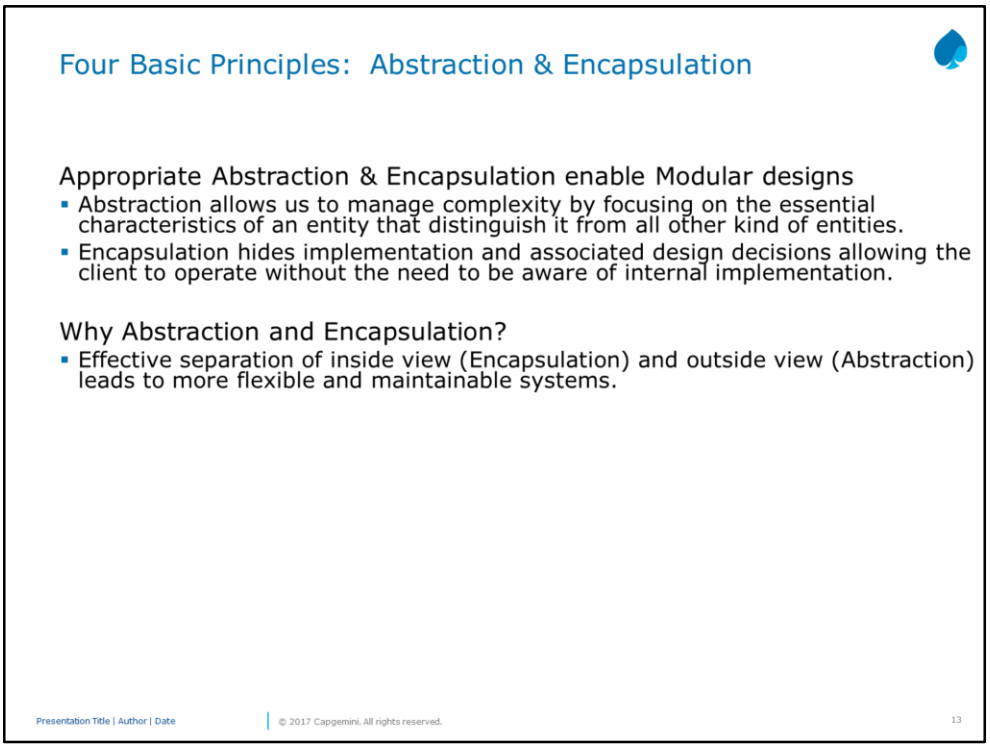
- Modern day applications are quite complex, and modularity gives a mechanism to handle that complexity by looking at the application in smaller units. **Modularity** is obtained through decomposition, that is, breaking up complex entities into manageable pieces. Each of these units can then be independently designed and developed.
- Architecture provides the application foundation in the form of high level organization of the system – through partitioning the system on the basis of separating out the various concerns of the application.
- Each layer would have several classes and interfaces. Managing these classes is a challenging issue. Maintenance becomes easier by placing these model elements (classes and the interfaces) into **logical groups** called **packages**.
- A **subsystem** is a model element that has the semantics of both the following:
    - A **package**, such that it can contain other model elements
    - A **class**, such that it has behavior (The behavior of the subsystem is provided by classes or other subsystems it contains.)
- A **subsystem** realizes one or more interfaces, which define the behavior it can perform.

Instructor Notes:

Explain that Abstraction is the "What" part, one that helps give the outside view of the system

Encapsulation is the How part giving the Internal View of the System

As a designer, we first abstract and then bother about corresponding encapsulation.



Object-Oriented Principles: Abstraction & Encapsulation

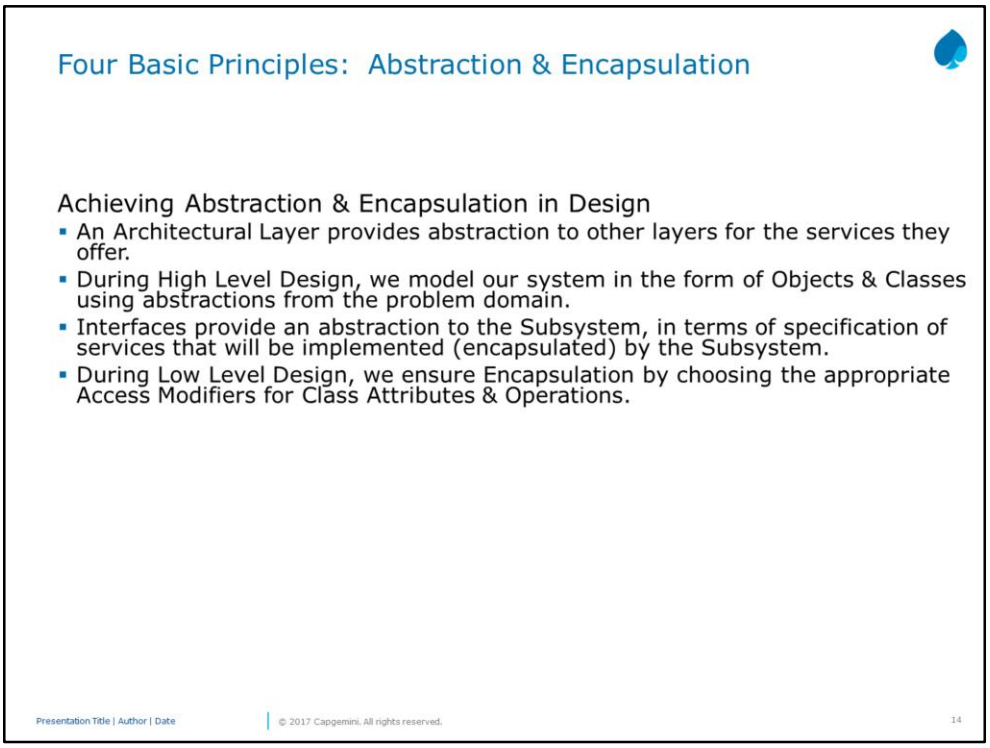
- Modularity requires appropriate use of Abstraction and Encapsulation. **Abstraction** is determining the essential qualities without getting into details. **Encapsulation** allows restriction of access of internal data.
- The concepts of **abstraction** and **encapsulation** are closely related. In fact, they can be considered to be two sides of a coin. 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 (that is, 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 helps in having less complex views and therefore a better understanding of the system. Increased **flexibility** and **maintainability** is achieved from maintaining a separation between "interface" and "implementation". Developers can change implementation details without affecting the user's perspective.
- We begin by abstracting out objects & classes from our problem domain; internal details ("encapsulated" part) is fleshed out and detailed as we move into Low Level Design.
- Interfaces provide an abstraction to services that can be offered by a subsystem.

Instructor Notes:

Explain that Abstraction is the "What" part, one that helps give the outside view of the system

Encapsulation is the How part giving the Internal View of the System

As a designer, we first abstract and then bother about corresponding encapsulation.

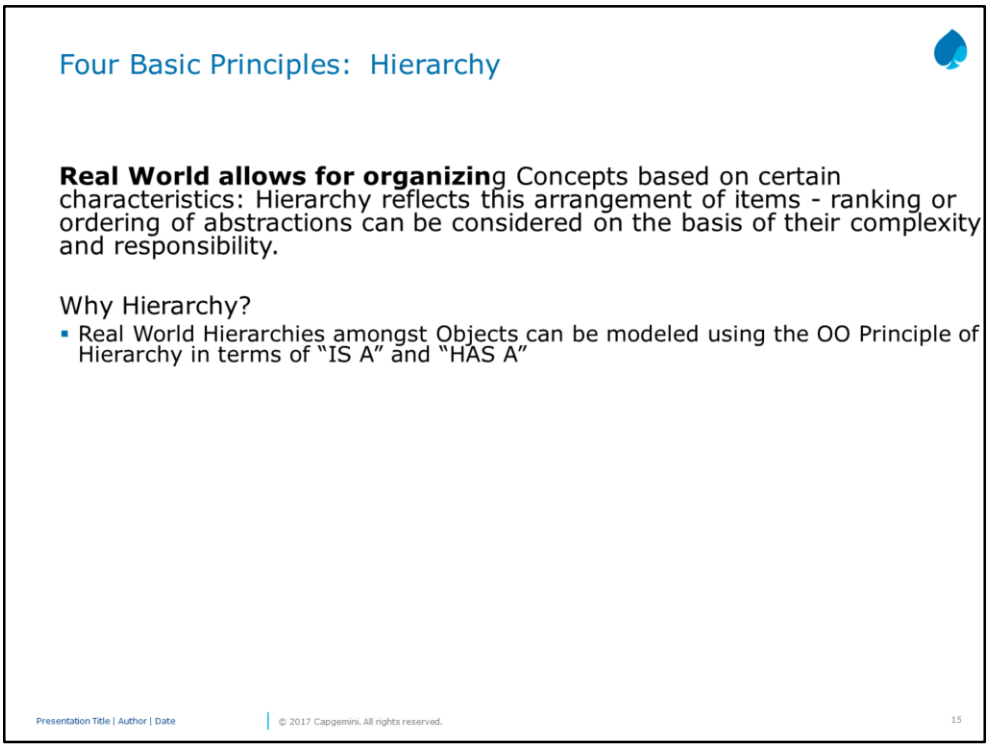


Object-Oriented Principles: Abstraction & Encapsulation

- Modularity requires appropriate use of Abstraction and Encapsulation. **Abstraction** is determining the essential qualities without getting into details. **Encapsulation** allows restriction of access of internal data.
- The concepts of **abstraction** and **encapsulation** are closely related. In fact, they can be considered to be two sides of a coin. 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 (that is, 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 helps in having less complex views and therefore a better understanding of the system. Increased **flexibility** and **maintainability** is achieved from maintaining a separation between "interface" and "implementation". Developers can change implementation details without affecting the user's perspective.
- We begin by abstracting out objects & classes from our problem domain; internal details ("encapsulated" part) is fleshed out and detailed as we move into Low Level Design.
- Interfaces provide an abstraction to services that can be offered by a subsystem.

Instructor Notes:

Explain that hierarchies are key to object-oriented platforms. Discuss differences between the class hierarchy and object hierarchy.



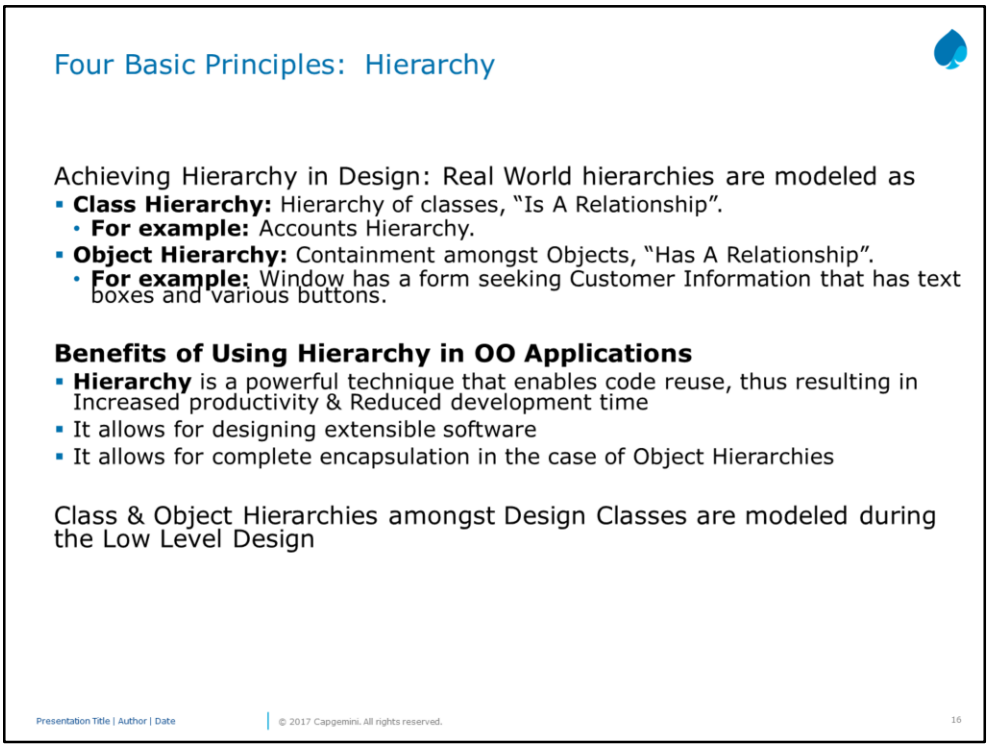
Object-Oriented Principles: Hierarchy:

- 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, such as current account and savings account, are part of the corresponding sub-classes. "Is A" relationship holds true, that is Current Account is an account as well as Savings Account is an account.
  - In an **object hierarchy**, it is the containership property, where one object is contained within another object. So a window contains a form. The form contains textboxes and buttons, and so on. Here we have "Has A" relationship, that is a form has a textbox.

- **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. However, it 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. However, it can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases the program reliability.

Instructor Notes:

Explain that hierarchies are key to object-oriented platforms. Discuss differences between the class hierarchy and object hierarchy.



Object-Oriented Principles: Hierarchy:

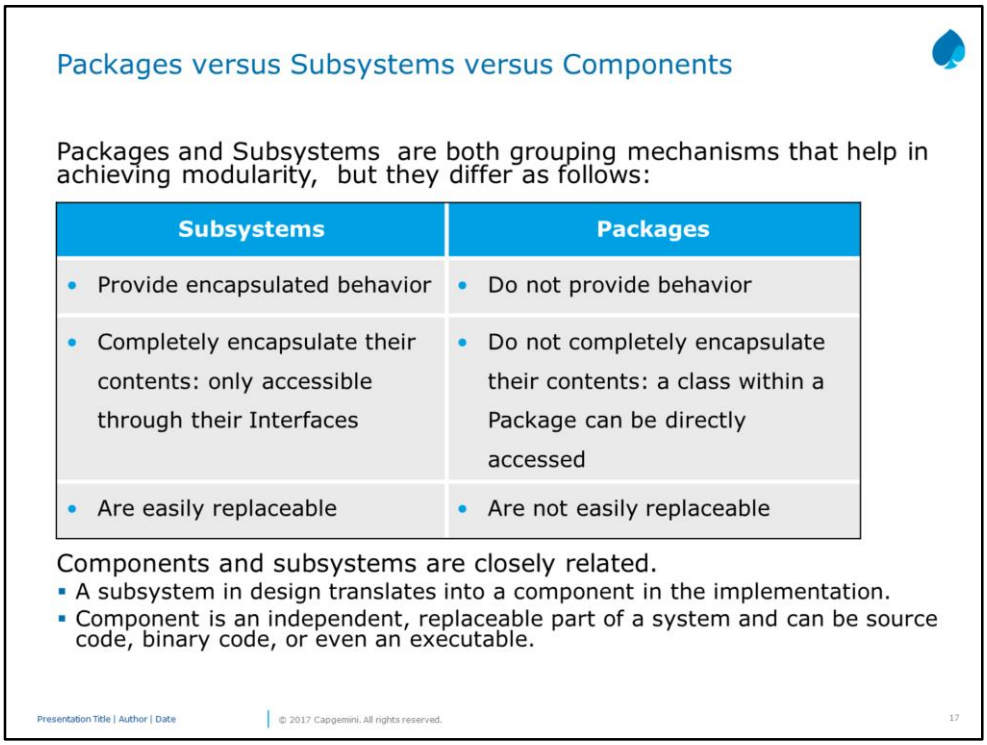
- Hierarchy is the systematic organization of objects or classes in a **specific sequence** in accordance to their **complexity** and **responsibility**.
  - In a **class hierarchy**, as we go up in the hierarchy, the abstraction increases. So all generic attributes and operations pertaining to an Account are in the Account superclass. Specific properties and methods pertaining to specific accounts, such as current account and savings account, are part of the corresponding sub-classes. "Is A" relationship holds true, that is Current Account is an account as well as Savings Account is an account.
  - In an **object hierarchy**, it is the containership property, where one object is contained within another object. So a window contains a form. The form contains textboxes and buttons, and so on. Here we have "Has A" relationship, that is a form has a textbox.

- **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. However, it 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. However, it can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases the program reliability.

Instructor Notes:

Given a choice between package and a subsystem for a given scenario, subsystem is always a better option!

Important to emphasize that a component is nothing but the physical implementation of a subsystem identified in design.



Packages versus Subsystems:

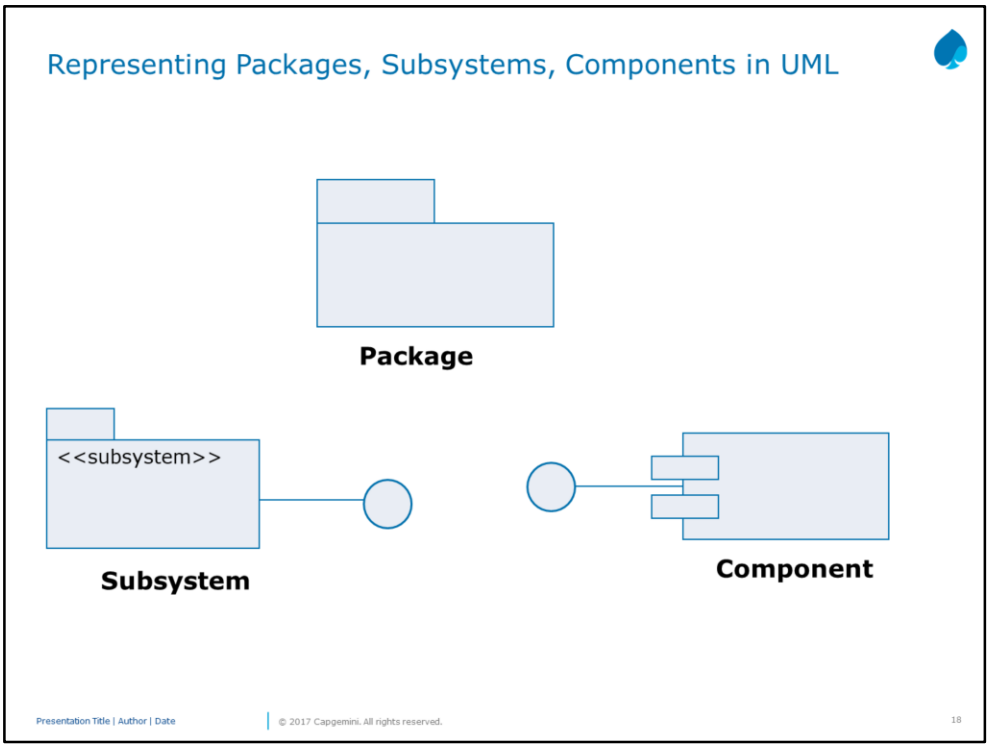
- Both **subsystems** and **packages** are means of grouping elements such that it becomes easy to understand the system.
- **Packages** are containers for elements that provide behavior. Packages are clubbing together of classes that are needed in common. Packages are used for model organizations and configuration management. We use packages when we import a jar file or use a namespace in our .NET/Java programs.
- **Subsystems**, on the other hand, are more compact and independent. They completely encapsulate their elements and provide behavior through their interfaces. Dependency is on the interface and not on the subsystem contents. In case of packages, dependency is on specific elements within the package. For eg. a calendar control in design is a subsystem, accessible only through exposed interfaces; one cannot access individual classes from within this subsystem.
- Subsystems are modeled such that their contents and internal behavior can change as long as the system's interface remains the same. This is not possible with packages. It is impossible to substitute one package with another unless both have the same public classes. External classes are dependent on the public classes and their public operations are grouped in a package. As a result, these packages once designed are difficult to eliminate or modify.

Components:

- A **component** is "a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces (Booch, 1999)."
- A component conforms to and provides the physical realization of a set of interfaces. Components are implementation things. They are the physical realization of an abstraction in your design.
- **Subsystems** and **components** have a relationship. A subsystem is the design representation of a component. They both encapsulate a set of replaceable behaviors behind one or more interfaces. What a subsystem is in a design model becomes a component in the implementation model.

Instructor Notes:

Note the stereotyped package notation along with interface for subsystem. Stereotypes are discussed towards the end of this lesson.



Representing Packages, Subsystems, Components in UML:

- In UML, a package is represented as a tabbed folder.
- A subsystem may be represented as a stereotyped package.
- A component has two small rectangles “protruding” out of a larger rectangle.

Packages help us model the real world structure of the system being developed. Each can be represented as a package. Off the Shelf components or any framework components which help us access individual constituents of the group can be represented as packages.
A UML Package may contain virtually any other UML element: for eg. while modeling the requirements or design using a tool, we may create packages to group related elements. A Use Case Model is a package which can contain Actors, Use Cases, Use Case Diagram and Activity Diagram. During the implementation stage, package typically constitutes of classes, interfaces or other packages.

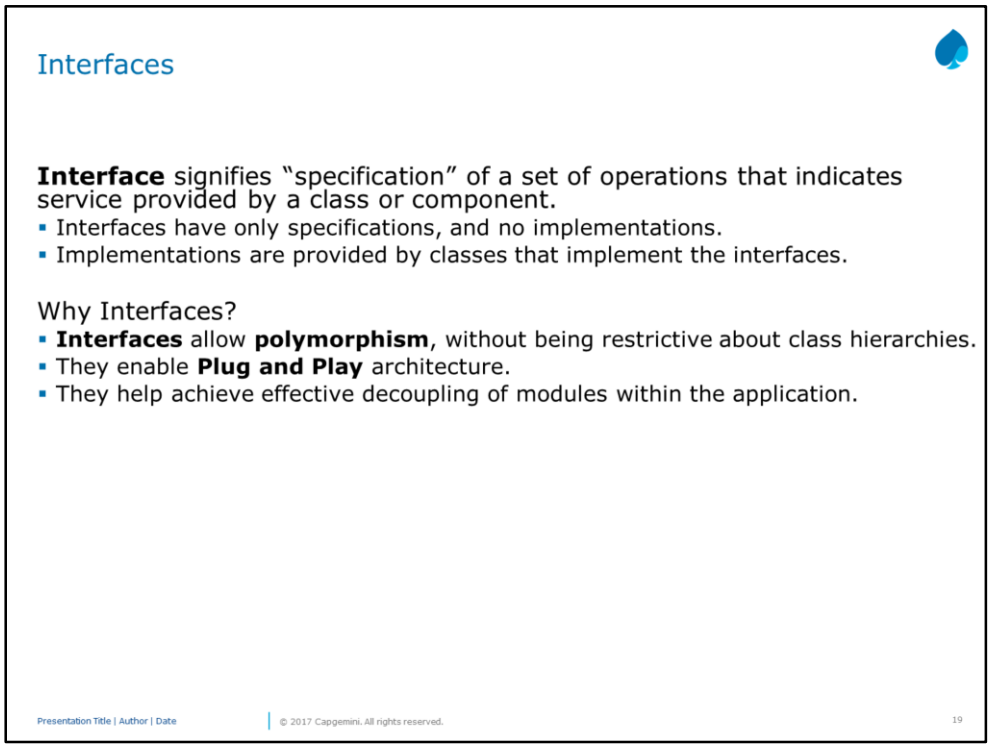
Subsystems are specialized groupings: specialized because the subsystem behavior will be accessible only through the exposed interfaces. The implementation of the Interface Operations will be done by the constituents of the subsystems, i.e. the classes which make up the subsystem. Unlike a package, one cannot directly access a constituent of a given subsystem.

A component is nothing but physical implementation of a subsystem. This may take the form of a DLL, EXE and so on.

Instructor Notes:

Interfaces is a key concept in object orientation. Even in OO world, languages like C++ do not really offer this kind of interface support.

Also encourage participants to compare interfaces with abstract classes



Interfaces:

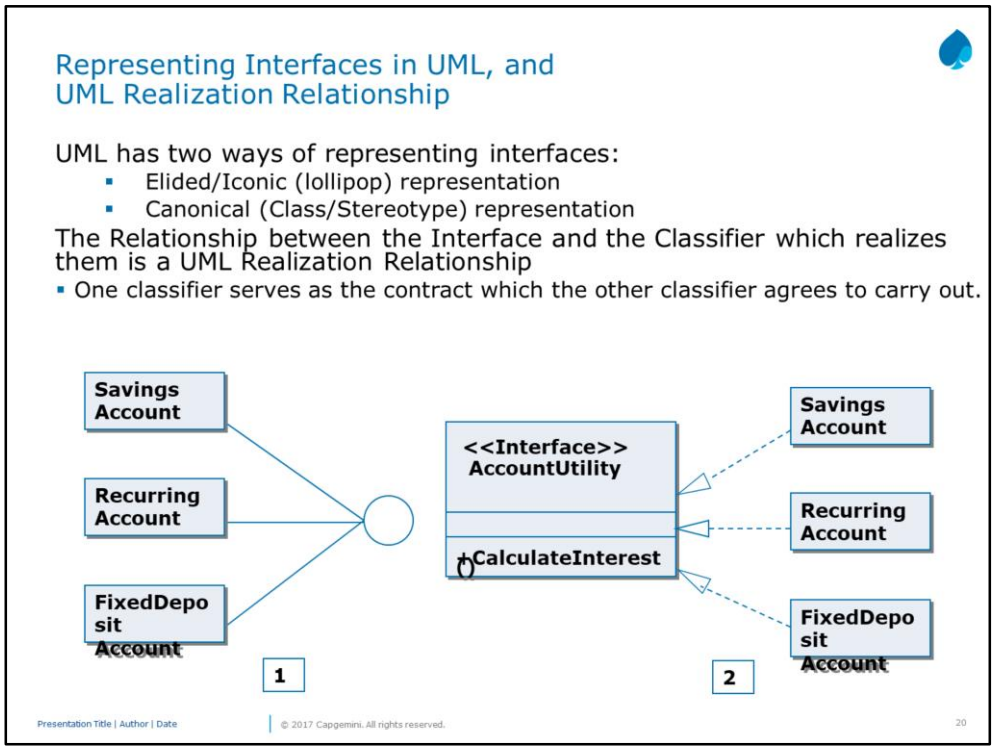
- Interfaces only "specify" operations, and not implementation of the operations. In that sense, they are like a contract specification, and to fulfill the contract, the corresponding class needs to provide implementations for all operations specified within the interface.
- **Polymorphism** and **interfaces** go hand-in-hand. Interfaces formalize polymorphism. If two objects use same methods to achieve different but more or less similar results, then they are polymorphic in nature.

- Plug and Play architecture, which allows easy maintainability and extensibility, is a benefit of using interfaces. Plug and Play architecture means that any classifiers (e.g., classes, subsystems, components) which realize the same interfaces may be substituted for one another in the system, thereby supporting the changing of implementations without affecting clients. For eg. in our subsystem example of Calendar Control, so long as we retain the same interfaces, we can change internal logic or set of classes or UI – none of this will change anything for the client using this control. We can even substitute one calendar control with another calendar control, so long as the supported interfaces are the same. All this means that modules will be effectively decoupled from each other.

Instructor Notes:

Distinguish between both approaches and in which scenario, which notation would be used (based on level of detailing expected to be denoted for an interface)

Realization emphasizes on separating out specification from implementation. Concept of interface allows one to do that. Another example of realization relationship is between use case and use case realization as we shall see in later sections. Use Case is the specification of the behavior and Use Case Realization is the implementation (in terms of designing the use case) of the behavior.



Representing Interfaces:

- The two ways of representing interfaces in a UML model are as follows:
  1. **Elided / Iconic representation:** This is commonly known as the lollipop method. In this method, you just get to known about the existence of an interface. Elided means to omit; operation details are omitted here.
  2. **Canonical representation:** It is also referred to as the Class or Stereotype representation. This method provides details of the interface.

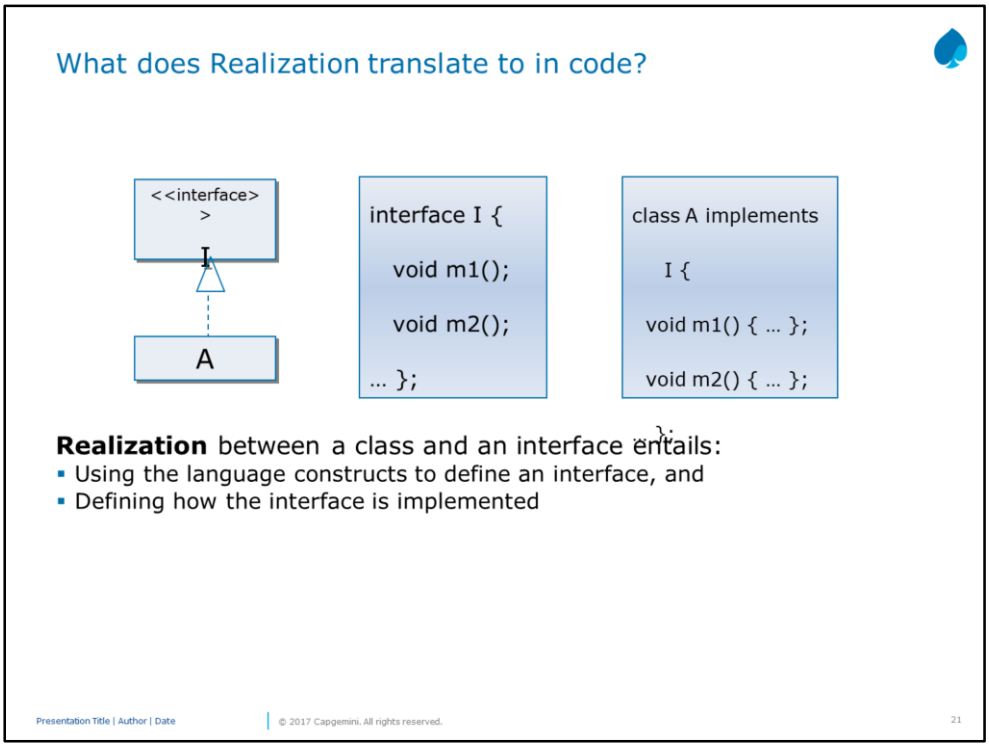
UML Relationships: Realization:

- **Realization** is a semantic relationship between two classifiers. One classifier serves as the contract that the other classifier agrees to carry out.
- The **realizes relationship** is a combination of a **dependency** and a **generalization**. It is not true generalization, as only the "contract" (that is to say, operation signature) is "inherited". This "mix" is represented in its UML form, which is a combination of dependency and generalization.
  - The realizes relationship may be modeled as a dashed line with a hollow arrowhead pointing at the contract classifier (canonical form as represented in relationship 2 above).
  - It can be modeled as a "lollipop" (elided form as represented in relationship 1 above) when combined with an interface.

In the example shown here, AccountUtility is an interface which provides the specification for operations related to Accounts. One such operation, CalculateInterest, is indicate here as part of the interface specification. The implementation of this operation is provided by the implementing classes i.e. in this example, the Savings Account, Recurring Account and FixedDeposit Account classes.

Instructor Notes:

C++ does not support interface so it will not have the language constructs. Syntaxes will be different for different languages.

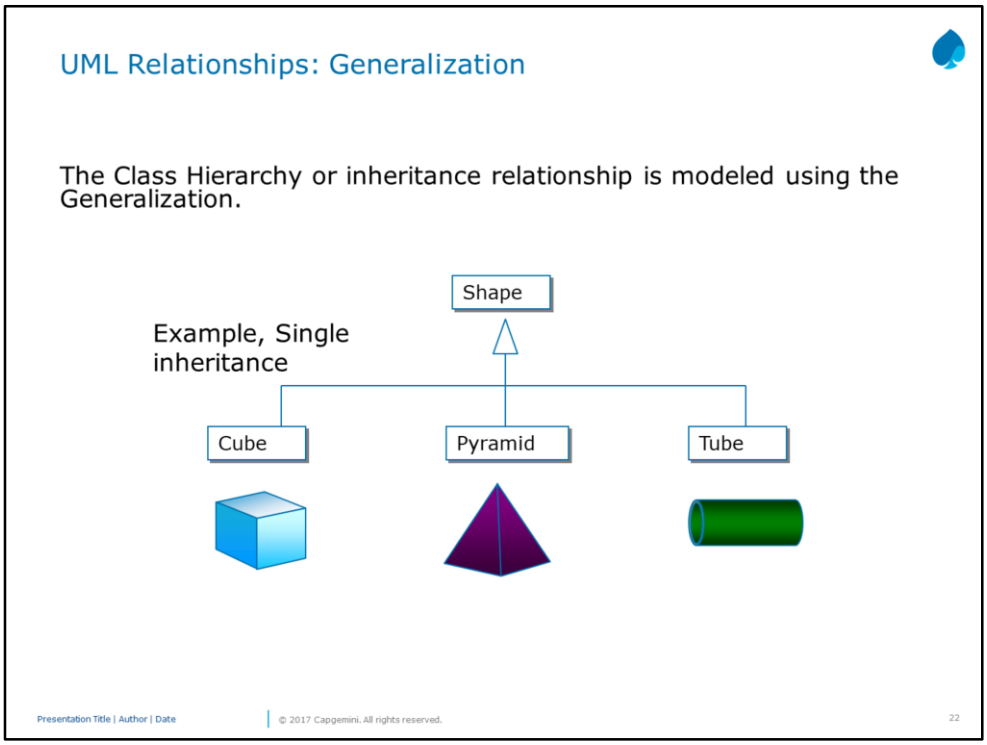


UML Relationships: What does Realization translate to in code?

Object Oriented languages provide language constructs for defining interfaces and implementing the interfaces. This will be used for coding realization.

Instructor Notes:

Participants are usually quite clear about generalization relationship. Emphasize what gets inherited – subclass can inherit structure, behaviour and relationships from the base class.

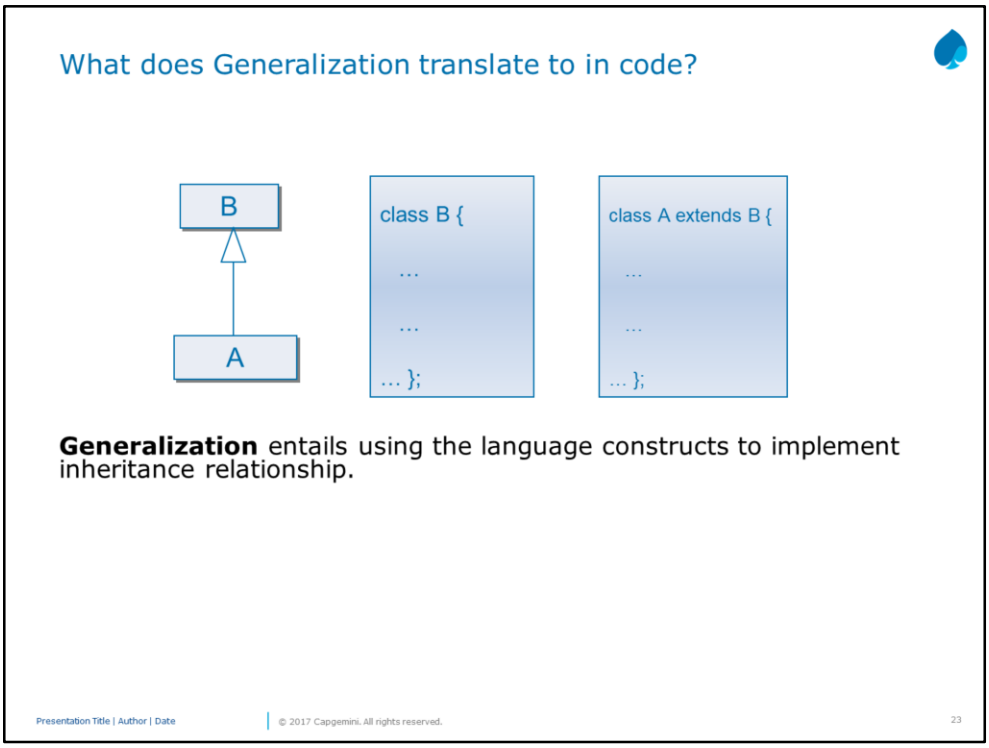


UML Relationships: Generalization:

- **Generalizations** are denoted as paths from specific elements to generic elements, with a hollow triangle pointing to the more general elements.
- In **single inheritance**, a subclass will have only one parent class.

Instructor Notes:

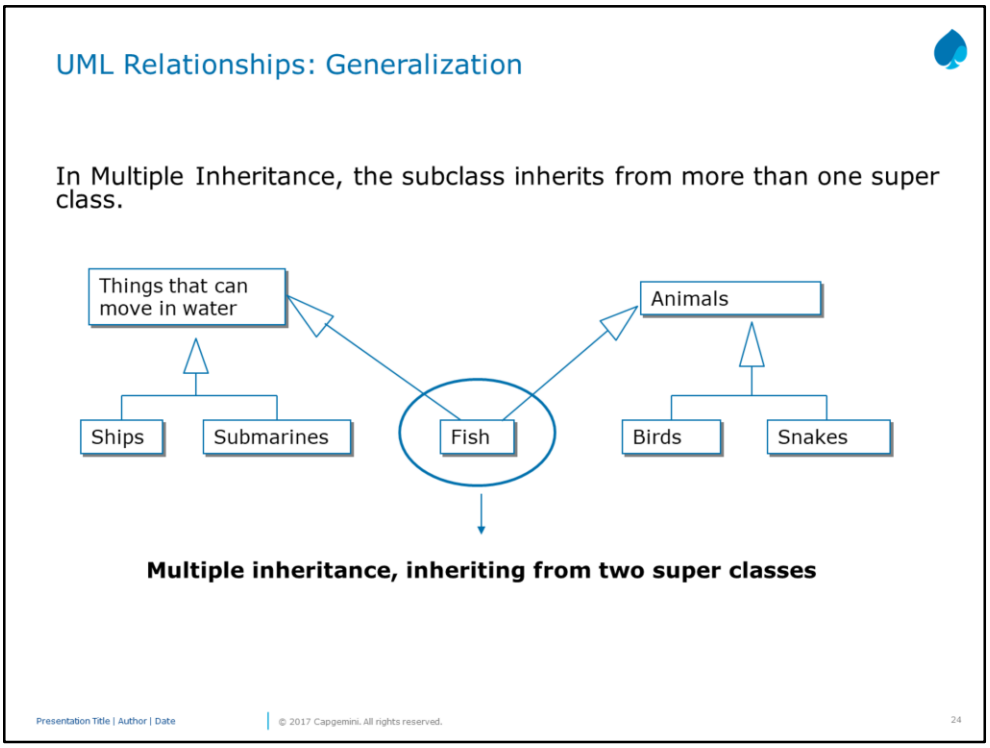
This usually will not need explanation. Syntax for inheritance will vary from language to language.



UML Relationships: What does Generalization translate to in code?
Object oriented languages provide language constructs for implementing inheritance relationship. This will be used for coding generalization.

Instructor Notes:

Languages like C++ have workarounds like virtual base classes to take care of the ambiguities that arise in multiple inheritance (the diamond problem). Java does not support multiple inheritance and interfaces are used instead.

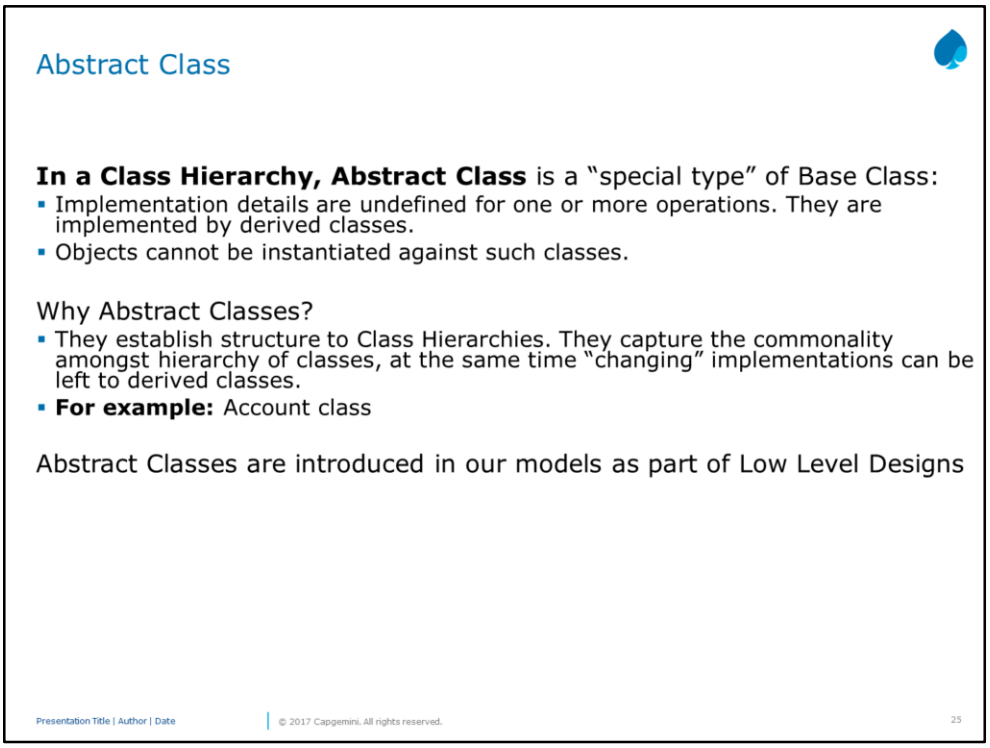


UML Relationships: Generalization:

- In the example in the slide, the Fish class shares structure and behavior of more than one super class.
- Multiple inheritance has its own ambiguities, and all programming environments may not support it.

Instructor Notes:

People may tend to mix this up with "Abstraction". Focus on need for abstract classes especially at top of hierarchy

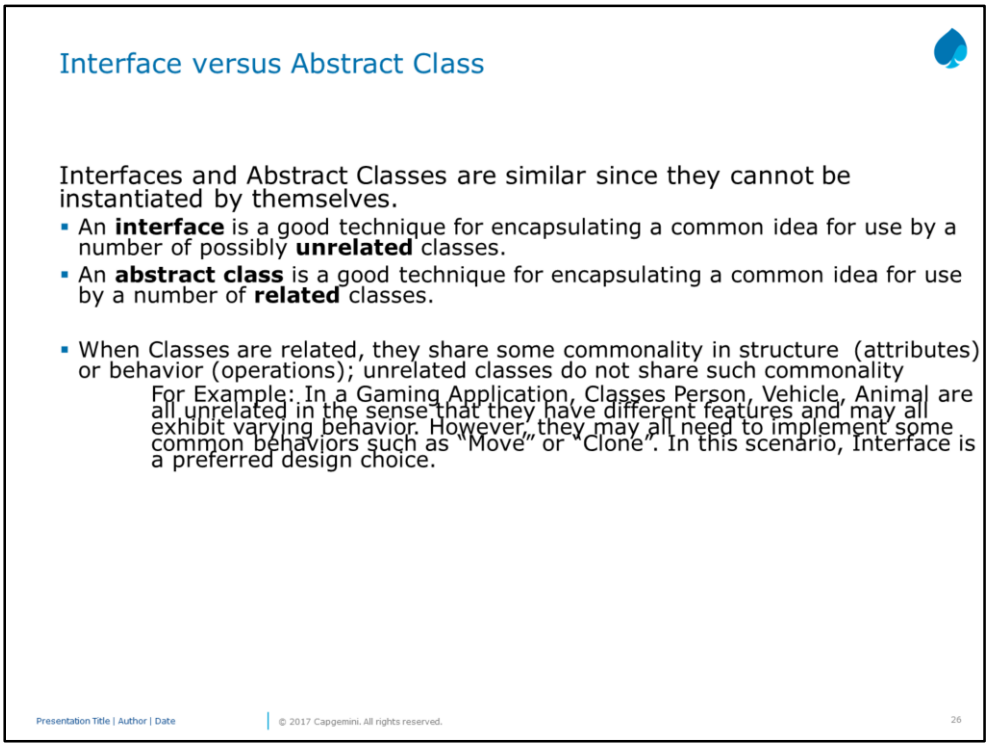


Abstract Class:

- **Abstract classes** are a "special type" of **base classes**.
  - In addition to normal class members, they have abstract class members. These abstract class members are "methods" that are declared without an implementation.
  - All classes derived directly from abstract classes must implement all these abstract methods.
- Abstract classes can never be instantiated because the members have no implementations.
- Abstract classes sit toward the top of a class hierarchy. They establish structure and meaning to code. They make frameworks easier to build. These functionalities are possible because abstract classes have information and behavior that is common to all derived classes in a framework. The aspects that are specific to different derived classes can be left open for implementation in the derived classes.
- **For example:** One does not really have just an Account. However, one has an account of specific type, say an account of savings account type or current account type.
- In UML, an abstract class is indicated in italics as we shall see in a later section.

Instructor Notes:

Explain the differences between interfaces and abstract classes

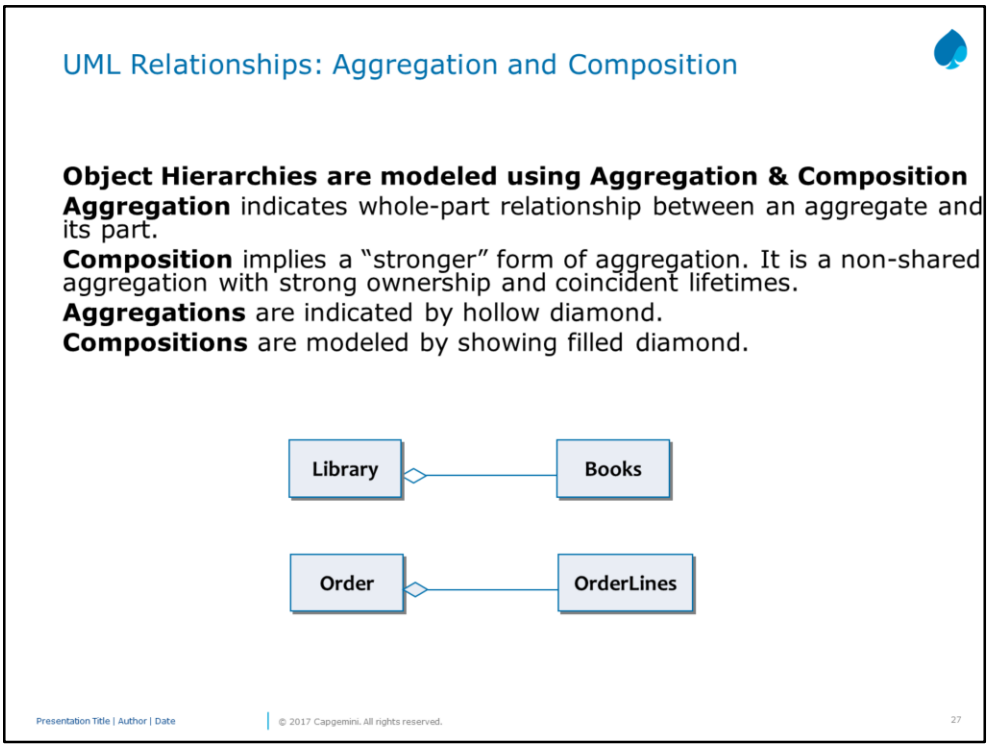


Interface versus Abstract Class:

- **Interfaces** do not contain implementation for any members. **Abstract classes** allow you to partially implement your class.
- **Interfaces** are used to define the peripheral abilities of a class. An **abstract class** defines the core identity of a class and there it is used for objects of the same type.
- If we add a new method to an **Interface**, then we have to track down all the implementations of the interface and define implementation for the new method. If we add a new method to an **abstract class**, then we have the option of providing default implementation, and therefore all the existing code might work properly.

Instructor Notes:

- Encourage participants to distinguish between Association, Aggregation and Composition. Association by itself does not imply containership, which Aggregation and Composition imply. How strong is the containership decides whether it is aggregation or composition. Best guideline is to check if lifetimes of whole and part are coincidental i.e. do they live and die together?
- Note that composition does not necessarily mean "physical containment" – as in example of Order and Orderlines.

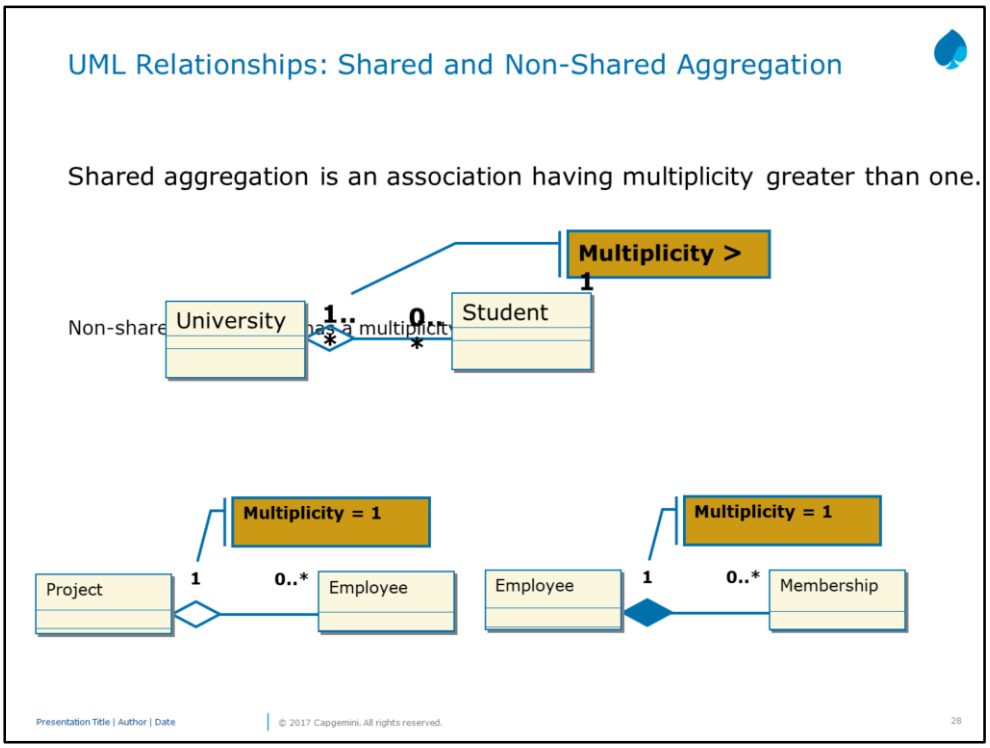


UML Relationships: Aggregation and Composition:

- Aggregation and Composition are forms of Association relationship, where we are looking at a "Has a" or a container relationship.
- **Aggregation:**
  - It is an association between two classes that are related to each other by virtue of being "Whole" and "Part" to each other. In an aggregation, it is possible for the "Part" to exist without the "Whole".
    - **Example:** Library-books, Company-employees
  - This is exactly the same as an association with the exception that instances cannot have cyclic aggregation relationships (that is, a part cannot contain its whole).
- **Composition:**
  - It is a stronger form of aggregation. The "Parts" live and die along with the "Whole". They cannot exist independent of the Whole.
    - **Example:** Order-OrderLines, Hand-Fingers
  - One can say that the lifetime of the "part" is controlled by the "whole". The important thing to note in a composed aggregation is that the part can only belong to the whole.
- Composition and Aggregation are modeled with filled and hollow diamonds, respectively, on the whole part.

Instructor Notes:

Composition is always non-shared aggregation but all non-shared aggregations need not be compositions – it can be aggregation too as indicated in the examples here.

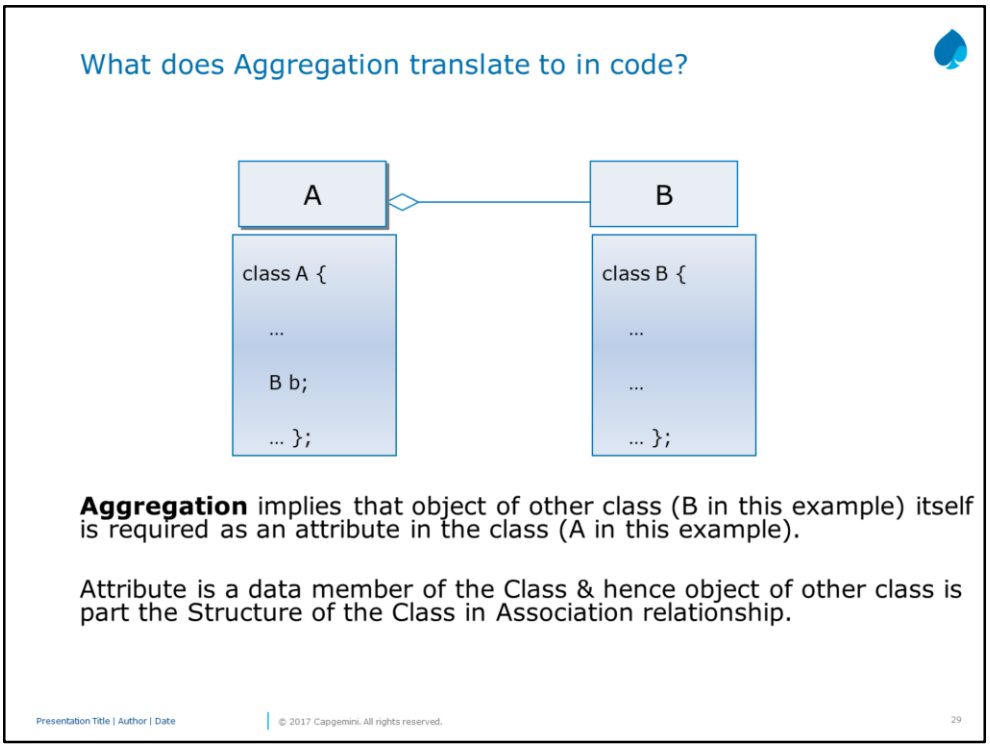


UML Relationships: Shared and Non-Shared Aggregation:

- For composition, the multiplicity of the aggregate end may not exceed one (it is unshared). The aggregation is also unchangeable. That is, once established its links cannot be changed. Parts with multiplicity having a lower bound of 0 can be created after the aggregate itself. However, once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.
- **Non-shared aggregation** does not necessarily imply composition.
- **Shared aggregation** is an aggregation relationship that has a multiplicity greater than one established for the aggregate. Furthermore, destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation either forms a graph or a tree with many roots.
  - An example of **shared aggregation** may be between a University class and a Student class (the University being the "whole" and the Students being the "parts"). With regards to registration, a Student does not make sense outside the context of a University. However, a Student may be enrolled in classes in multiple Universities.
  - In the relation between Project and Employee, the **non-shared aggregation** will imply that employee can work only on one project at a time. It is an aggregation relationship because Employees can exist independently of the Project.
  - Similarly, information about employee membership to various clubs is strongly bound to individual employee. If the Employee objects are destroyed (for example, Employee quits), then information related to their membership is no longer maintained – this is what is conveyed through the composition relationship.

Instructor Notes:

For languages which do not support “reference”, association and aggregation would translate to the same thing in the code. Then “containership” defined by the aggregation relationship will have to be managed by the developer in the code.

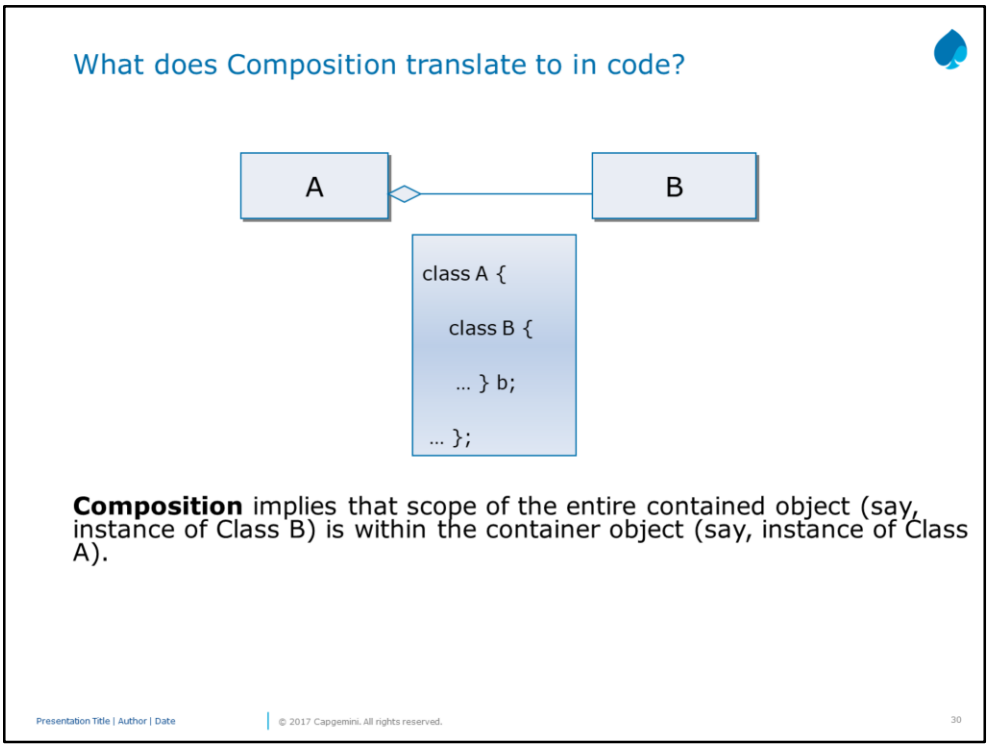


UML Relationships: What does Aggregation translate to in code?

- **Aggregation** translates to a need of maintaining the object of the other class as an attribute in the class. As in the case of association, it becomes a part of the structure itself in the “Whole” class. Class B, the “Part” class is separately defined.
- Typically, in an aggregation relationship, multiplicity may be more than one on the “Part” class. In such a case, in the code this would translate to the use of an appropriate collection class to store the contained objects.

Instructor Notes:

For languages where nested classes are not available, aggregation and composition would be defined in the same way. But for composition, the lifetimes of the contained objects will need to be within the scope of the container object…which is not so for aggregation.



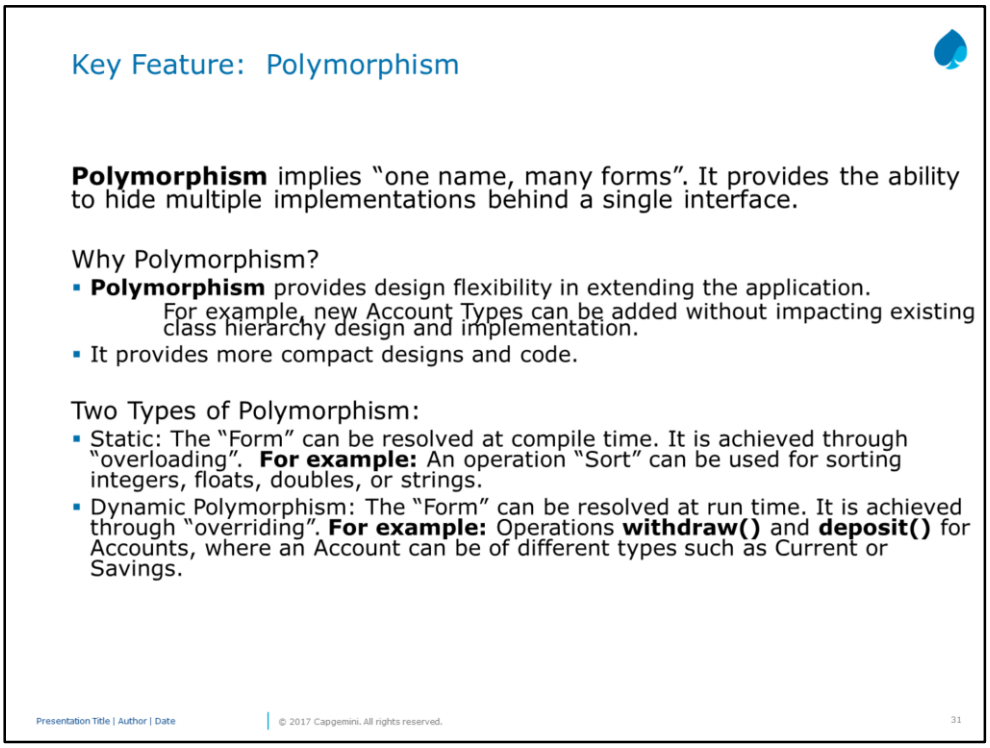
UML Relationships: What does Composition translate to in code?

- **Composition** means that lifetime of a container and contained objects are coincidental, that is, the part lives and dies with the whole. Hence it is natural that its scope has to be within the container (Class A in the above example).
- For languages that offer the facility of nested classes, it is the best construct to use for composition relationship. This is because it provides for the fact that the nested or the inner class cannot be instantiated outside the scope of the outer class. For languages where this construct is not available, the classes will get defined as we saw in aggregation. Furthermore, the developer has to appropriately manage the lifetimes of the objects in the code.

Instructor Notes:

Polymorphism is a key feature of object oriented languages. Explain both types, their differences.

Small activity to drive home concept of Polymorphism: Ask participants to draw a shape. Then randomly ask a few members what shape they drew. Invariably they would be different! Then the punch line: "What you have just done is exhibited polymorphic behavior".

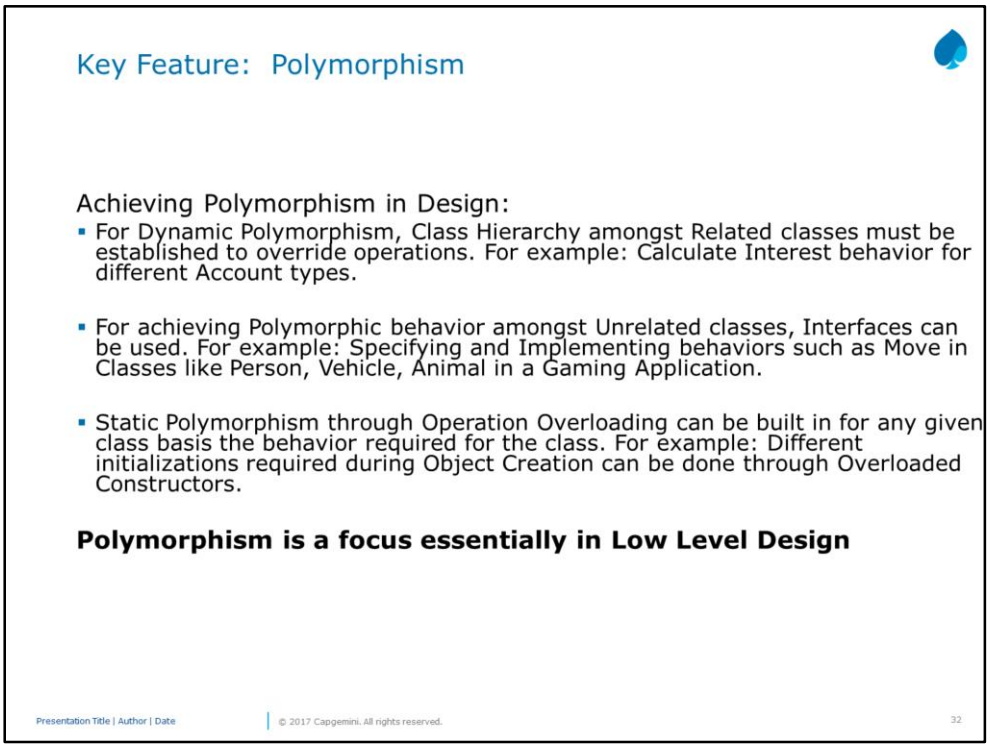


Object-Oriented Principles: Key Feature: Polymorphism:

- 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!
- Suppose the banking system needs a new kind of account. Then extending without rewriting the original code is possible with the help of polymorphism.
- In a non-OO system, our code would have multiple Conditional Statements to implement this kind of a feature. For eg. If Account is of type Current Account, the CalculateInterest of Current Account needs execution; if Account is of type Savings Account, the CalculateInterest of Savings Account needs execution and so on. This leads to elaborate coding and changes when new Account Types are introduced.
- In an OO system, the same will be **myAccount.calculateInterest()**. With object technology, each Account is represented by a class, and each class will know how to perform appropriate operation for its type.
  - The requesting object simply needs to ask the specific object (for example, SavingsAccount) to withdraw.
  - The requesting object does not need to keep track of three different operation signatures.
- **Overloading** is when functions having same name but different parameters (types or number of parameters) are written in the code. When multiple sort operations are written, each having different parameter types, the right function is called based on the parameter type used to invoke the operation in the code. This can be resolved at **compile time** itself since the type of parameter is known.
- On the other hand, the functions **withdraw()** and **deposit()** can be coded across different account classes. At runtime, based on the type of Account object (namely, object of Current Account or Savings Account) that is invoking the operation, the right operation will be referenced. **Overriding** occurs when functions with same signature provides for different implementations across a hierarchy of classes.

Instructor Notes:

Interfaces allow complete decoupling can be used if there is no common operations/data to be considered

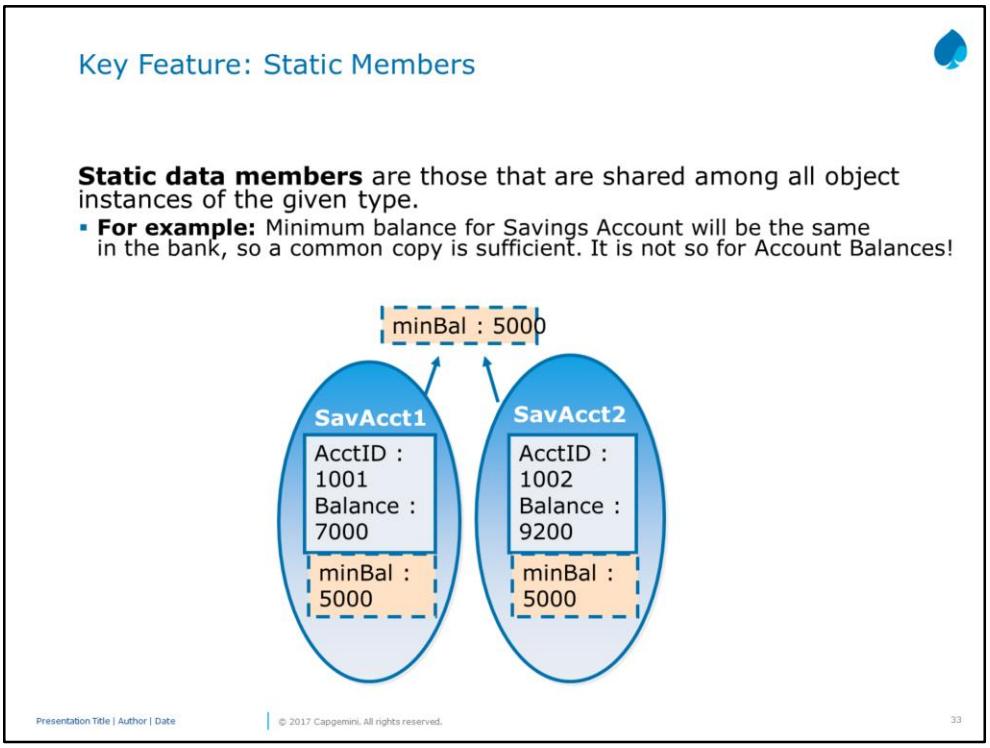


Polymorphism can be achieved through a variety of options: In the Account example of earlier slide, an Account Hierarchy will be required to enable Dynamic Polymorphism. If Classes are unrelated (Gaming Application Example), we can use Interfaces to enable polymorphic behavior. Polymorphism will essentially be taken care of during Low Level Design.

Instructor Notes:

Explain Static data as well as functions

Fig explanation : A class can have instance as well as class variables. For class variables, objects share this single copy of data…



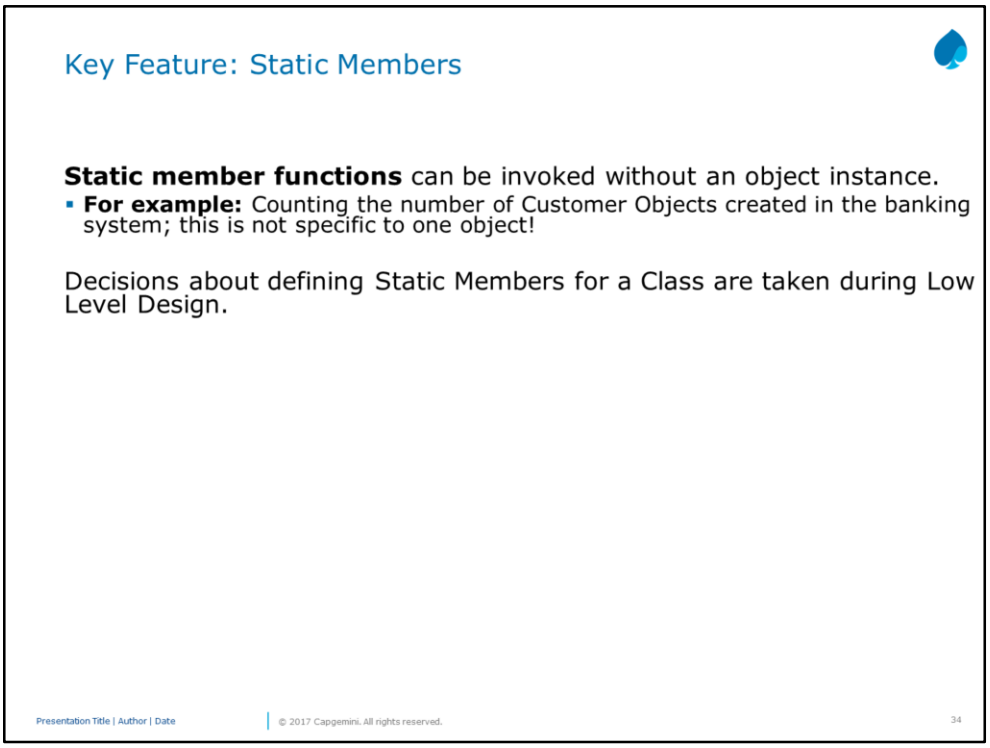
Some More Object-Oriented Concepts: Static Members:

- **Static data** is allocated once and shared among all object instances of the same type.
- **For example:** If you create three instances of Savings Account, then each Savings Account object maintains the same copy of the Balance field. However, the minimum balance will be the same across all the instances.

- In UML, a static member (or class variable) is indicated with an underline as we shall see in one of the later sections.

Instructor Notes:

Explain Static data as well as functions

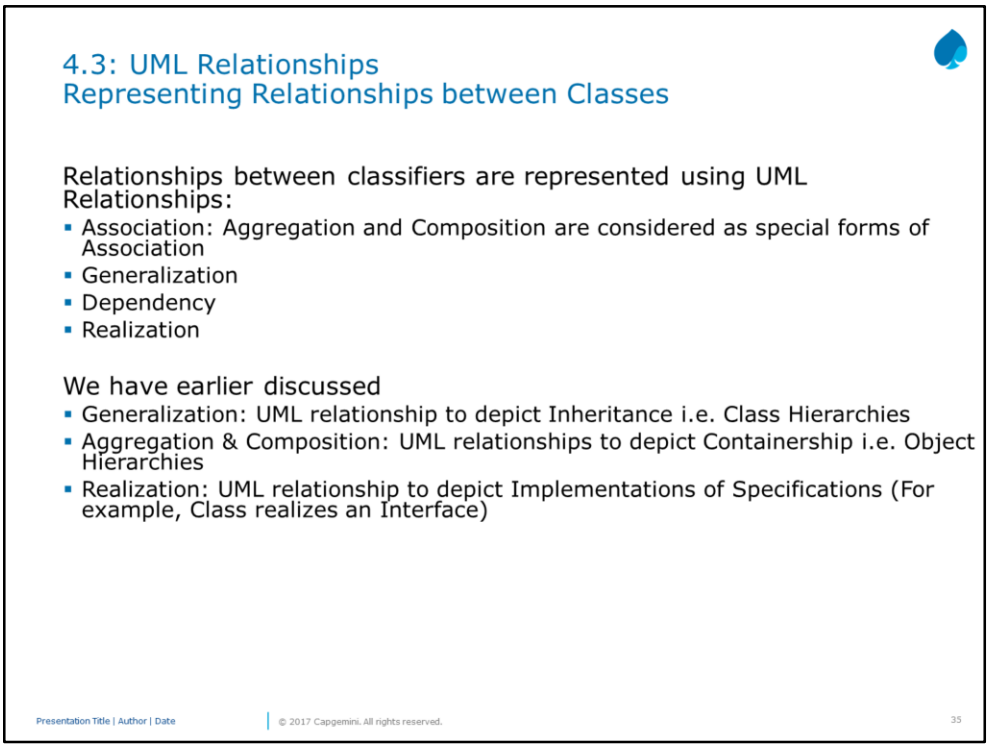


Some More Object-Oriented Concepts: Static Members:

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

Instructor Notes:

Emphasize meaning of each one; and difference as against other relationships. Very important to know what it means and implies in the code.

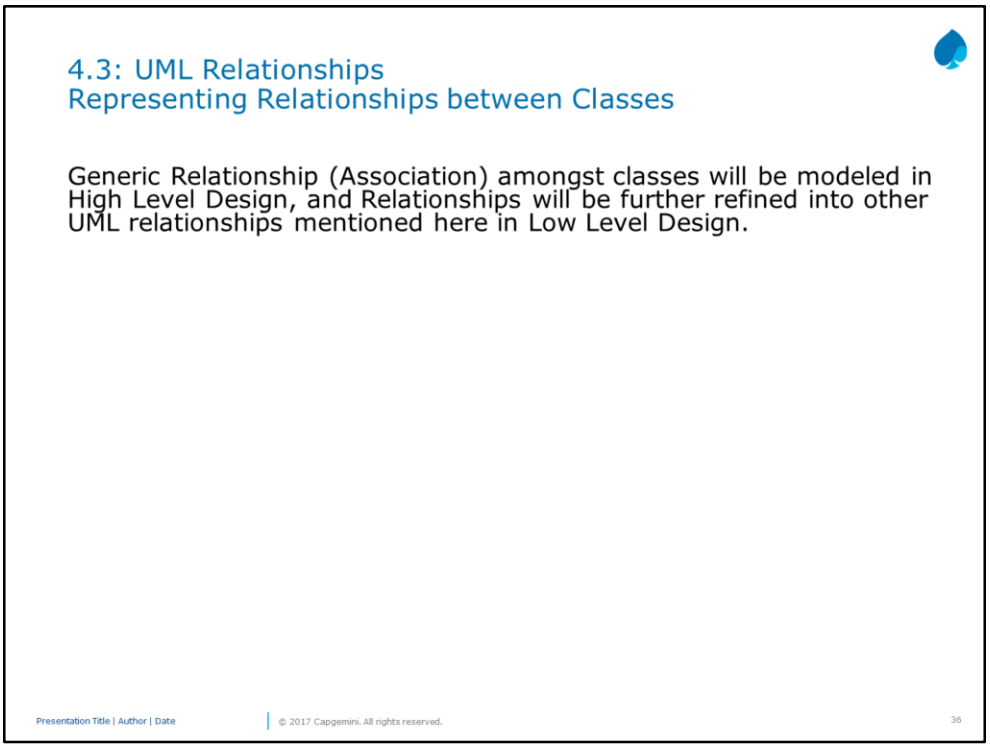


<u>UML Relationships</u>:

- A classifier by itself does not serve any purpose. Collaboration is required amongst these entities. Classifiers can be related to each other by different UML relationships.
- Four standard UML relationships exist. They are as shown in the above slide.
- While we have seen some of these on earlier slides, we shall now discuss the remaining, i.e. Association and Dependency.

Instructor Notes:

Emphasize meaning of each one; and difference as against other relationships. Very important to know what it means and implies in the code.

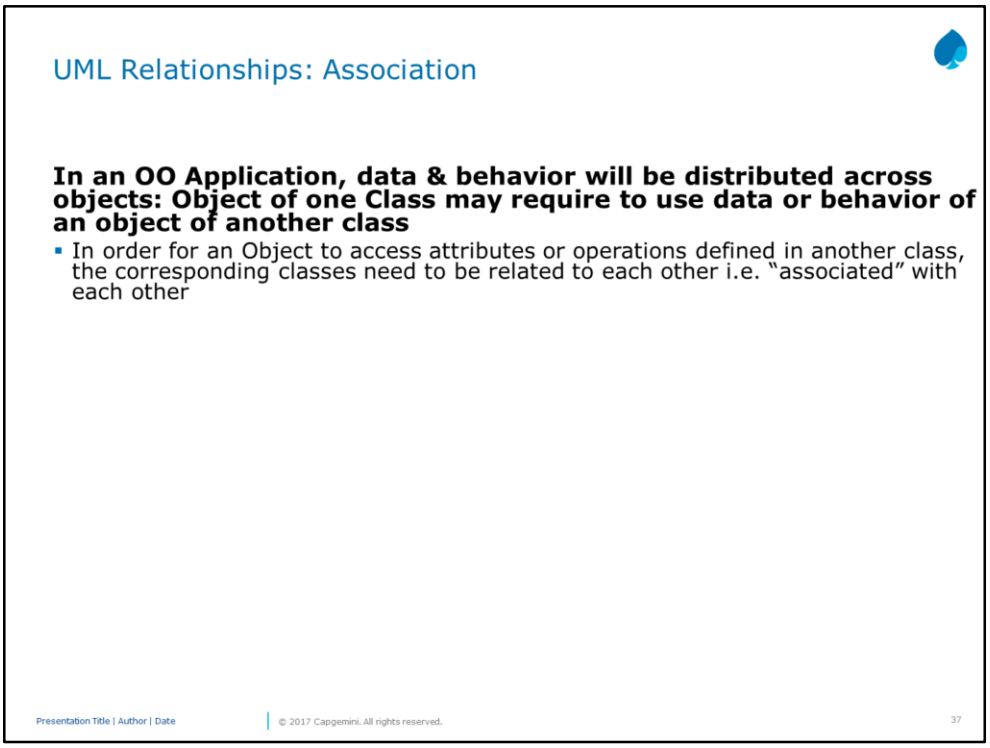


UML Relationships:

- A classifier by itself does not serve any purpose. Collaboration is required amongst these entities. Classifiers can be related to each other by different UML relationships.
- Four standard UML relationships exist. They are as shown in the above slide.
- While we have seen some of these on earlier slides, we shall now discuss the remaining, i.e. Association and Dependency.

Instructor Notes:

For simple access amongst objects across two different classes, association relationship is necessary.



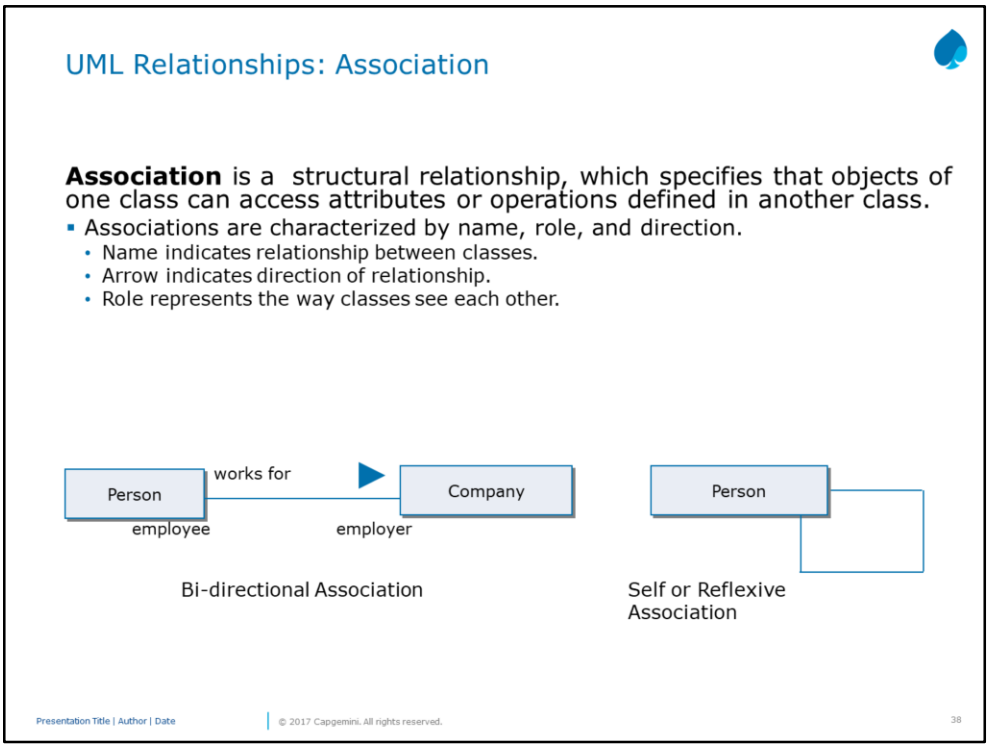
UML Relationships: Association:

What is Association?

- When an object of a class needs access to attributes or methods defined in another class, it will need a way to access the objects of the other class. One class will be visible to objects of another class if they are related by UML association relationship.
- **Associations** are normally simple relationships between two classes, and are indicated by using solid paths between the two class symbols.
- Associations may be characterized by name, role, and directions. None of them are mandatory.
  - The name signifies purpose of the association, and is written along with the line indicating the association.
  - In case there are specific roles played by classes in the association, then the role name, which is written near the class, indicates the same.
  - Arrows are used to indicate uni-directional or bi-directional relationships. Absence of any arrows indicates that either no inferences can be drawn about navigability or the association can work in both directions.
- There are instances of classes having **association** to **itself**. It usually means that an instance of the class has association with another instance of the same class. For example, an employee object is related to a manager object, with both employee object and manager object being instances of the Person class. Another scenario can be of a relationship between a parent and child object, both being instances of Person class.

Instructor Notes:

For simple access amongst objects across two different classes, association relationship is necessary.



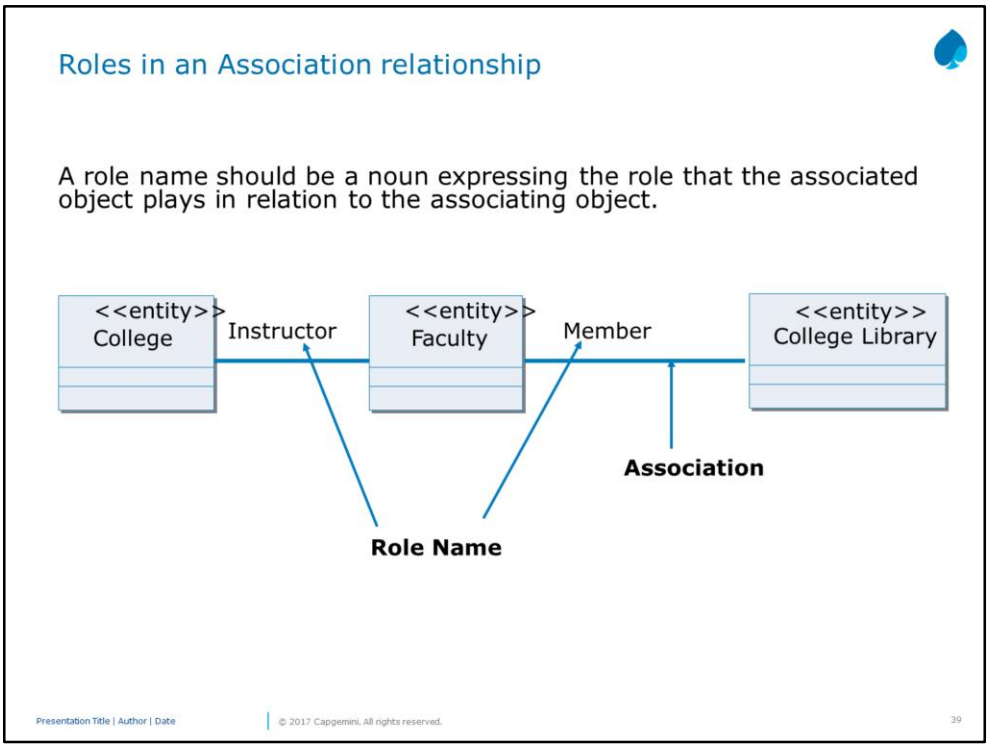
<u>UML Relationships: Association</u>:

What is Association?

- When an object of a class needs access to attributes or methods defined in another class, it will need a way to access the objects of the other class. One class will be visible to objects of another class if they are related by UML association relationship.
- **Associations** are normally simple relationships between two classes, and are indicated by using solid paths between the two class symbols.
- Associations may be characterized by name, role, and directions. None of them are mandatory.
  - The name signifies purpose of the association, and is written along with the line indicating the association.
  - In case there are specific roles played by classes in the association, then the role name, which is written near the class, indicates the same.
  - Arrows are used to indicate uni-directional or bi-directional relationships. Absence of any arrows indicates that either no inferences can be drawn about navigability or the association can work in both directions.
- There are instances of classes having **association** to **itself**. It usually means that an instance of the class has association with another instance of the same class. For example, an employee object is related to a manager object, with both employee object and manager object being instances of the Person class. Another scenario can be of a relationship between a parent and child object, both being instances of Person class.

Instructor Notes:

While roles are not mandatory, tools that forward engineer use the role name while generating code skeletons

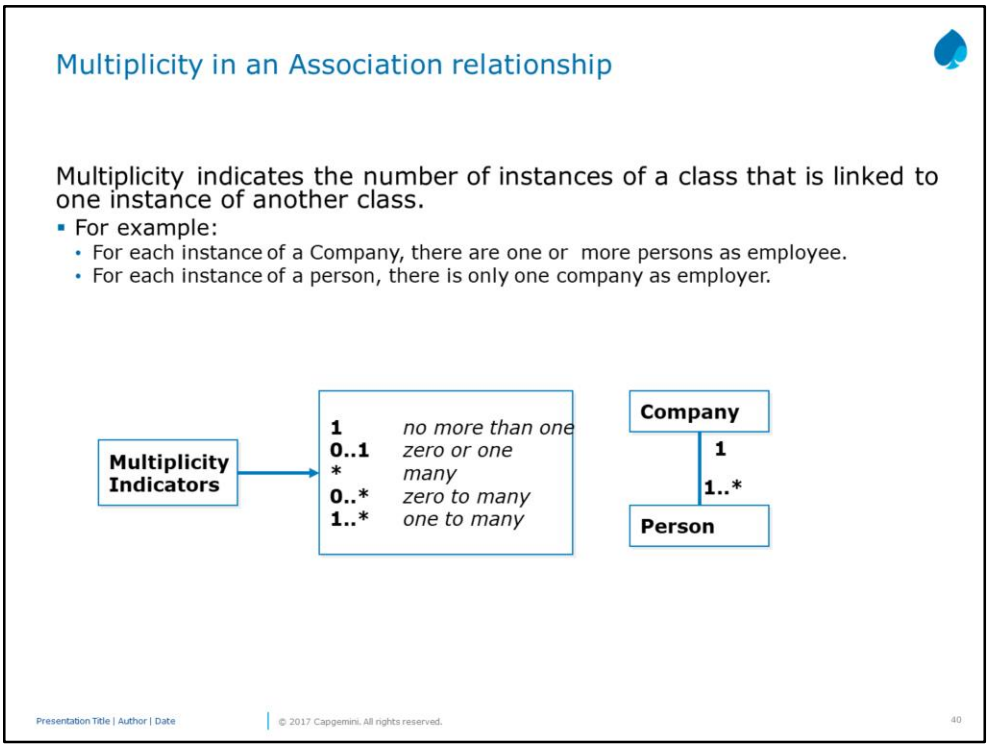


UML Relationships: Roles in an Association relationship:

- Each end of an association has a role in relationship to the class on the other end of the association. The role specifies the face that a class presents on each side of the association.
  - A role must have a name, and the role names on opposite sides of the association must be unique.
  - The role name should be a noun indicating the associated object's role in relation to the associating object.
- The use of association names and role names is mutually exclusive, that is, one would not use both an association name and a role name. For each association, select the one that conveys more information.
- The role name is placed next to the end of the association line of the class it describes.
- In case of self-associations, role names are essential to distinguish the purpose for the association.
- In the above example, the Faculty participates in two separate association relationships, playing a different role in each.

Instructor Notes:

Multiplicity is not mandatory either, but when captured, gives a good understanding of the relationships involved.

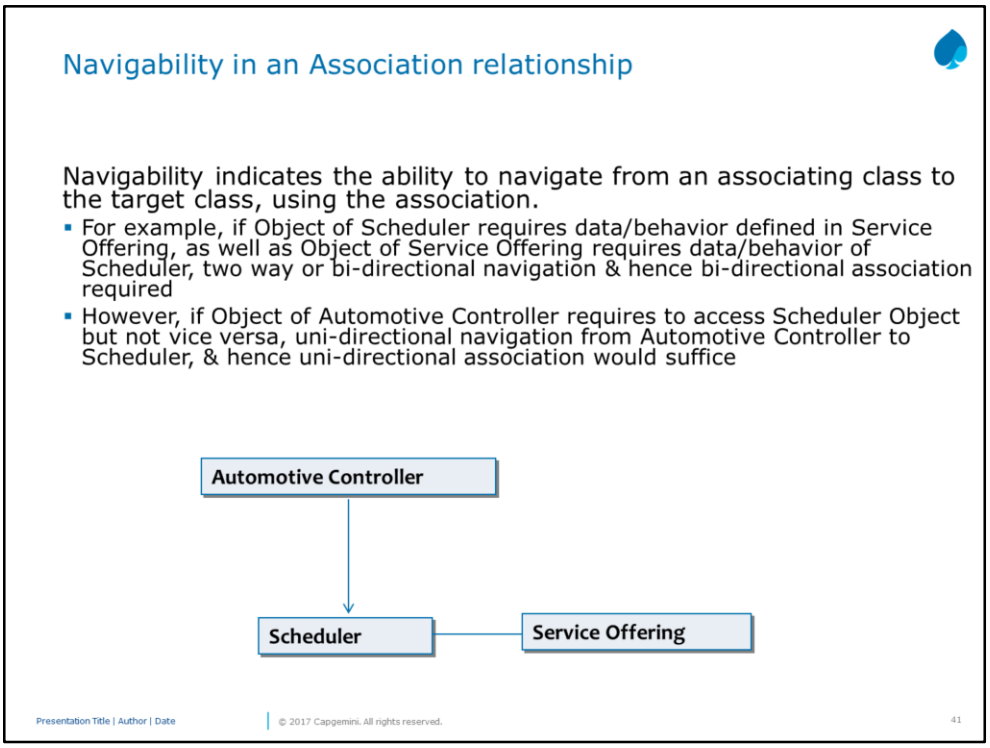


<u>UML Relationships: Multiplicity in an Association relationship</u>:
What is Multiplicity?

- **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 persons, and a person is associated with one company".
- Take a role and specify the multiplicity of its class, and how many of objects of the class can be associated with one object of the other class.
- In UML, multiplicity is usually indicated by a text-expression on the role. This text-expression is a list of integers separated by a comma.
- As multiplicity is relationship between instances of two classes, it is essential to define this multiplicity at both ends of the association.
- In case no multiplicity is specified, it just means that nothing is defined as far as the multiplicity is concerned.

Instructor Notes:

The direction of the arrow indicates what is visible to other class.



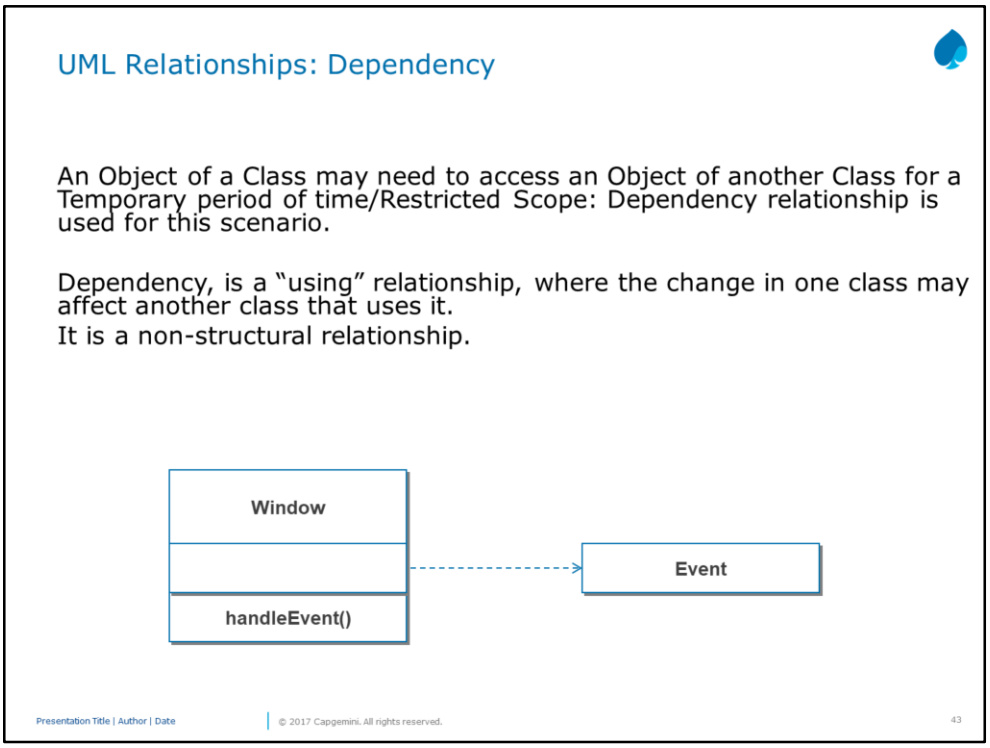
<u>UML Relationships: Navigability in an Association relationship</u>:

- In the example shown in the above slide:
  - **Navigability** is shown from the object “Automotive Controller” to the object “Service Offerings” through “Scheduler”. This is a uni-directional association. The automotive controller has to know the scheduler that is available. However, on the other hand, scheduler is not aware of the controller’s implementations.
  - Association between the objects “Scheduler” and “Service Offering” is navigable in both directions. A scheduler has to know the offerings assigned to it. In addition, a service offering has to know the scheduler it has been placed in.

Instructor Notes:

Here, based on the event like clicking on maximize, minimize or close of the window, appropriate method of window will be invoked. So window class depends on the event class.

Other examples of dependencies are relationship between a class and related exceptions: based on the kind of exception raised, appropriate actions need to be taken in the class behavior.

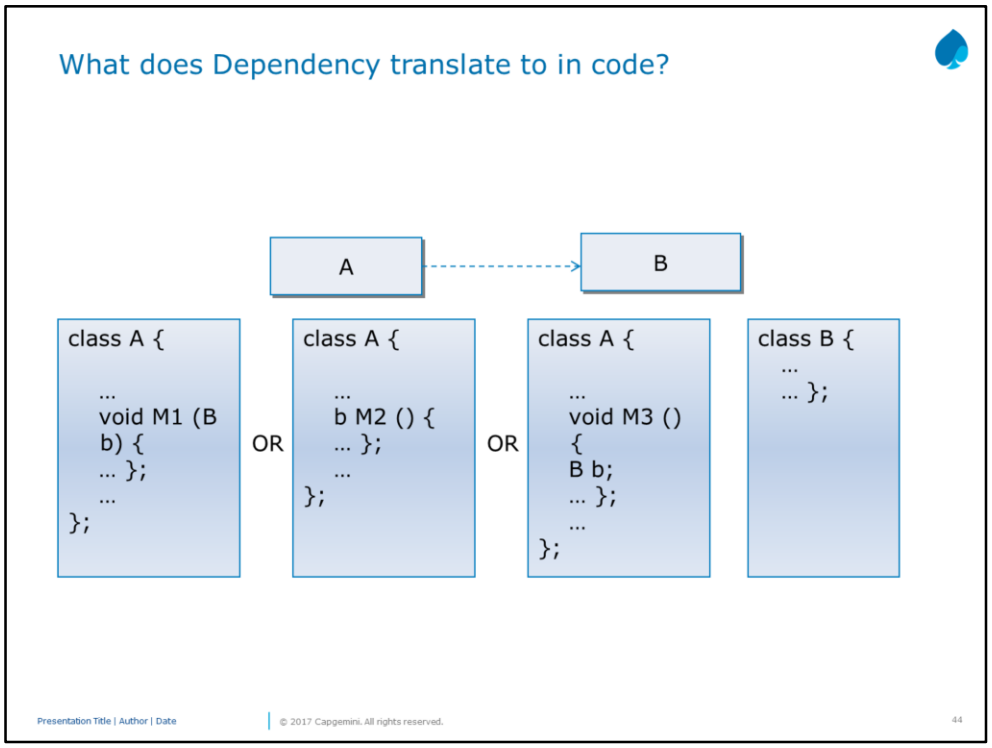


UML Relationships: Dependency:

- A **dependency** relationship is a weaker form of relationship showing a relationship between a **client** and a **supplier** where the client does not have semantic knowledge of the supplier.
- A dependency relationship denotes a semantic relationship between model elements, where a change in the supplier may cause a change in the client.
- The dependencies are denoted as dashed arrows with arrow head pointing to the independent element. In the example in the above slide, the structure and behavior of the Window Class is dependent on the structure and behavior of the Event Class.

Instructor Notes:

Emphasize on the non-structural part of the dependency relation. The fact that dependency reflects not as an attribute but within the methods (as a parameter, return or local variable) is important to note.



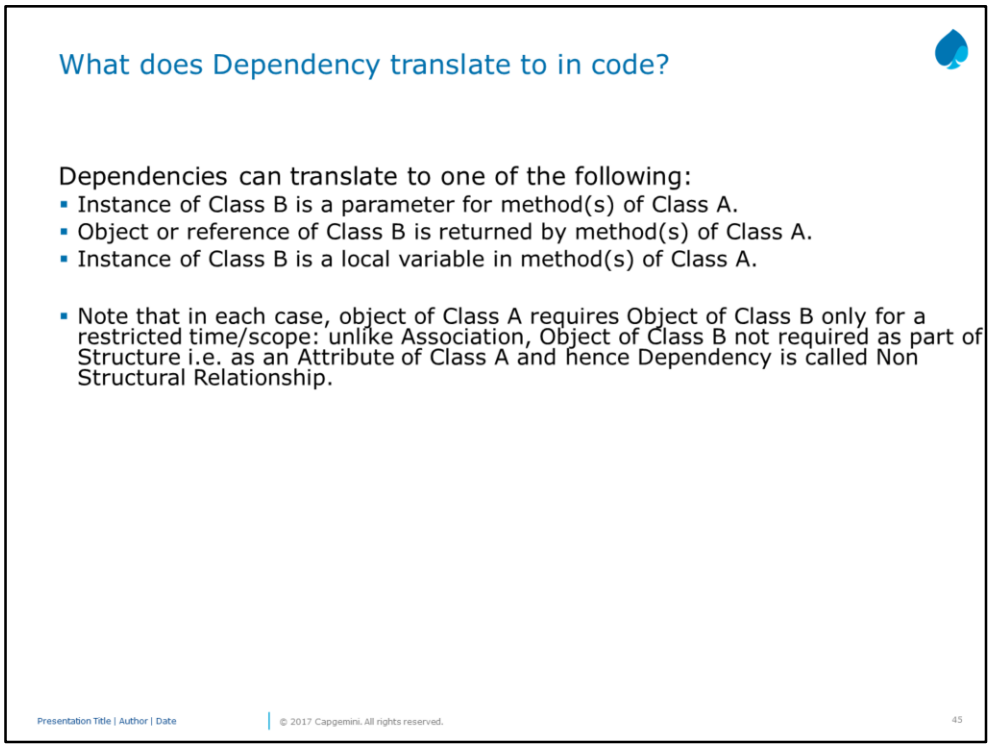
UML Relationships: What does Dependency translate to in code?

- In a dependency relationship, an instance of the independent class (B) will be used in the dependent class (A) in one of the following manners:
  1. Instance of B can a parameter to one or more methods of Class A.
  2. Instance of B can be returned by one or more methods of Class A.
  3. Instance of B can be a local variable in one or more methods of Class A.

- Note that dependency is non-structural, that is, instance of Class B does not come as an attribute of Class A and hence not part of the structure of Class A. Class A is aware of existence of Class B only when the concerned method is called. The relationship is temporary in nature too…once the method invocation is completed, Class A need not maintain information about Class B.

Instructor Notes:

Emphasize on the non-structural part of the dependency relation. The fact that dependency reflects not as an attribute but within the methods (as a parameter, return or local variable) is important to note.



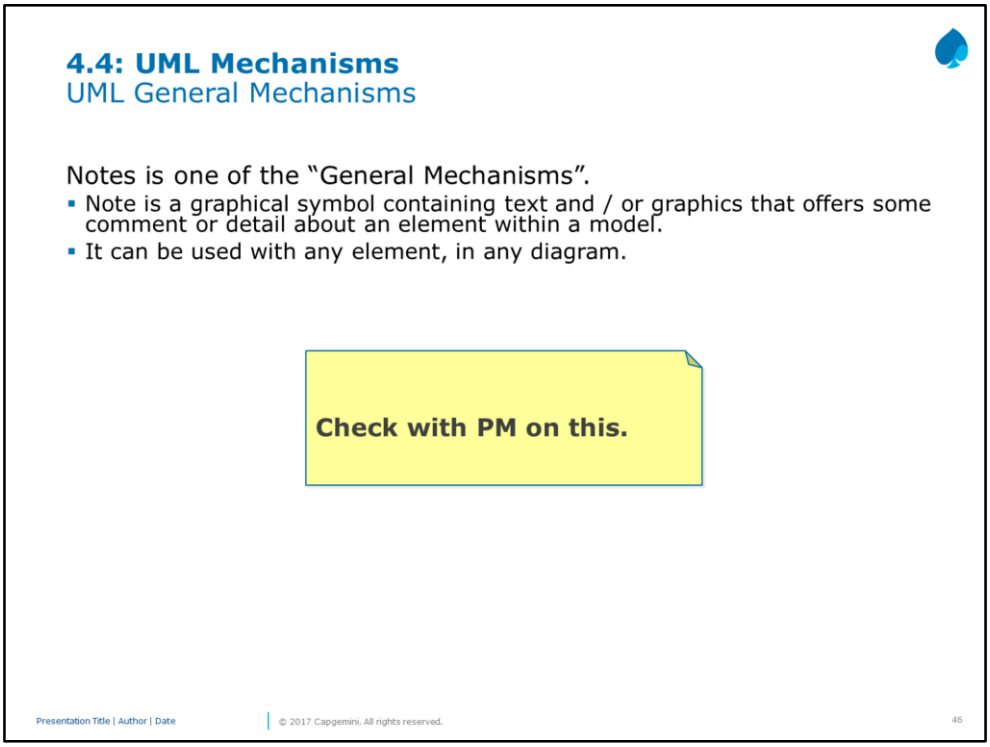
UML Relationships: What does Dependency translate to in code?

- In a dependency relationship, an instance of the independent class (B) will be used in the dependent class (A) in one of the following manners:
    1. Instance of B can a parameter to one or more methods of Class A.
    2. Instance of B can be returned by one or more methods of Class A.
    3. Instance of B can be a local variable in one or more methods of Class A.

- Note that dependency is non-structural, that is, instance of Class B does not come as an attribute of Class A and hence not part of the structure of Class A. Class A is aware of existence of Class B only when the concerned method is called. The relationship is temporary in nature too…once the method invocation is completed, Class A need not maintain information about Class B.

Instructor Notes:

Notes is to UML Constructs what Comments and Documentation are to Programs.



UML Mechanisms: General Mechanisms:
The General Mechanisms include:

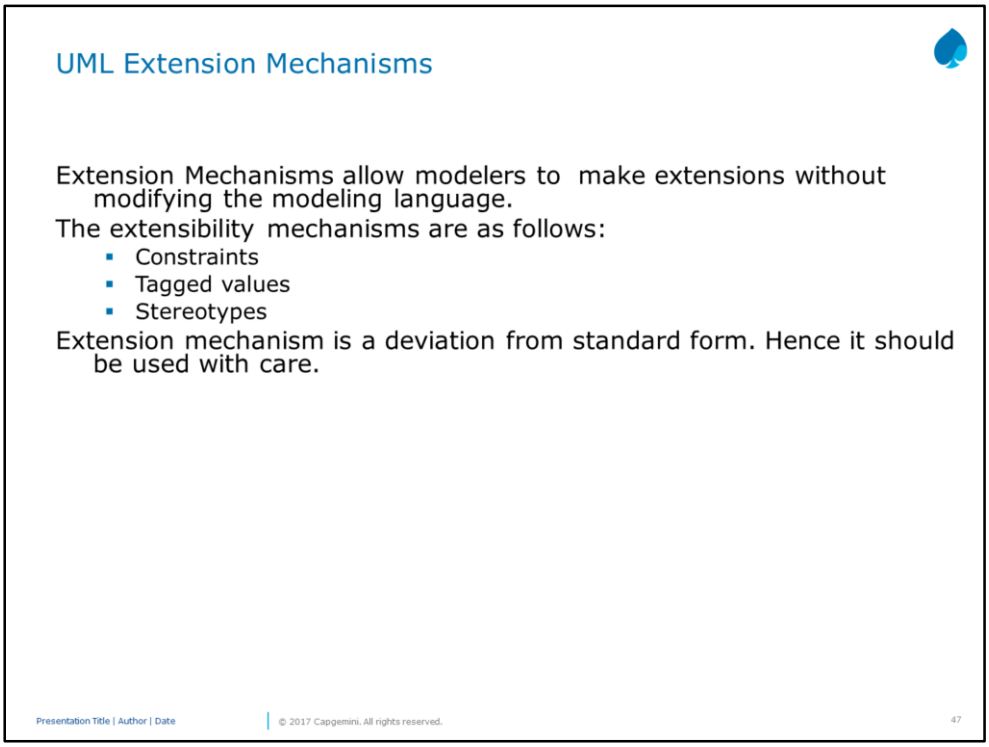
Adornments such ash numbers to indicate multiplicity and labels for roles

Specifications such as visibility and persistence

Apart from the above, Notes can be used to provide comments in the diagrams. They can be attached to any element in any diagram to give more details or clarifications, as required.

Instructor Notes:

Of the three, stereotypes are frequently used and needs more focus.

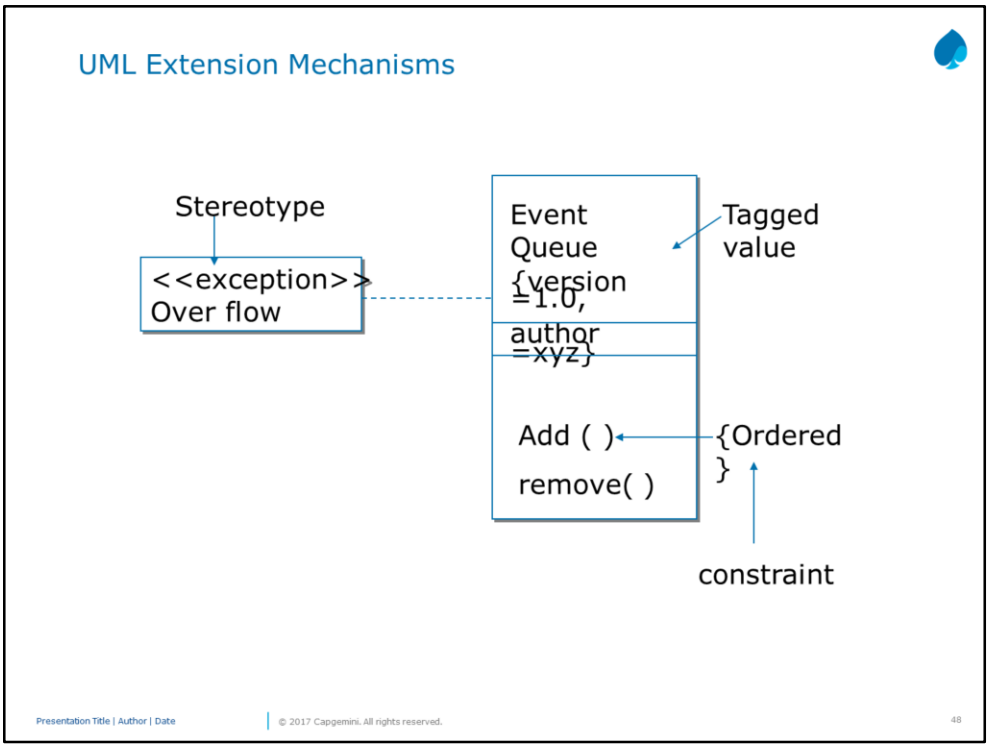


UML Mechanisms: Extension Mechanisms:

- UML, being a language, has a fixed set of notations with associated notation and semantics. However, users may need constructs that are not provided for in UML. Hence UML users may tend to add their own notations. So UML provides for ways by which extensions can be made to existing modeling language.
- These extension mechanisms of constraints, tagged values, and stereotypes are meant for Customizing and Extending UML. However, they need to be used with care and well documented – after all, they still are deviations from the standards!

Instructor Notes:

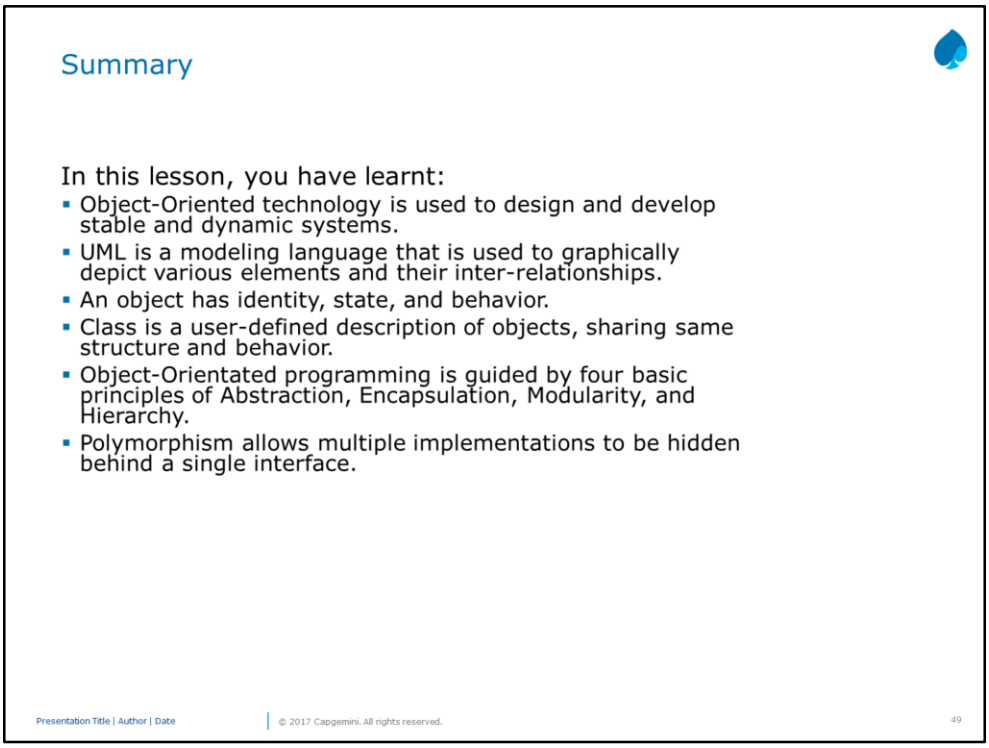
Encourage participants to come up with more examples for these extension mechanisms



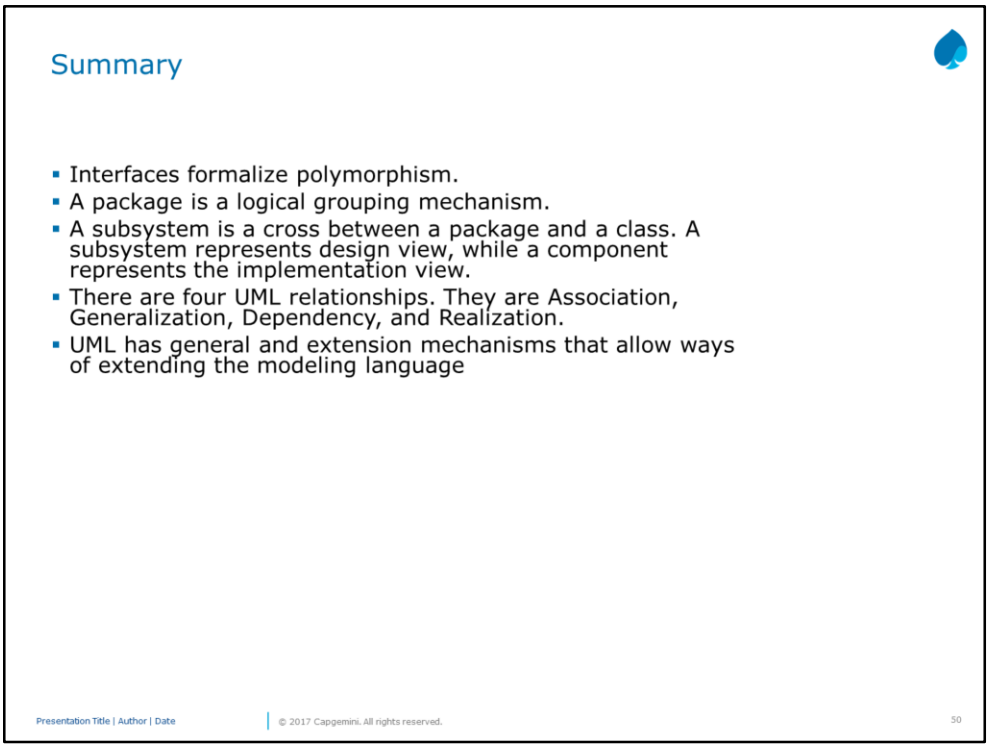
UML Mechanisms: Extension Mechanisms:

- In this class diagram, the stereotype “exception” indicates that Overflow is a class meant for handling exceptions that occur in the Event Queue class.
- The tagged values for Event Queue give information about the version and author of the Event Queue.
- The constraint ordered associated with the method Add() of Event Queue denotes that elements are added to the Event Queue based on some ordering.

Instructor Notes:



Instructor Notes:



Instructor Notes:

Answers for Review Questions:

Answer 1: Abstraction, Encapsulation, Modularity and Hierarchy

Answer 2: Option 1, Option 2

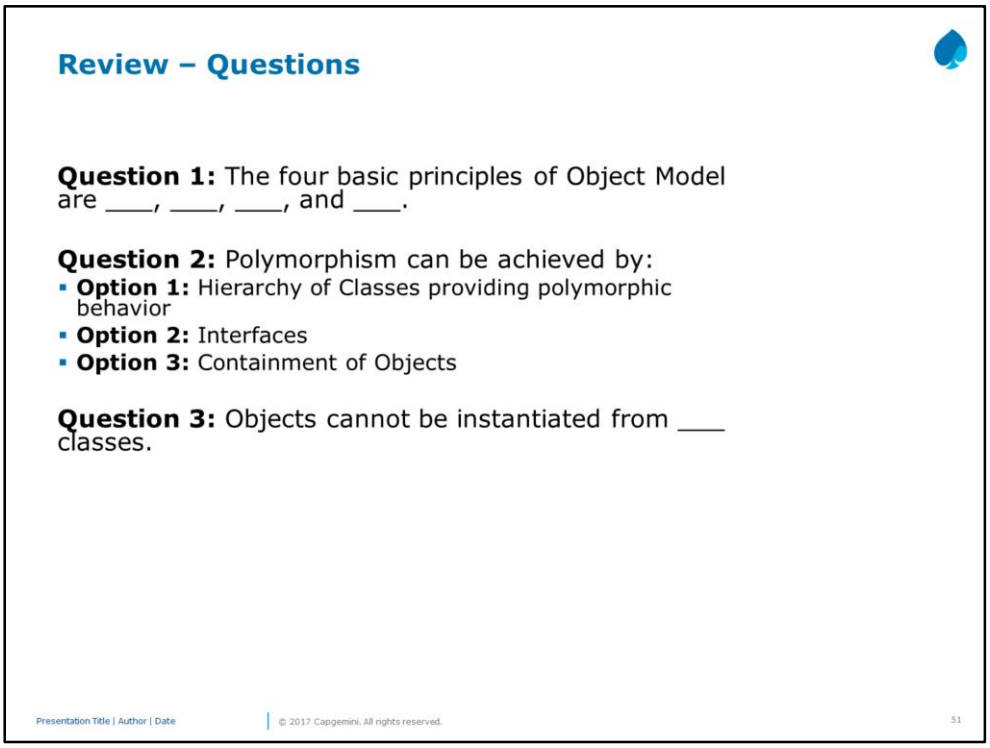
Answer 3: Abstract

Answer 4: Package

Answer 5: False – it is the other way round

Answer 6: True

Answer 7: Stereotype, tagged values and constraints



Instructor Notes:

Answers for Review Questions:

Answer 1: Abstraction, Encapsulation, Modularity and Hierarchy

Answer 2: Option 1, Option 2

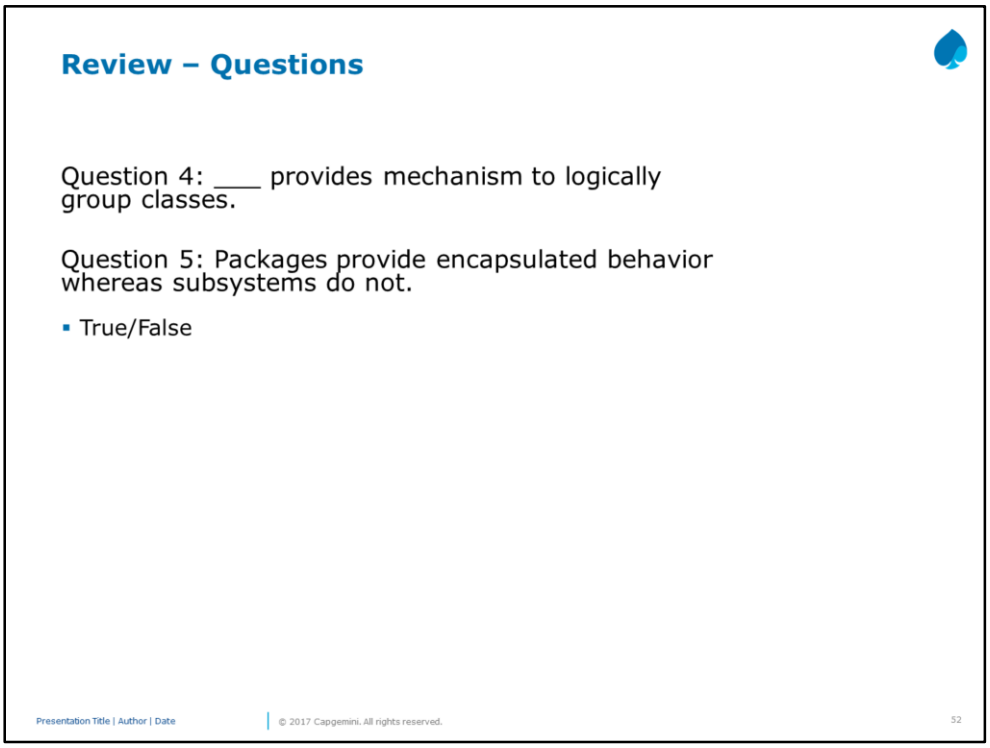
Answer 3: Abstract

Answer 4: Package

Answer 5: False – it is the other way round

Answer 6: True

Answer 7: Stereotype, tagged values and constraints



Instructor Notes:

Answers for Review Questions:

Answer 1: Abstraction, Encapsulation, Modularity and Hierarchy

Answer 2: Option 1, Option 2

Answer 3: Abstract

Answer 4: Package

Answer 5: False – it is the other way round

Answer 6: True

Answer 7: Stereotype, tagged values and constraints