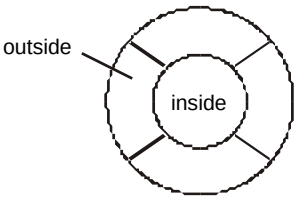
Classes and Relationships

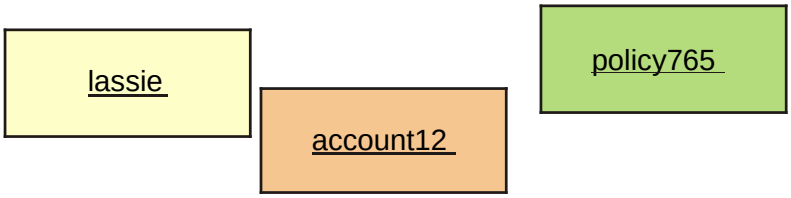
- Encapsulation
- Navigability and Multiplicity
- Composition and Aggregation

Review: What is an Object?

- An object is a set of state and the behaviors that act on them
- Objects have an inside and an outside



Object Name	Sample State	Sample Behavior
lassie	weight	bark
account12	balance	debit
policy765	amount	payOut



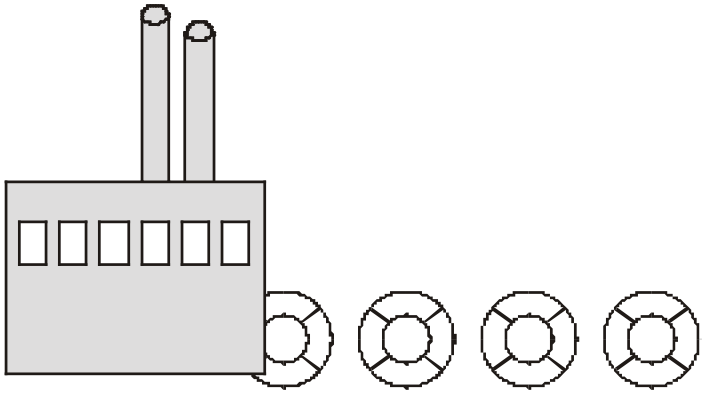
1 - 2

As you've already learned, an object consists of data (state) and methods (behavior) and that each object is separate for all others.

You also learned the UML symbol for an object, which is a box with the object's name underlined.

Review: What is a Class?

- A class is blueprint for objects
- Most object-oriented languages require you to define the class before creating objects



1 - 3

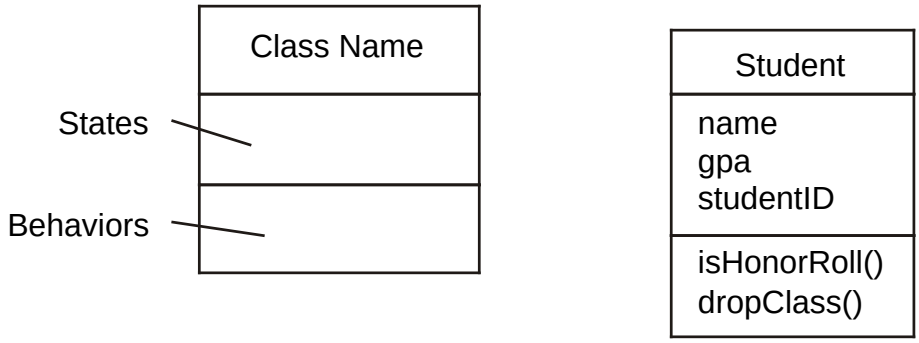
You've also learned that classes define the state and behavior for sets of objects and that in most object-oriented programming languages, the class acts as a factory for creating objects.

In this chapter, we will now learn more about modeling classes in UML and how to define relationships between classes.

It's important to note that classes are really just a modeling technique -- at runtime, it's the objects that have state, behavior and relationships and implement the software system.

UML Class Diagrams

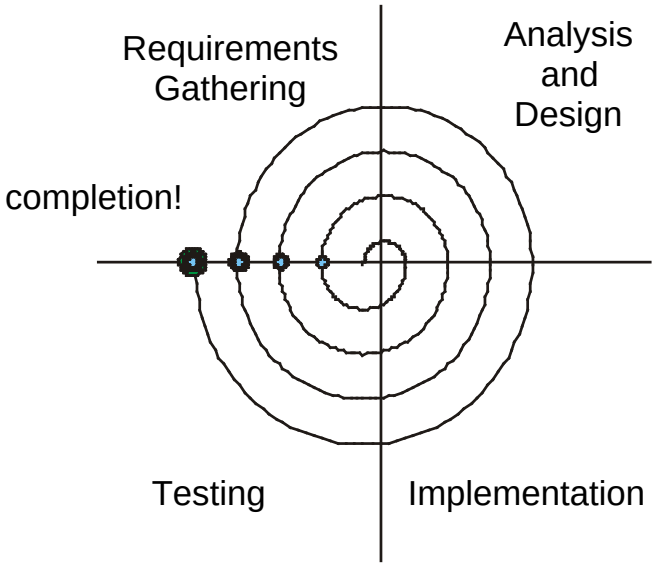
- UML provides the class diagram to let you model the structural (static) aspects of your software system
- The class diagram is probably the most well known and often-used UML diagram



The class diagram lets you define the blueprint for objects of that class and includes compartments for the class name, states and behaviors. Note that the state and class name are nouns, while the behavior names usually include a verb and have parenthesis.

Class Diagrams in the Unified Process

- You can create class diagrams both during analysis (domain modeling) and design



1 - 5

Class diagrams are an integral part of any iterative process that uses UML. We perform domain analysis to create class diagrams that help us model the entities in our software system. We then refine the class diagrams to create more detailed, design-level diagrams that we can then translate into actual software.

Analysis and Design Level Class Diagrams

- Analysis-level diagrams generally contain less detail than design-level diagrams

Analysis (Domain) Diagram

Student
name gpa studentID
isHonorRoll() dropClass()

Design Diagram

Student
name:String gpa:double studentID:int
isHonorRoll():boolean dropClass(classNo:int):void

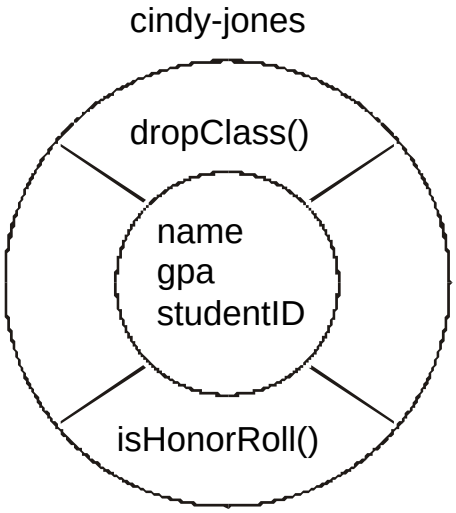
During analysis, we are trying to understand the system, so it's not generally useful to spend much time thinking about implementation details, including the language types for states and the parameter lists for methods.

On the other hand, design-level diagrams should be detailed enough so that we can write (or generate) software that implements the model. Therefore, design-level diagrams need to have sufficient detail.

So during an iterative process, you first create an analysis model for the iteration, then flesh it out to create the design model. You should note however, that there's not always a one-to-one list of classes in the two models -- often while transforming the analysis model, you discover other required classes or determine that multiple domain classes should be combined.

Encapsulation

- Encapsulation is one of the three pillars of object-orientation
- Encapsulation decreases the brittleness of your software system



1 - 7

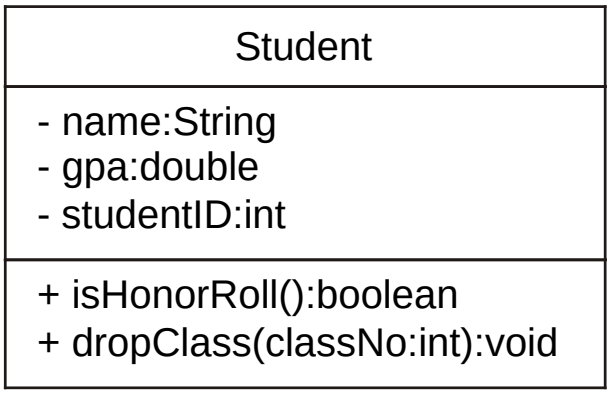
The more implementation details you can hide, the better, since that guarantees that the outside world cannot depend on the details (i.e. you can change the details without worrying that something outside of the class will break). Of course, a completely encapsulated class is of no use, so you will need to find a balance between encapsulation and actually getting something done.

A general rule of thumb (with many exceptions!) is that you should make state private and methods public. This is not a hard-and-fast rule, however -- it's quite common to have private methods that are only called from within the class.

Public, Private and Protected

- Most object-oriented programming languages support public and private access
- Some programming languages support additional access specifiers

- private
+ public
# protected



Private is the access specifier that most languages use to indicate the highest degree of encapsulation -- private parts of a class can only be accessed from within the class itself. We use the minus sign in UML to indicate private access.

Public access means that any code can access the state or behavior. UML uses the plus symbol to indicate public access.

Some languages support additional levels of encapsulation, for example, Java and C++ both support "protected", which is like private, except that subclasses can also access the state or behavior. Protected access breaks encapsulation to a degree, since it lets subclass methods directly access things in the superclass, but it can lead to better performance.

Java Class With Encapsulation

```
1    public class Student
2    {
3      private String name;
4      private double gpa;
5      private int studentID;
6
7      public boolean isHonorRoll()
8      {
9        if (gpa > 3.0) return true;
10       else return false;
11     }
12     public void dropClass(int classNo)
13     {
14       // drop the class
15     }
16   }
```


Here we show a Java class that implements the design-level class diagram shown earlier. Note how the states are private (lines 3 to 5), while the behaviors are public (lines 7 to 15). Note also how the types of states and method parameters in the design-level class carry over into the Java class.

Also note line 14, which is a Java comment line. In a complete implementation, we would write Java code within the dropClass() method, but here we just show a comment indicating what needs to be done.

Get/Set Methods

- Though we generally make state private, the outside often needs to access an object's state
- To provide controlled, encapsulated access, you can write get/set methods

```
1    public class Student
2    {
3      private String name;
4
5      public String getName()
6      {
7        return name;
8      }
9      public void setName(String s)
10     {
11       name = s;
12     }
13     . . .
14   }
```

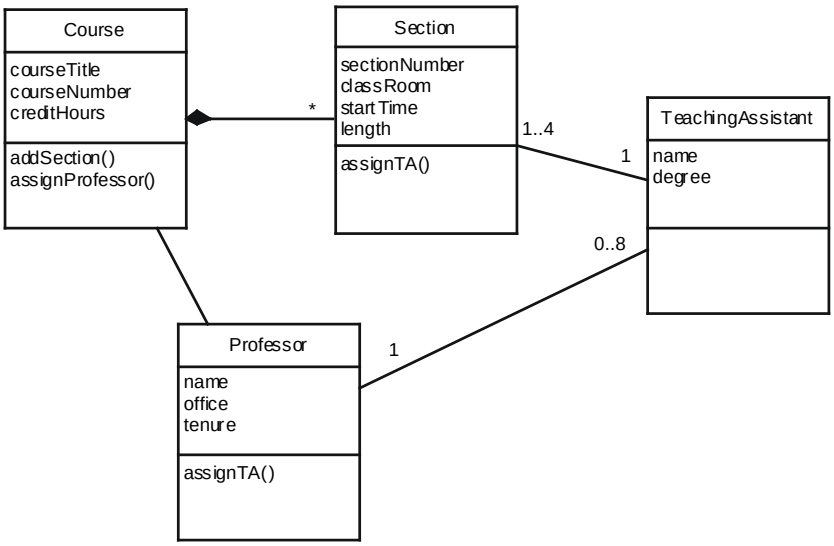
Here we show a portion of the Student class, showing how we can use get/set methods to provide controlled access to state. Note how the get/set methods are public, while the state itself remains private. Though we didn't have enough room to show all of the code, the remaining states and get/set methods follow the same pattern.

The get/set methods are sometimes referred to as 'accessor' and 'mutator' methods. Furthermore, the naming pattern shown here is very common in Java classes referred to as JavaBeans.

One other note -- it's common to omit get/set methods in class diagrams, even design-level diagrams. That's because it's such a common facility, most developers just assume that they need to exist for each state item.

Introduction to Relationships

- Most business objects need to reference other objects in the same domain
- We can model references using **relationships** in a class diagram

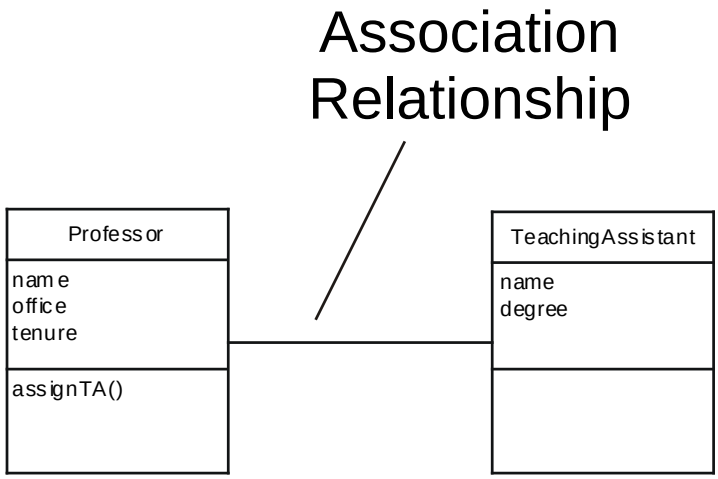


1 - 11

The UML provides a rich set of symbols and annotations so that we can model the relationships between classes. It is important to note however, though we model relationships between classes, at runtime, the relationships are actually between objects of the various classes.

Associations

- An association is a simple relationship that indicates that objects in two classes are related in some way
- We will examine several refinements to the simple association

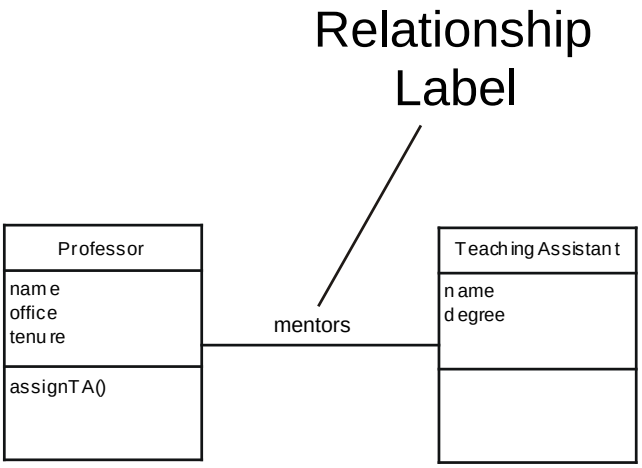


In UML, to define a simple association, you draw a line between the classes. In the figure on this page, we are indicating that there's some relationship between a Professor and a TeachingAssistant.

In an analysis-level diagram, it's common to leave the association unadorned, unless during domain analysis, you can discover additional attributes about the relationship. We will cover some refinements on the next few pages.

Relationship Labels

- For readability and documentation, you can label relationships
- Often, labels are redundant and unnecessary

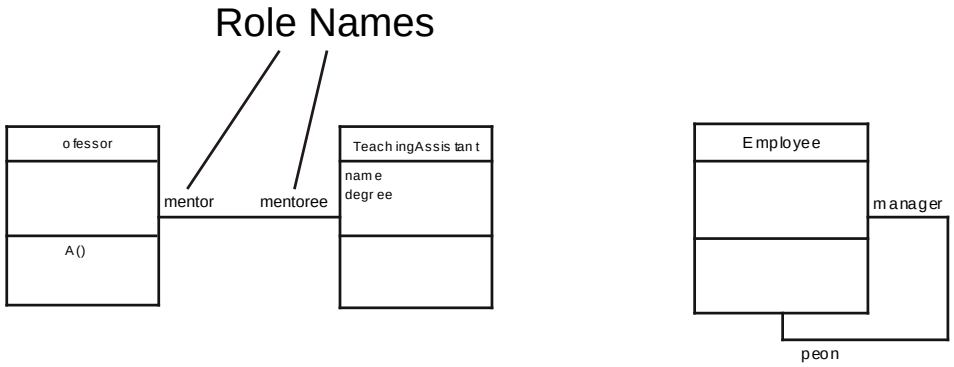


Labels can convey useful information, but in many cases, they are redundant. For example, here one could argue that it's obvious what the relationship is between a Professor and a TeachingAssistant, so it would be OK to omit the label.

Note that we read relationships from left to right. If for some reason you cannot layout the diagram in left-to-right fashion, you can decorate labels with an arrow that shows the direction of the label.

Relationship Roles

- Instead of labeling a relationship, in some cases it's more informative to label the roles that source and destination objects play
- In a design-level model, the role names are often used to name the variables that implement the relationship

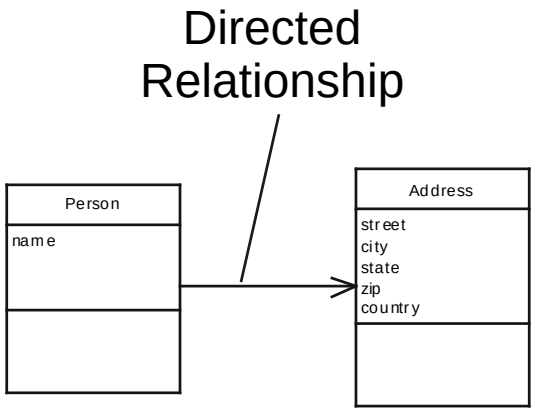


As you saw with relationship names, often role names are obvious and redundant and just clutter the diagram.

One case where they are useful is in a self-referencing relationship like the one shown on the right. The role names make it clear what we are trying to model with the self-relationship.

Relationship Navigability

- Navigability refers to objects of a class being to directly send messages to objects of the referenced class
- Navigability implies that the source object keeps a reference to the destination object
- Note that it's common to omit navigability in an analysis-level diagram



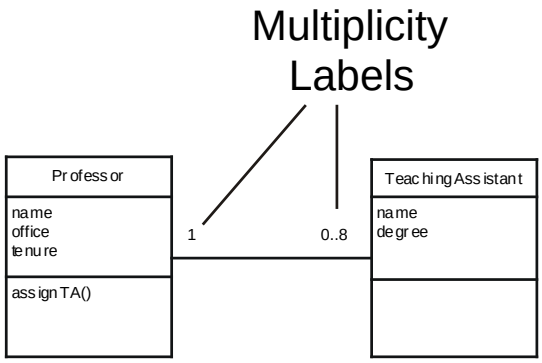
1 - 15

On a relationship line, an arrowhead indicates navigability from the source to the destination. Lack of any arrowheads implies navigability in both directions.

In this model, it's important to find an Address, given a Person, but it's not important to be able to find a Person given an Address.

Relationship Cardinality

- Cardinality (a.k.a multiplicity) refers to the number of references from the source object to the destination
- Multi-valued references imply that the source object maintains a list or collection of references to the destination objects
- Note that it's common to omit cardinality in an analysis-level diagram



Multiplicity labels are an optional annotation that let you model the count of references on each side of the relationship. In this case, each TeachingAssistant "belongs" to a single Professor, and each Professor can mentor up to eight TeachingAssistants.

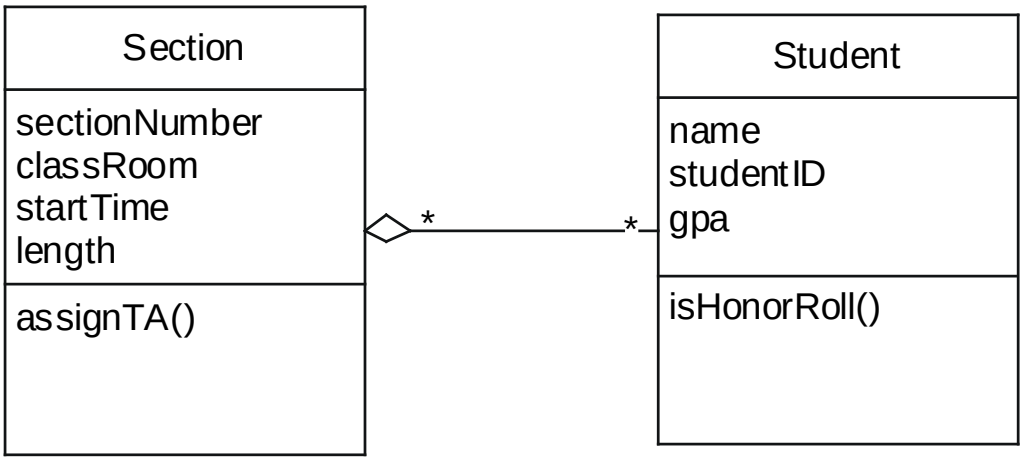
Cardinality Options

- UML provides several ways to indicate cardinality

Label	Meaning
1	Exactly one
0..* or *	Many
1..*	One or more
3..4 or 8	Exact number
0..2, 4..6, 8..*	Complex relationship, in this case any number except 3 or 7

Aggregation

- Aggregation is a stronger form of a simple reference
- Aggregation implies that the destination object(s) are part of the source object

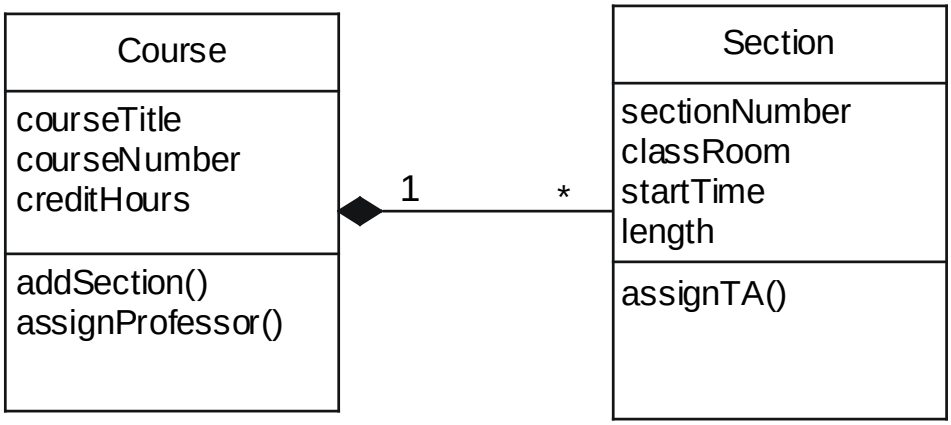


Aggregation indicates a whole/part relationship, which is a stronger relationship than a simple association. In UML, we use a diamond arrowhead on the 'whole' side of the relationship.

In this model, we are saying that a Section is made up of Students. Furthermore, this model says that it's possible for a Student to exist, but not be enrolled in any Section -- this notion of "standing alone" is what differentiates aggregation from composition as described on the next page.

Composition

- Composition is an even stronger relationship that implies that the parts cannot "live" without the whole and that the parts cannot "live" in any other whole



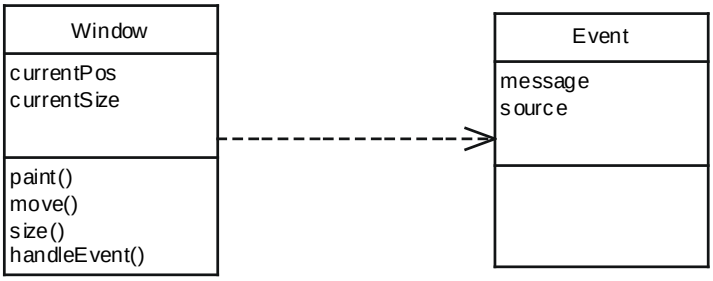
Like aggregation, composition indicates a whole/part relationship, but with the added proviso that the "parts" cannot exist without the "whole". In UML, we use a filled diamond arrowhead on the "whole" side of the relationship.

This model, therefore indicates that it doesn't make sense for a Section to exist without being assigned to a Course and that a given Section cannot belong to any other Course.

If the difference between aggregation and composition is bit hazy, don't worry -- there's not wide agreement in the UML community about them. In fact, according to Martin Fowler, the author of "UML Distilled", even the Three Amigos don't see exactly eye-to-eye on the subject!

Dependency Relationships

- A dependency relationship is a weak relationship that indicates that changes to one class may affect another, but that source objects don't maintain a reference to the destination objects
- In most cases, this relationship results from a method in the source class accepting arguments of type of the destination class

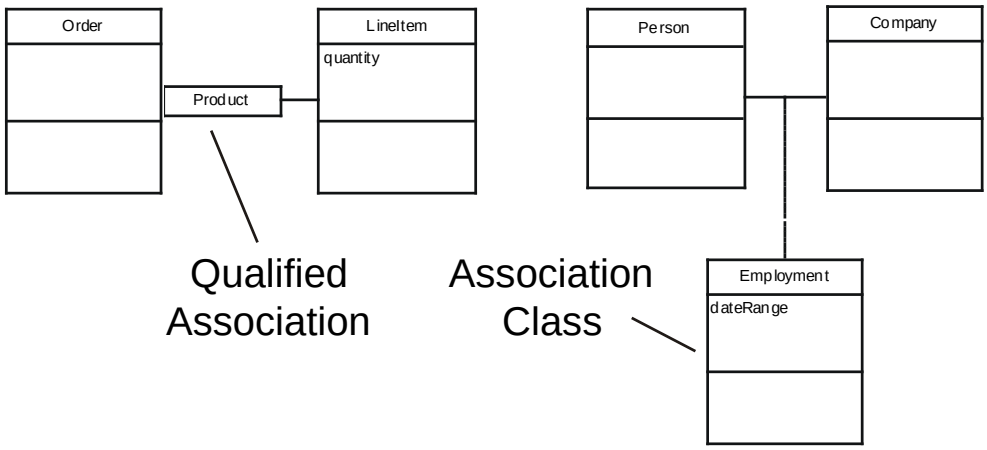


Dependency lets us model objects that in some fashion use objects of another class, but don't keep a reference to the destination object. UML uses a dashed line to indicate this dependency relationship, which is often a directed relationship (one-way).

In this model, note that the Window class has a method named handleEvent() that most likely accepts an argument that's an Event object. So if we update the Event class, it's possible that those changes may break the Window class -- that brittleness is the property we model with the dependency relationship.

More on UML Relationships

- **Qualified associations** let you model the scenario where it's possible for a source class to lookup objects of another class given additional information
- **Association classes** let you model associations that themselves have attributes



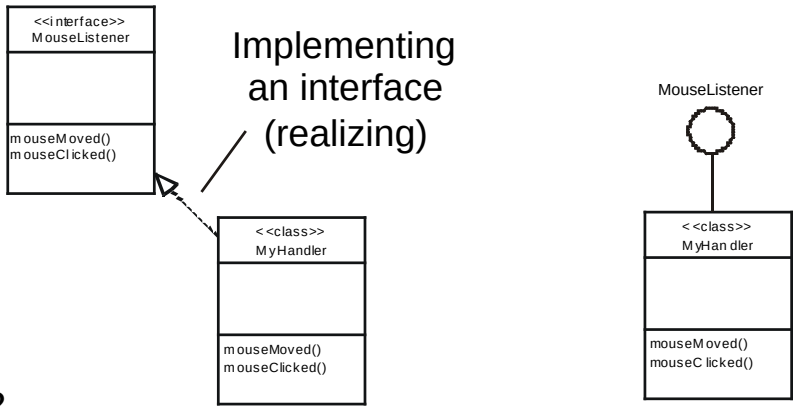
These two UML notations are less common than the other relationships, but can be useful in some cases.

Qualified associations let you model the notion of a hash table (sometimes known as an associative array), that consists of key-value pairs. Given the key, you can lookup the value. In the model shown here, we can uniquely identify a given line item for an Order given the Product object.

Association classes let you model complex associations where the association itself has state. In this case, we model an Person's tenure at a Company -- the association state is the date range. There is a twist to association classes however -- by definition, in UML, there can be only one instance of the association class for each pair of related objects. So in this case, a Person can have only one stint at a Company, which is probably not realistic. If that restriction is unappealing, you can always model using a standard class to contain the association properties, inserting it between the otherwise related classes.

Introduction to Interfaces

- An interface is a set of behaviors with no implementation
- Architects use interfaces to define roles that classes can implement
- Many languages (e.g. Java) support interfaces as native syntax



Interfaces let architects define sets of behaviors that programmers can implement. They are common in toolkits for user interfaces. In fact, the model shown here is taken from Java's Swing user-interface toolkit, which lets you register listeners that the system notifies of significant events, in this case, the user's actions on a window on the screen.

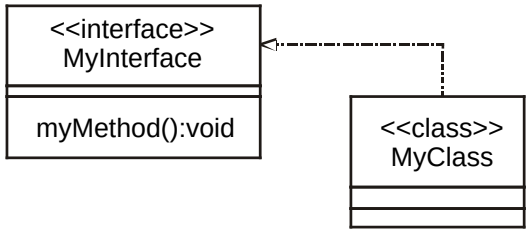
UML provides two notations for interfaces -- the one on the left shows details about the interface, while the one on the right uses a "lollipop" instead of a fleshed-out symbol for the interface, Lollipops are most useful for well known interfaces where the additional detail clutters the diagram.

By the way, even though we show them here, it's common to leave out the behaviors in the implementing class -- in other words, that behavior is implied by implementing the interface.

One more note: some languages, like Java and C++, define the notion of "abstract" classes, which are like interfaces except that abstract classes can have some methods with an actual implementation. Interfaces, by definition, can have no implementation.

Sample Interface in Java

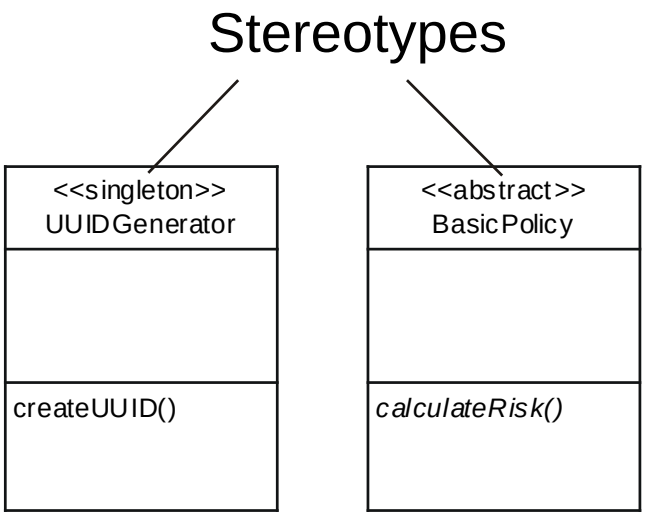
```
1    public interface MyInterface
2    {
3      public void method1();
4    }
5
6    public class MyClass implements MyInterface
7    {
8      public void method1()
9      {
10       // do something interesting
11     }
12   . . .
13   }
```



In Java, we use the keyword "interface" to define the set of behaviors and use the keyword "implements" on the class that provides the behaviors.

Stereotypes

- **Stereotypes** are a notation that let you extend the UML
- There are several commonly used stereotypes and you can define domain-specific stereotypes



1 - 24

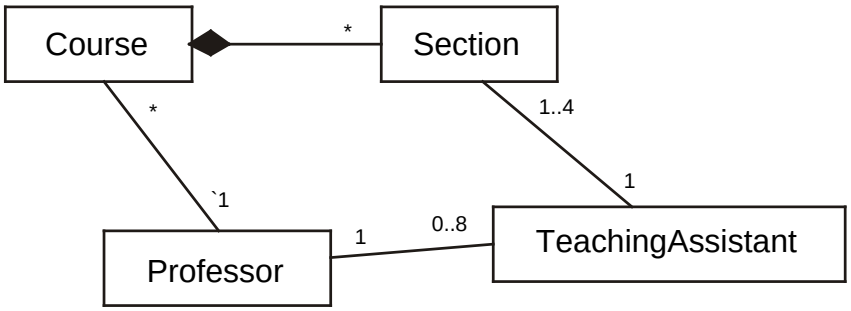
Stereotypes effectively let you add to the UML by creating new annotations. To indicate a stereotype, you use guillemets (chevrons) around the stereotype name.

The "singleton" stereotype is actually a design pattern from the well known Design Patterns book written by Gamma, et. al. We will cover more design patterns later in the course.

An abstract class is a class with some methods with no implementation (it's sort of like an interface). You indicate the abstract (unimplemented) methods by writing the method name in italics.

Elided Diagrams

- To make complex diagrams easier to read, you can **elide** diagrams by omitting well known information



Some OOA/D tools let you elide diagrams by temporarily hiding compartments, for example, hiding the state and behaviors of a class and showing only the class name.

In addition, during analysis, you may first created elided diagrams and then flesh them out as you learn more about the domain.

Review Questions

- Describe how you would use class diagrams in differently in an analysis-level diagram versus a design-level diagram.
- Describe how composition and aggregation are stronger relationships than simple associations.
- What is the difference between an interface and a class?

Chapter Summary

In this chapter, you learned:

- How to model classes at both the analysis and design levels
- The different ways to model relationships between classes