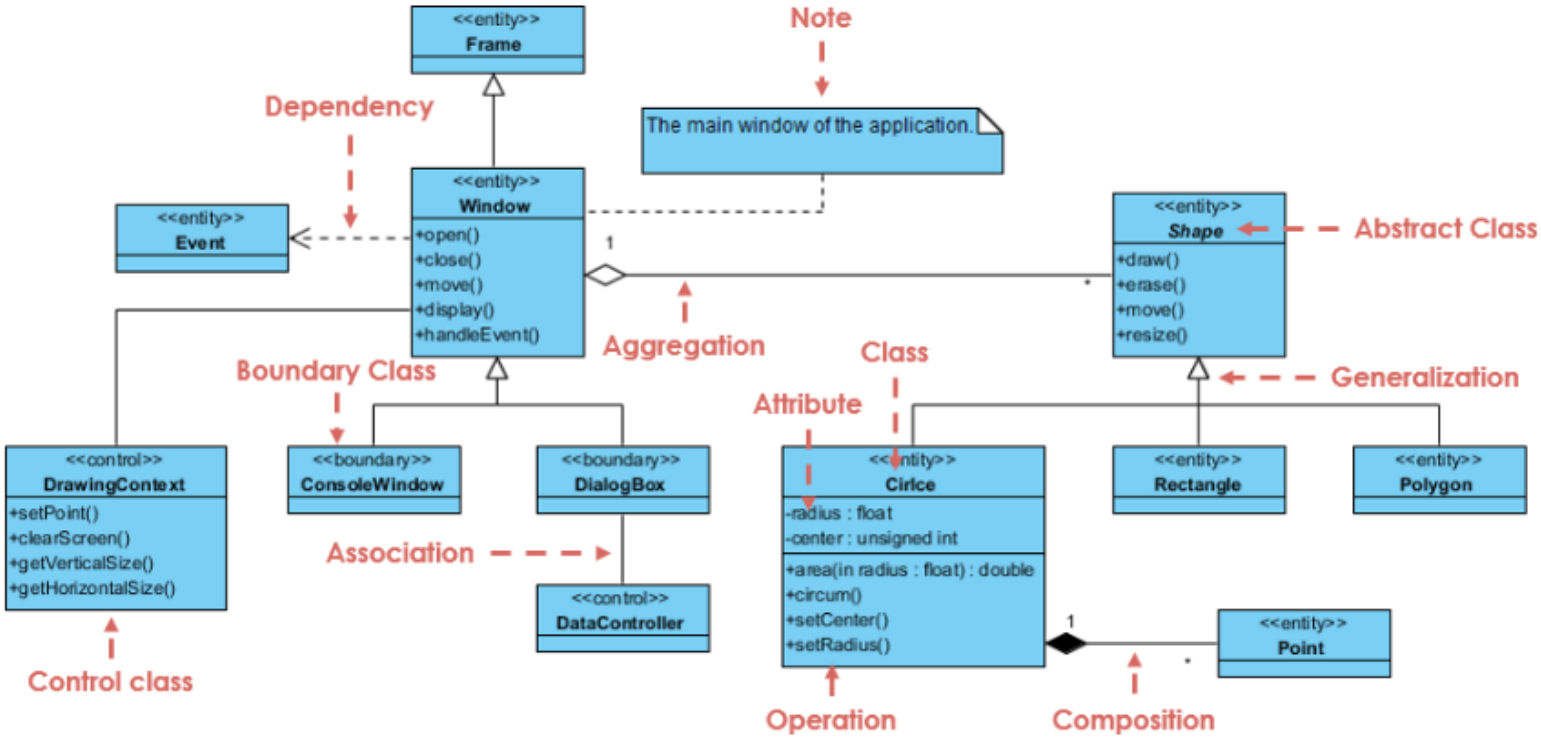
Chapter Two

UML Review – class diagrams

Remember UML Diagrams

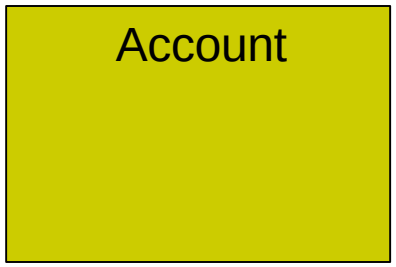
What does a Class diagram illustrate?



UML Review: Class diagrams

- A UML Class Diagram represents classes and their relationships to one another
  - **UML is not specific to Java**, so a Class Diagram really represents the generic concept of a class **without regard to what language implements the actual class**
  - The name of the class always appears at the top of a Class Diagram rectangle:

UML

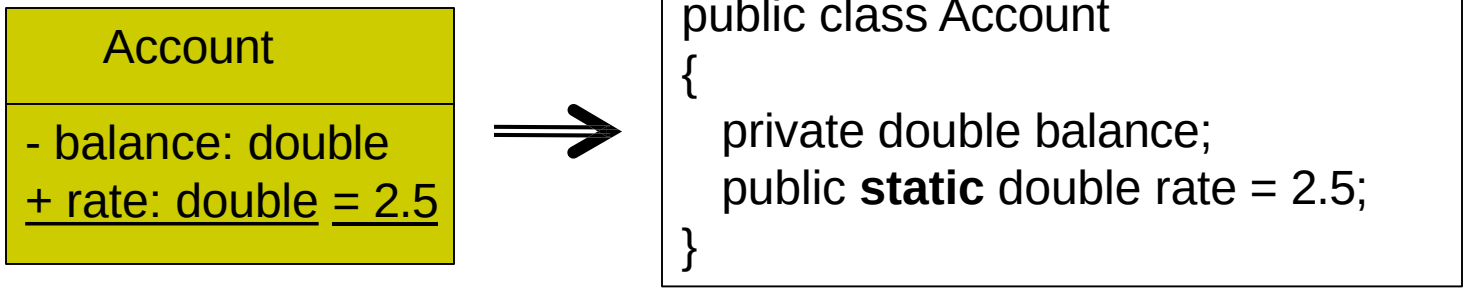


equivalent (Java) code

```
public class Account
{
}
```

Class diagrams: Attributes

- A Class Diagram can also show class *attributes* or *fields*
  - Syntax: [visibility] <attr_name> [: <type> [=default_value]]
  - Note: The **static** specification may be illustrated with an underscore or with *italics*



What if **balance** is declared as with protected visibility? How about package visibility?

Class diagrams: Operations

- A Class Diagram can also show class *methods* or *operations*
  - UML Syntax: [visibility] <name>([[in|out] param:type]*] [:<return_type>]

Account
- balance: double + rate: double= 2.5
+ deposit(amount : double) : void ~ getBalance() : double + setRate(intRate: double) : void

→

```
public class Account
{
   private double balance;
   public static double rate = 2.5;

   public void deposit (double amount)  {
                balance += amount;
   }
   /*package*/ double getBalance()  {
                return balance;
   }
   public static void setRate( double intRate ) {
                rate = intRate;
   }
}
```

Methods can also have protected or package visibility.

What is [in|out] about?

Repeat: UML Class Diagrams typically illustrate some type of *relationship* between classes

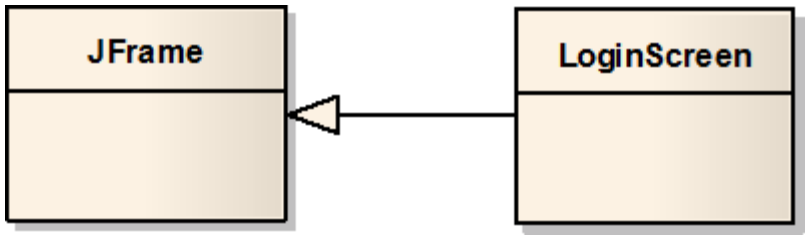
The easiest to master are those that illustrate a **permanent** (that is, **static**) relationship between classes

There are two basic categories of static relationships:

- Inheritance (2 forms of this)
- Dependency

Generalization is a form of inheritance that indicates that a class (LoginScreen) inherits behavior defined and implemented in another class (JFrame)

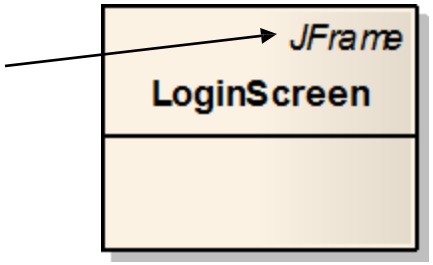
We also say: LoginScreen is a subclass of JFrame



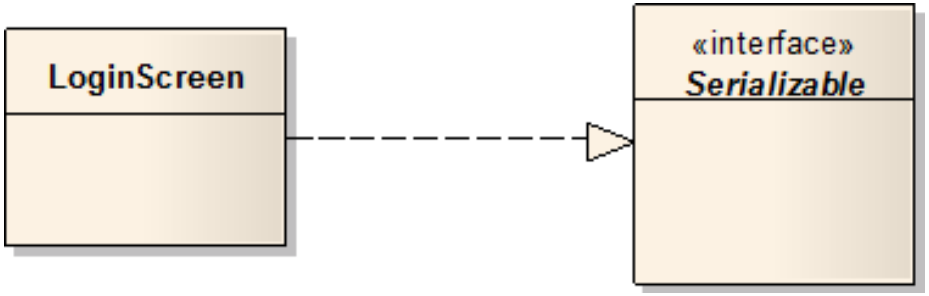
The connector is a solid line pointing to the class whose behavior is being inherited with a triangle (not an arrow!)

The connector "points" from the subclass to the parent class.

Reducing clutter: *Generalization* can be illustrated in the alternate notation shown if the parent class is not present in the class diagram



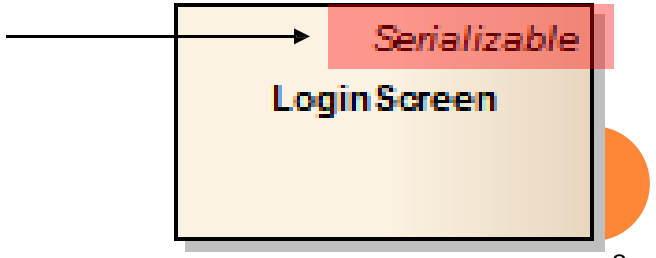
Realization is also a form of Inheritance that indicates that a class implements the behavior defined in an interface



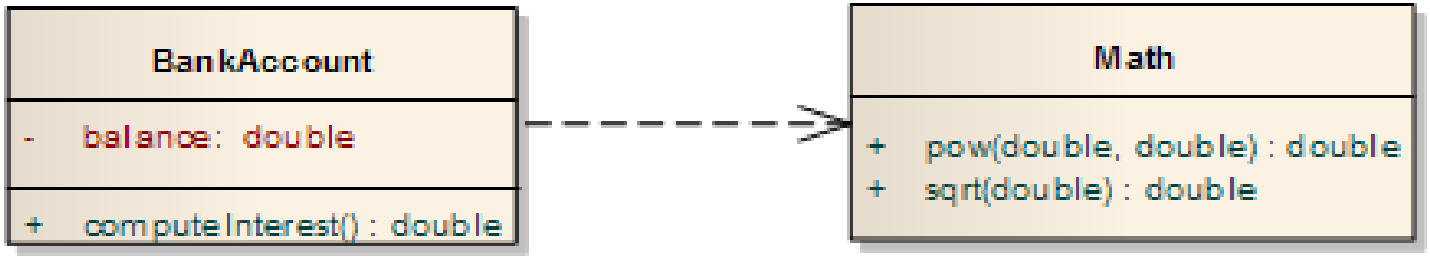
Here, the **LoginScreen** implements the behavior of the **Serializable** interface.

The connector is a dashed line pointing to the class whose behavior is being implemented with a triangle (not an arrow!)

Realization can also be illustrated in the alternate notation shown if the parent class is not present in the class diagram

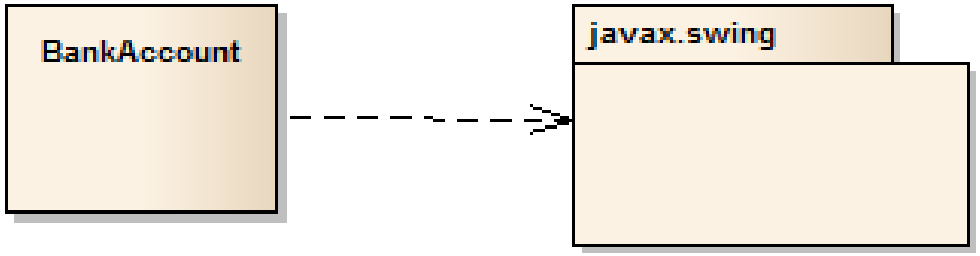


Class diagrams sometimes explicitly illustrate Dependency relationships of classes on other classes (or packages)



Dependency is indicated by a dashed-line **connector**, with the line originating from the dependent class and pointing (with an arrow) at the other class it depends upon.

The arrow may also point at a **package** (e.g. javax.swing), implying a dependency on many of the classes within that package.



There are additional relationships between classes that are not necessarily permanent – meaning they can change during program execution.

Dynamic relationships between instances of classes (that is, between objects), or between classes and objects (if a class is non-instantiable, as in a static class) are:

- Association
- Aggregation (2 forms of this)

The Association connector is used to indicate a run-time (dynamic) interaction between instances of classes

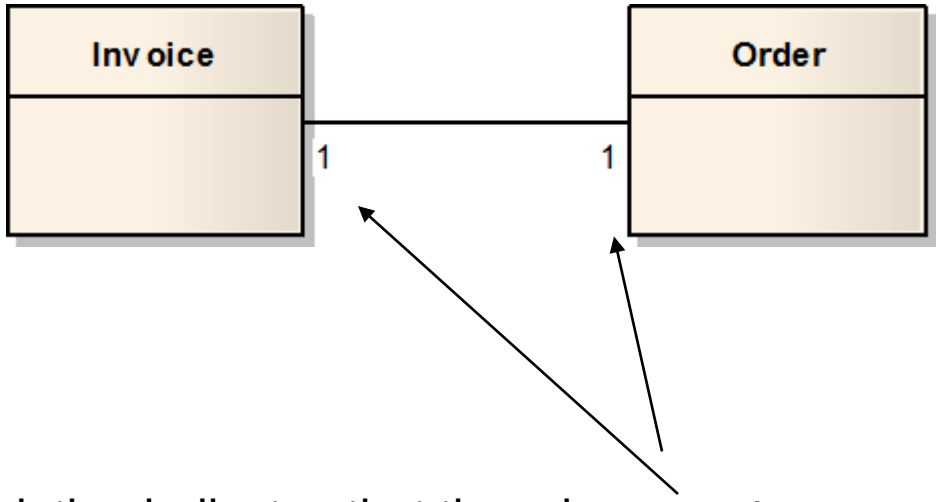
The single-line connector doesn't indicate anything specific – that is, it's an *unspecified association.*

About all we can say is that objects of these classes somehow interact when the application executes – perhaps an **Invoice** somehow accesses an **Order** (or vice versa).

In fact, the Invoice can create the Order at runtime (and later delete it); or the Order can create the Invoice at runtime – thus these relationships are dynamic.

We may not know enough to be more specific when we're **designing** our application – it may have to wait until we get into **implementation**

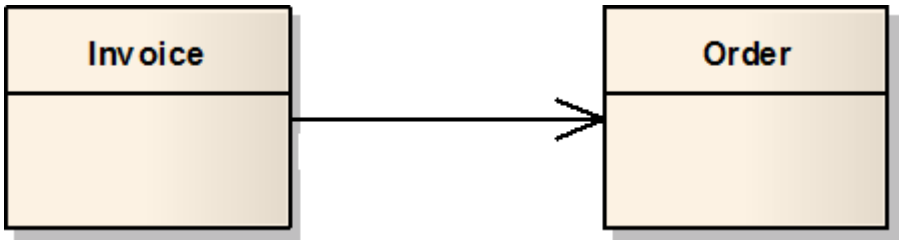
An Association can indicate *multiplicity* – that is, how many objects of one class interact with objects of the other



This Association indicates that there is a **one-to-one** correspondence between Invoice instances and Order instances; that is, for every Invoice object, there is one corresponding Order object, and vice versa.

If the numbers are absent, you can assume that there is a one-to-one correspondence.

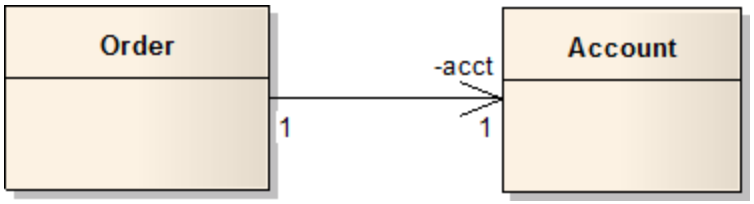
Associations can indicate *navigability* – that is, whether an object holds a reference to the other



Note that we've omitted the multiplicity here, so it's assumed to be one-to one

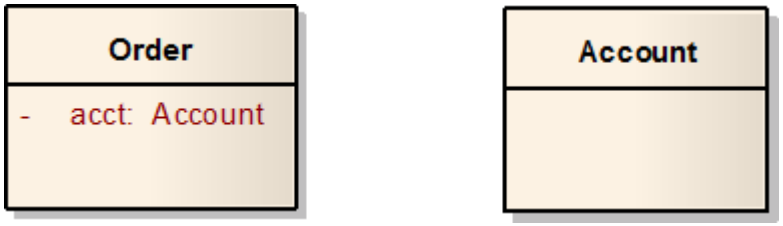
This one-way directed association indicates that an Invoice object holds a reference to a single Order object, but the **Order** does not hold a reference to an **Invoice**.

End Roles indicate that an association is maintained via a specific attribute defined within a class



Note that we've included the multiplicity here, but we could have omitted them since the correspondence is one-to-one

This one-to-one, one-way association with **end role acct** indicates that an **Order** object holds a reference to a single **Account** object via a private **Account** attribute named **acct**.

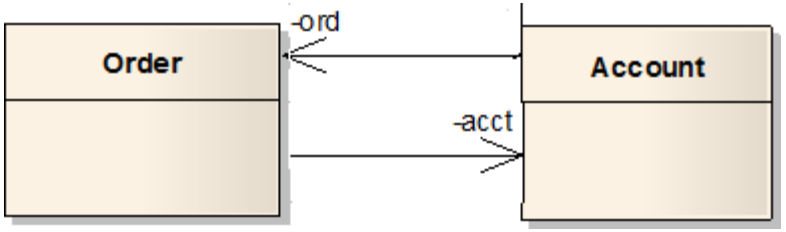


Here is another way of showing the same relationship.

Is this more or less illustrative?

Bi-directional navigability

- This **bi-directional association** indicates that an **Order** object holds a reference to a single **Account** object via a private **Account** attribute named **acct**, and that an **Account** object holds a reference to a single **Order** object via a private **Order** attribute named **ord**.

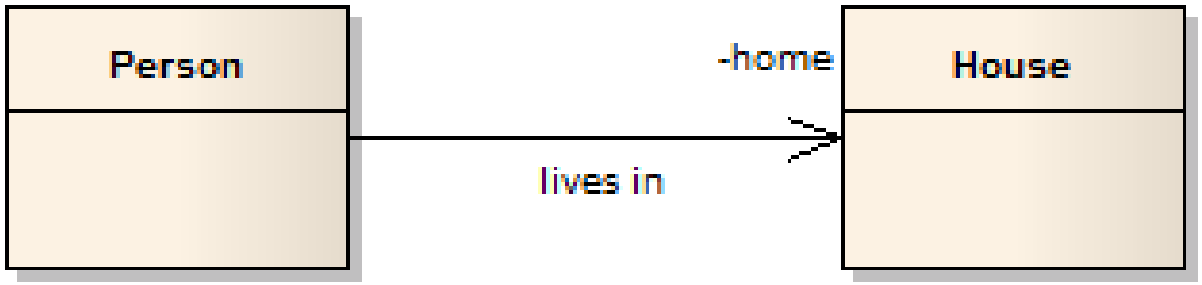


Note that we've omitted the multiplicity here, so it's assumed to be one-to one.

It also reduces clutter.

However, it is always best to avoid bi-directional associations, and instead to use two uni-directional arrows to illustrate this relationship.

An Association can be labeled to indicate the nature of the relationship

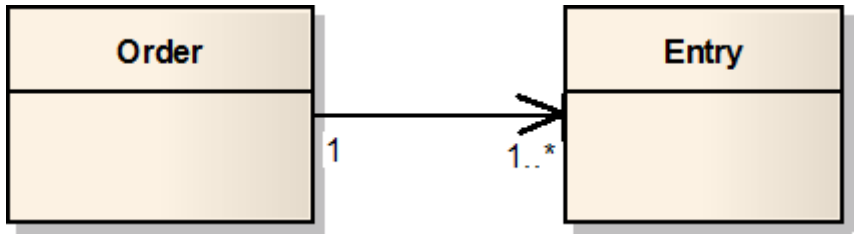


This diagram can be read as follows: “A **Person** lives in a **House**, which the **Person** refers to as home”.

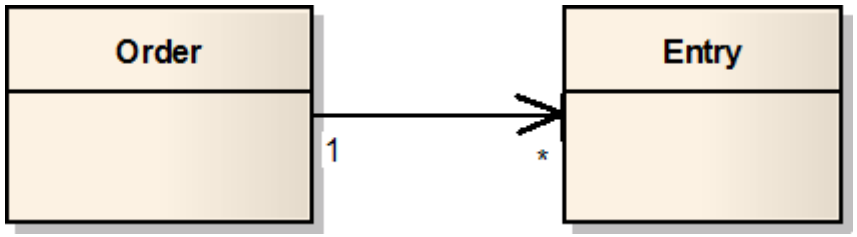
Again, we’ve omitted the multiplicity here, so it’s assumed to be one-to one.

It also reduces clutter.

Associations can indicate various degrees of multiplicity



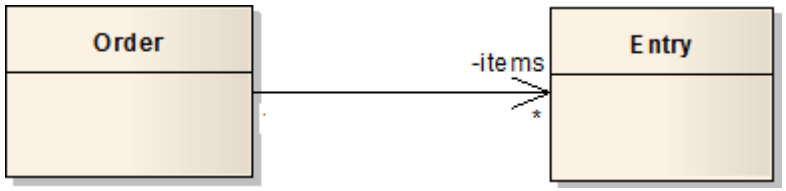
This Association indicates that for every **Order** object, there is at least one corresponding **Entry** object.



Here, any number (including zero) of **Entry** objects correspond to each **Order** object

We can omit the 1 next to the Order class to reduce clutter

Associations with multiplicity >1 implies that a reference is a *collection* of objects



This one-way association indicates that an **Order** object holds a reference to a **collection** of zero or more **Entry** objects via a private attribute named **items**.
However, the type of collection is not specified here.

The same association, where the collection is explicit.
However, this approach is not preferred.

Good UML Class Diagram practices

- Consider the following code:

```
public class Student {
        private String firstname;
        private String lastname;
        private Date birthdate;
        private String major;
        private int id;
        private double gpa;
        private List<Course> courses;
}
```

What does the UML class diagram look like?

Which class(es) should be explicitly shown? Why?

Which class(es) should appear as attributes? Why?

Advanced Associations:

Containment (aka Shared Aggregation) is a form of an association that indicates a whole-part relationship.

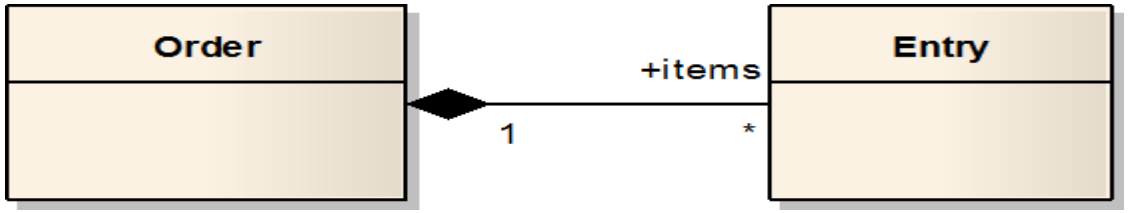
Containment indicates that an Invoice is logically comprised of an Order part (one, in this case). It does not make sense for Invoice object to exist independently of the Order object, because an Invoice always must contain an Order.

However, the Order can exist without the Invoice. That is, if the Invoice object is deleted, the Order can still exist (maybe it can be fulfilled free of charge).

Some believe that it is not necessary to indicate Containment/Shared Aggregation, and that more general directed Association is sufficient.

Another Advanced Association:

Composition (or Composite Aggregation) is a stronger form of aggregation that implies a whole- part relationship and lifetime of the parts



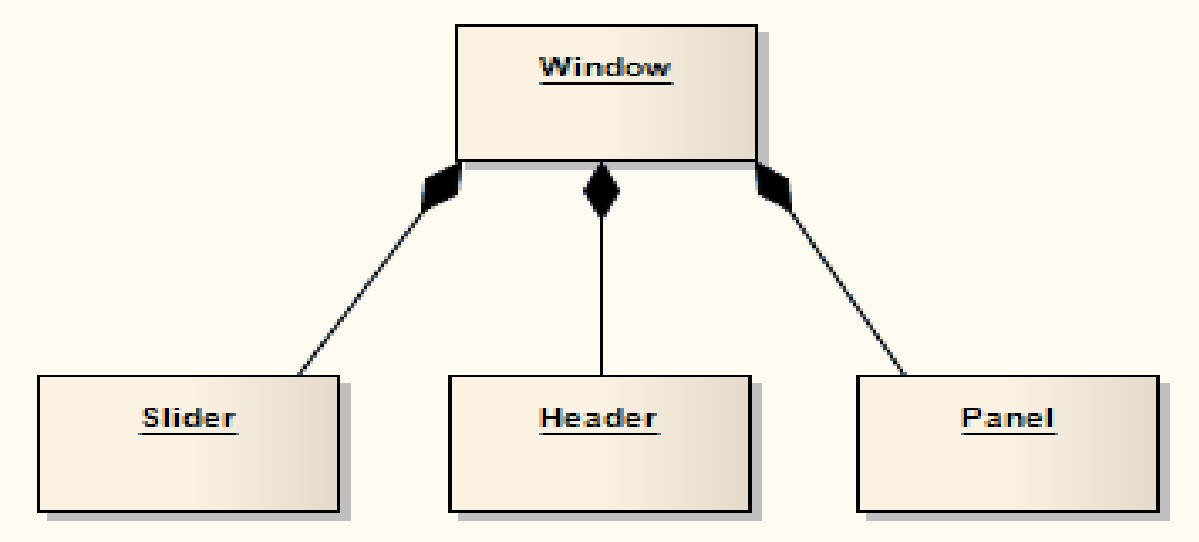
Composition implies that an Order is comprised of Entry parts (an Order logically does not exist without any Entry objects), and that the Entry objects cannot exist independently of the Order object; that is, the Order exclusively owns the Entry objects.

The lifetime of the Entry objects is controlled by the Order object. If the Order is deleted, all the Entries are deleted as well.

Again, some believe that it is not necessary to indicate Compostion/Composite Aggregation, and that more general directed Association is sufficient.

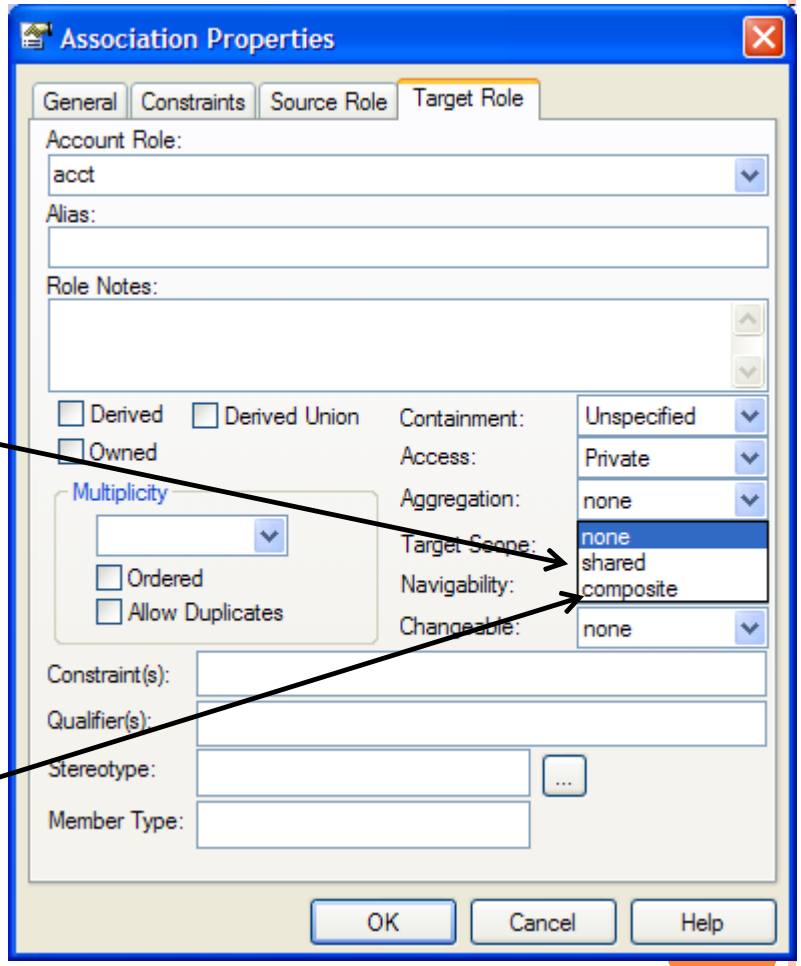
If a Composition is deleted, all of its parts are deleted with it

- A component (part instance) can be included in a maximum of one Composition at a time.
- A part can be individually removed from a Composition without having to delete the entire Composition.

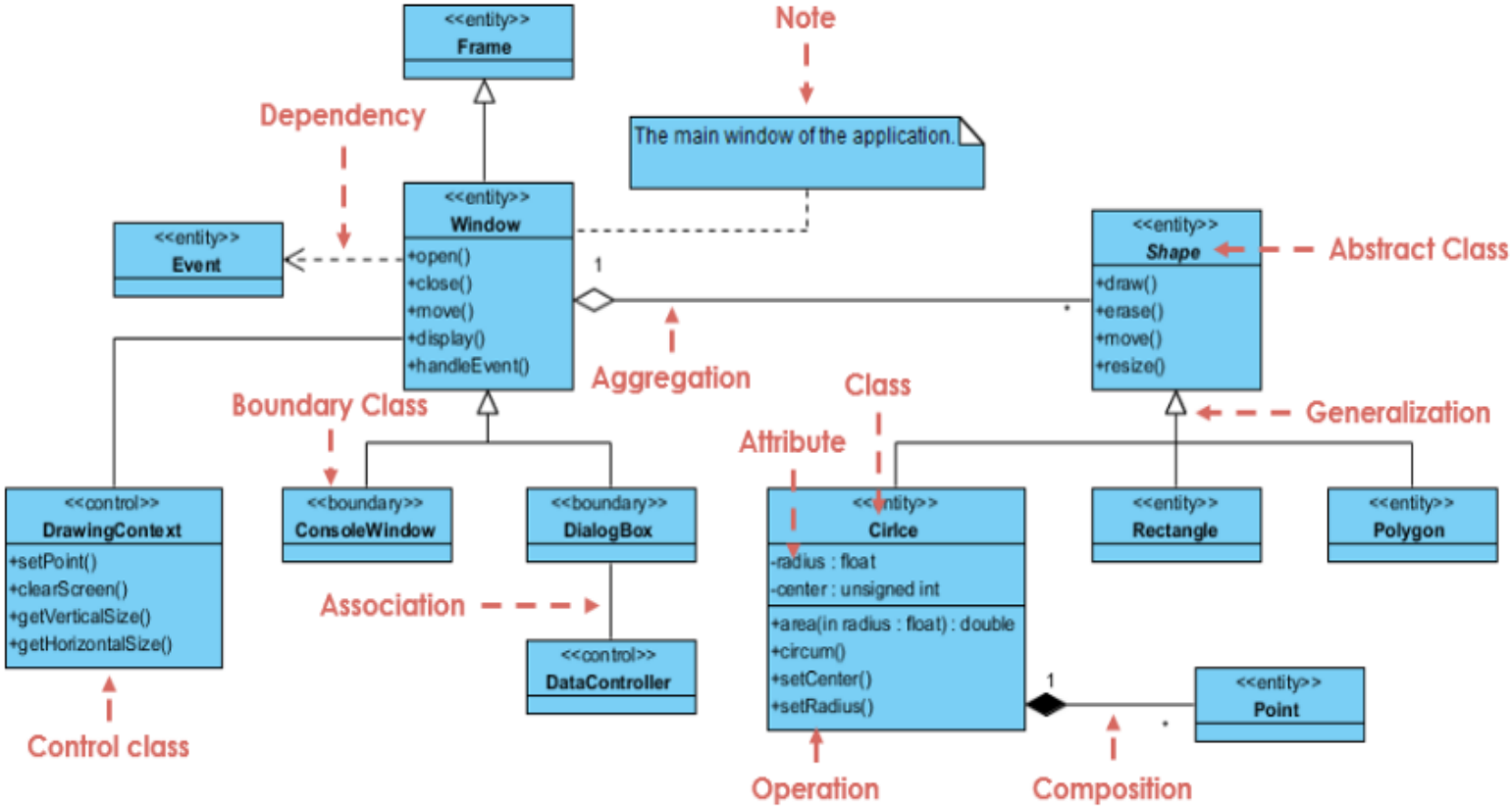


Inconsistent terminology between strict UML and EA

- **UML Containment**
  - A is composed of B; B can exist without A
  - EA refers to this as **Shared Aggregation**
- **UML Composition**
  - A is composed of B; B cannot exist without A
  - *EA refers to this as **Composite Aggregation***



FINALLY


<<entity>>
Frame
Note
The main window of the application.
Dependency
<<entity>>
Window
+open()
+close()
+move()
+display()
+handleEvent()
<<entity>>
Event
1
<<entity>>
Shape
+draw()
+erase()
+move()
+resize()
Abstract Class
Aggregation
Boundary Class
Class
Attribute
Generalization
<<control>>
DrawingContext
+setPoint()
+clearScreen()
+getVerticalSize()
+getHorizontalSize()
<<boundary>>
ConsoleWindow
<<boundary>>
DialogBox
<<entity>>
Cirlce
-radius : float
-center : unsigned int
+area(in radius : float) : double
+circum()
+setCenter()
+setRadius()
<<entity>>
Rectangle
<<entity>>
Polygon
Association
<<control>>
DataController
1
<<entity>>
Point
Control class
Operation
Composition

Thank You!