Class relationships in UML Class diagrams

In object-oriented programs, classes and/or class instances (that is, objects) are frequently related to other classes or objects in some fashion. In UML class diagrams, various connectors are used to illustrate the relationships among not only classes, but also class **instances** at run-time.

In this wiki article, we'll provide an explanation of the various relationships, along with UML diagrams and examples showing how the relationship is typically expressed in Java code.

Categorization of class relationships

There are four basic categories of class relationships:

1. **Inheritance**
2. **Dependency**
3. **Association**
4. **Aggregation**

The **Inheritance** relationship is further distinguished by *Generalization* and *Realization*, which are specific types of inheritance.

Similarly, **Aggregation** is further distinguished as either *Composition* (aka *Composite Aggregation*) or *Containment Aggregation* (aka *Shared Aggregation* or just plain *Aggregation*). Each category will be discussed in a separate section, below.

There is a fifth, less common class relationship called **Nesting** that is also discussed in a separate section below.

Inheritance relationships

Inheritance relationships illustrate only the **structural** relationships between classes and interfaces. These relationships are **static** - they are established when the code is written and do not change at run-time (when the code executes).

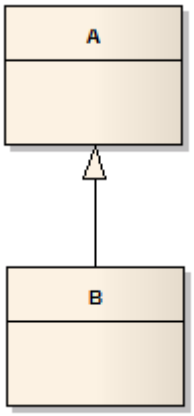
Generalization

The Generalization relationship denotes **implementation inheritance** between a parent class **A** (aka *base class* or *superclass*) and a child class **B** (aka *derived class* or *subclass*). This relationship is expressed in Java code is as follows:

```
public class B extends A
```

In this case **A** is either an **abstract** class or a “regular” class. In either case, **B** inherits both the attributes and the actual implementation of behaviors from **A** - thus the term **implementation inheritance**. Note that in Java, a class can only extend one other class.

In a UML Class diagram, this relationship is shown as



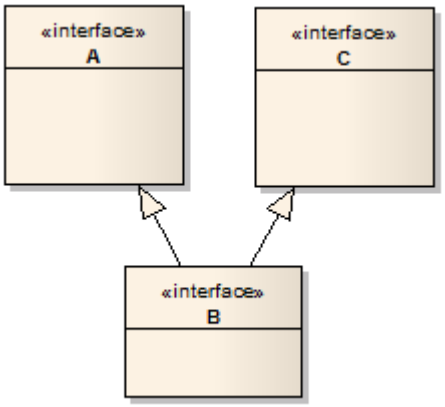
Another possible Java expression of the Generalization relationship is

```
public interface B extends A
```

In this case, both **A** and **B** must be interfaces (because of the use of the “extends” keyword). Note that in Java, an interface can extend one or more other interfaces, so it is also possible to write:

```
public interface B extends A, C
```

The UML Class diagram is



Here, the UML Class diagram makes use of the «interface» **stereotype** to further help illustrate that **A**, **B**, and **C** are all interfaces, rather than “regular” classes.

Realization

The Realization relationship denotes **behavioral inheritance** between a parent **interface A** and a child class **B**. In Java, this relationship is written as

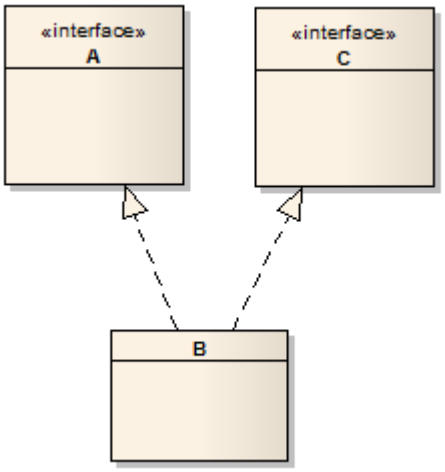
```
public class B implements A
```

In this case **A** must be an interface, while **B** can be either an **abstract** class or a “regular” class. Again, in Java, a class can implement one or more interfaces, so it is also possible to write:

```
public class B implements A, C
```

provided that both **A** and **C** are interfaces.

The corresponding UML Class diagram is

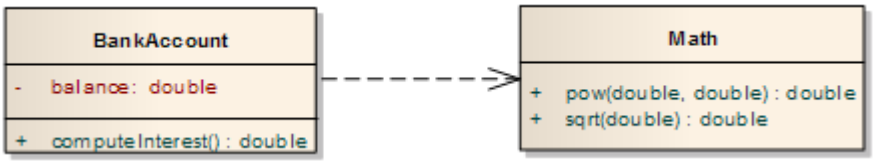


Since interfaces only declare behaviors - but cannot defined them - those behaviors must be implemented in **B** if **B** is a “regular” class. The behaviors do not have to be implemented in **B** if **B** is an abstract class.

The Dependency relationship

The Dependency relationship is a simple one used to denote that a class relies (temporarily) on another class in its implementation. As an example, consider a **BankAccount** class that computes the interest earned by the account based on the compound interest formula, which requires evaluation of an exponential expression. To compute the exponential value, the **BankAccount**'s **computeInterest**() method might use the **Math.pow()** method. In a Dependency context, a class like **BankAccount** depends on a class like **Math** for its operation only temporarily. Generally, if some class **A** doesn't need to define an attribute to maintain a reference to a class **B** instance for an extended period of time, or if class **A** only uses local variables to reference a class **B** instance, then that suggests a Dependency relationship.

The Dependency relationship is illustrated by a dashed line with an arrow pointing to the class that is depended upon:



The Association relationship

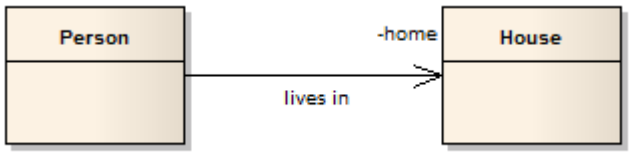
Association denotes a general relationship between classes which may change between consecutive executions of an application. Association implies only that two classes *have* a relationship - that is, they interact in some fashion (which is context-specific) at run-time. This relationship is usually implemented as an instance variable (although a local variable might also be used) in one class that references the other class. Suppose class **Person** has an Association with class **House** (that is, the Association is that a Person *lives in* a House); we'd express it in Java as

```
public class House {
// attributes (such as address) and code for class House go here...
}

public class Person {
   private House home; // Person includes a reference to an instance of House
```

```
// other attributes (such as name) and code for class Person go here, which may make use of the
'home' reference
}
```

The UML Class diagram illustrating this relationship is:



The **role name** (found next to the arrow) corresponds to the attribute name, **home** (in the Person class). Role names are optional, but they are useful to more completely indicate the exact mechanism with which the association is made.

Since the **home** attribute in the Person class can hold a reference to a House object, it provides a way to navigate from a Person to his/her House. Here the arrowhead on the Association connector indicates a one-way navigation from Person to House. This means that, given a Person, you can find (via the **home** reference) the House in which the Person lives. However, given the House, you cannot find the Person that lives there (since the House contains no attribute referencing the Person). In fact, this diagram does not denote a way to navigate from a House to all the Persons who consider it their home (since of course there is no such mechanism implemented in the actual code!).

The Association connector can include named roles at both ends when bi-directional navigation is possible.

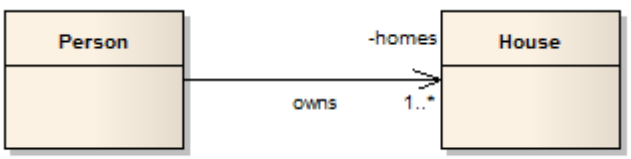
The **lives in** association name describes (in simple terms) the relationship. Here, we specify that a Person "lives in" a House. This is the relationship we mean to imply in our design and implementation of these classes. Suppose our design instead meant to illustrate that a Person owns a House - then we might have used a phrase like "owns a" instead of "lives in".

Now suppose class **Person** has an Association with multiple instances of class **House** - that is, a Person owns one or more Houses. One way to express this in Java is

```
public class House {
// attributes (such as address) and code for class House go here...
}

public class Person {
   private List<House> homes; // Person includes a reference to an collection of Houses
}
```

In this case, the **multiplicity** specifier 1..* indicates that a Person owns "one or more" Houses. Multiplicity >1 indicates that some type of Collection is used in the implementation (in this example, it is a List<House>). The **role name** in this case is a reference to the Collection (e.g. a List<House> named **homes**) rather than to a House.

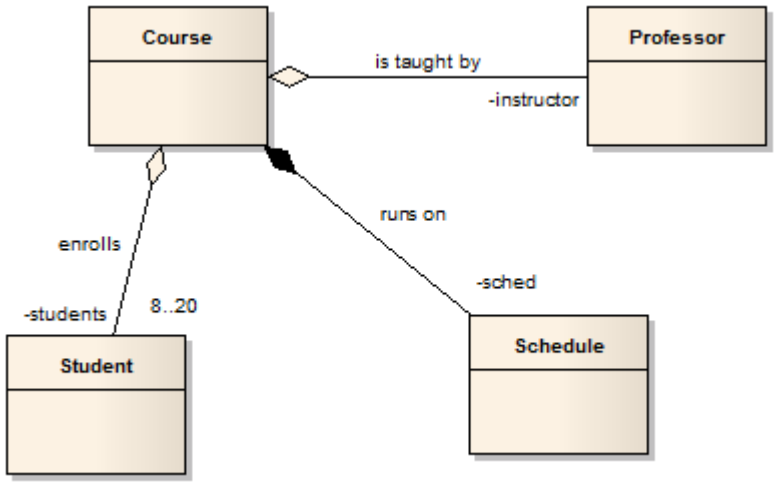


Aggregation relationships

Aggregation relationships are used to indicate that instances of one class *contain* or *are composed of* instances of other classes. In either case, aggregation denotes a "whole-part" relationship where an instance of class **A** is comprised of one or more subparts **B** (and possibly **C**, **D**, etc); we can also say (in both cases) that **A** "has-a" **B** (and a **C**, and a **D**, etc). That is, **A** is really an aggregate of multiple objects.

As an example, consider a **Course** class. The definition of a **Course** includes the following subparts: a collection of **Student**s, one **Professor**, and a **Schedule**. It makes no sense for a **Course** instance to exist without all of the subparts, since they are all vital to the concept of a **Course**. Thus, for a **Course** to be valid, a **Course** instance must either contain or be composed of all of its subparts. No subpart can be missing, or else the **Course** is invalid.

The UML Class diagram illustrating a possible relationship between the various classes is



Shared Aggregation (**Containment**) is illustrated by the connectors using the white (or hollow) diamond. Composite Aggregation (**Composition**) is illustrated by the connector with the black (or filled) diamond.

In Java, these relationships are expressed as follows:

```
public class Course {
   private List<Student> students; // reference to a collection of Students,
   private Professor instructor;  // a reference to an single instance of a Professor,
   private Schedule sched; // and of a single Schedule

// other attributes and code for the Course class go here...
}
```

With **Containment**, we say that an instance of class **Course** *contains* from 8 to 20 **Student**s and one **Professor**. As **Course** is composed of multiple instances of **Student**, the aggregation is once again accomplished with the help of a collection class, as shown above in the Java code.

With **Composition**, we say that an instance of class **Course** *is composed of* an instance of **Schedule**.

In Java, the relationship for **Composition** is defined exactly the same way as it is for **Containment**, as you can see above. The immediate question is then: "how do **Composition** and **Containment** differ?" The important distinction is this: in **Containment**, the subparts are **NOT** exclusively owned by the containing class; that is, the subparts can exist independently of **Course** and their

lifetimes are not managed by **Course** - both **Student**s and **Professor**s can exist both before and after **Course** is instantiated. On the other hand, a **Schedule** *only makes sense to exist as long as the* **Course** *exists; if the* **Course** *is deleted, then the* **Schedule** *should cease to exist too*. For this reason, **Containment** is considered a "weaker" form of aggregation compared to **Composition**.

In **Composition**, the intention is that an instance of **A** *owns* the instance of **B** *exclusively*. This means two important things:

- **B** cannot exist independently of **A**; **B** only exists as a subpart of the entire aggregate with is **A**.
- If the instance of **A** is deleted, then **B** by definition must also be deleted. No other object can own **B**, since **A** manages **B** and controls **B**'s lifetime.

A secondary question is: "How are the lifetimes of contained and composed objects managed?" There is no simple answer. It is up to the aggregating class to implement whatever code is needed to manage an owned object's lifetime. For example, the creation of a **Schedule** may take place in the **Course** constructor, while the creation of the **Student** and **Professor** objects may take place before the **Course** is constructed (since the **Course** does not manage their lifetimes). In Java, an object is destroyed implicitly (by the Garbage Collector of the Java Virtual Machine) when all references to that object cease to exist. If **Course** contains the ONLY reference to a **Schedule** instance, then when **Course** is destroyed, **Schedule** will also be destroyed. It is of course up to the Java code author to ensure that no other references to the **Schedule** object are passed along to some other part of the code.

Note that in languages like C++, **destructor** methods must be explicitly provided by the code author, so lifetime management becomes more burdensome to the application developer. But that is another topic.

The Nesting relationship

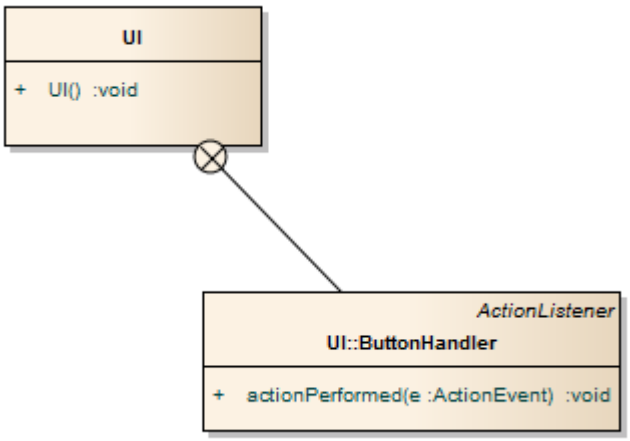
The Nesting relationship is used to indicate that a class contains a nested, or inner class. In Java, this relationship is expressed as follows:

```
public class UI {
// This UI class implements the graphical user interface code for an application.
// Attributes for the UI class go here...
   public UI() { // constructor for UI
      // user interface initialization goes here...
      JButton okButton = new JButton("OK");
      okButton.addActionListner( new ButtonHandler() ); // subscribe to button events

   private class ButtonHandler implements ActionListener {
   // This inner class handles the button events coming from the UI class's JButton components
       public void actionPerformed(ActionEvent e) {
          // code for responding to button presses goes here...
        }
   } // end ButtonHandler inner class

} // end UI (outer) class
```

The UML Class diagram illustrating that class **UI** nests a (private) inner class **ButtonHandler** is shown below:


UI
+ UI() :void
ActionListener
UI::ButtonHandler
+ actionPerformed(e :ActionEvent) :void