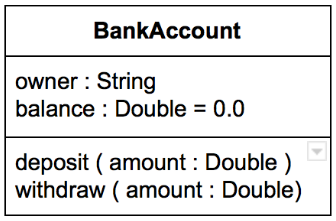
UML Class Diagram

Class diagrams are a neat way of visualizing the classes in your system before you actually start coding them up. They're a static representation of your system structure.

Class Representation in UML

A class is represented as a box with 3 compartments. The uppermost one contains the class name. The middle one contains the class attributes and the last one contains the class methods. Like this:



Visibility of Class Members

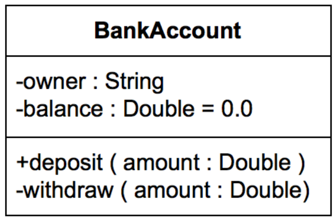
In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: public, protected, private, and package.

The +, -, # and ~ symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations
- ~ denotes package attributes or operations

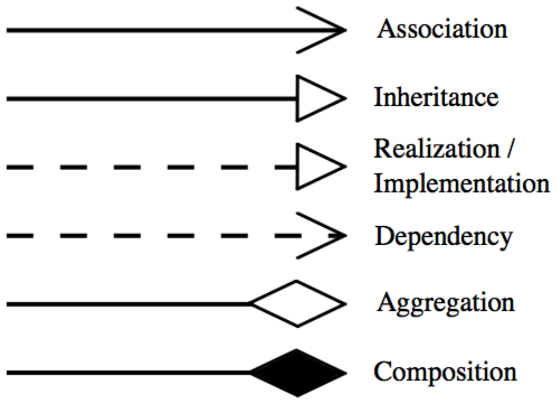
Class members (attributes and methods) have a specific visibility assigned to them. See table below for how to represent them in UML.

Let's specify the visibility of the members of the Bank Account class above.



We made the `owner` and balance private as well as the withdraw method. But we kept the deposit method public. (Anyone can put money in, but not everyone can take money out. Just as we like it.)

Relationships



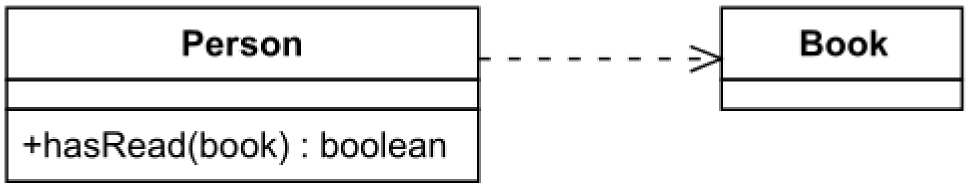
Association

a relationship between two separate classes. It joins two entirely separate entities. There are four different types of association: bi-directional, uni-directional, aggregation (includes composition aggregation) and reflexive. Bi-directional and uni-directional associations are the most common ones.

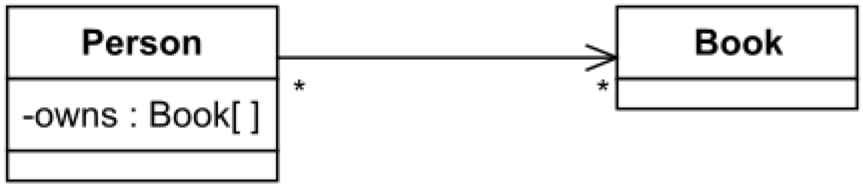
This can be specified using multiplicity (one to one, one to many, many to many, etc.).

UML class diagrams include the following types of use-relationships, in order from weakest to strongest.

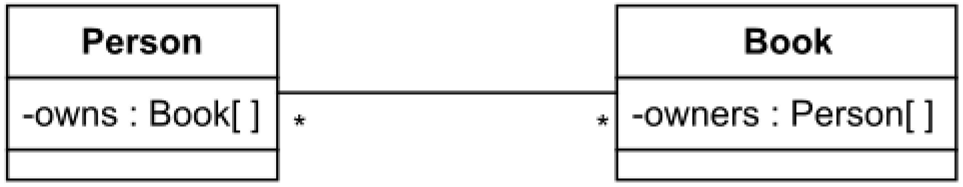
- Dependency: An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship. For example, the Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).



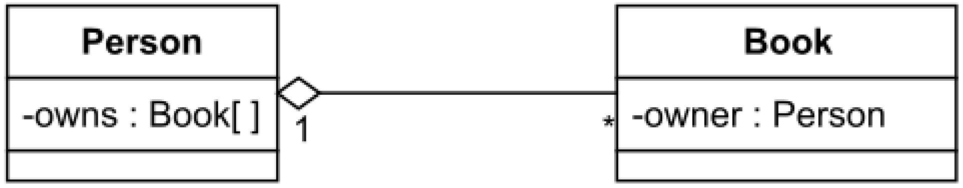
- Unidirectional Association: An object might store another object in a field. For example, people own books, which might be modeled by an `owns` field in Person objects. However, a book might be owned by a very large number of people, so the reverse direction might not be modeled. The *'s in the figure indicate that a book might be owned any number of people, and that a person can own any number of books.



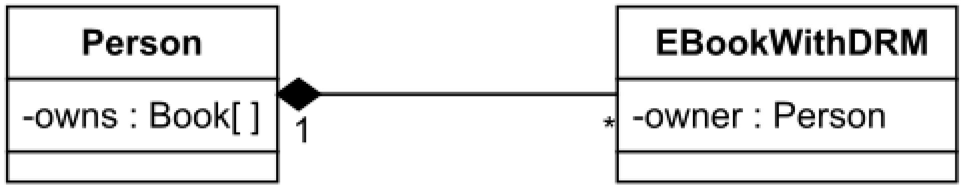
- Bidirectional Association: Two objects might store each other in fields. For example, in addition to a Person object listing all the books that the person owns, a Book object might list all the people that own it.



- Aggregation: One object A has or owns another object B, and/or B is part of A. For example, suppose there are different Book objects for different physical copies. Then the Person object has/owns the Book object, and, while the book is not really part of the person, the book is part of the person's property. In this case, each book will (usually) have one owner. Of course, a person might own any number of books.



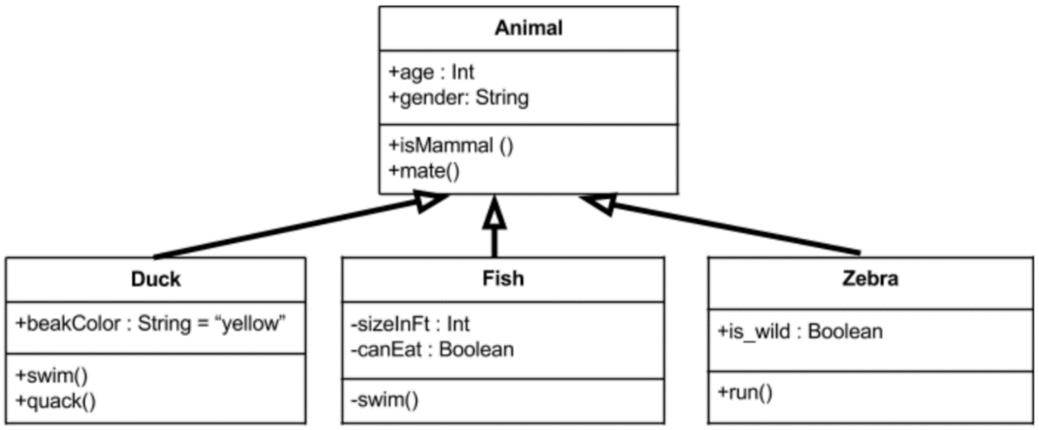
- Composition: In addition to an aggregration relationship, the lifetimes of the objects might be identical, or nearly so. For example, in an idealized world of electronic books with DRM (Digital Rights Management), a person can own an ebook, but cannot sell it. After the person dies, no one else can access the ebook. [This is idealized, but might be considered less than ideal.]



a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other.

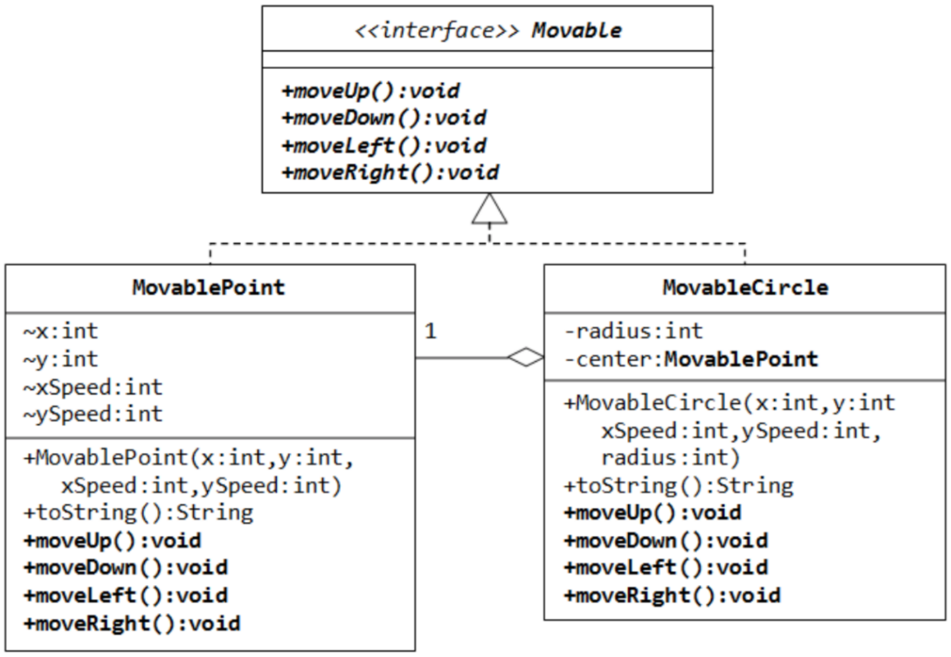
Inheritance

indicates that child (subclass) is considered to be a specialized form of the parent (super class). For example consider the following:



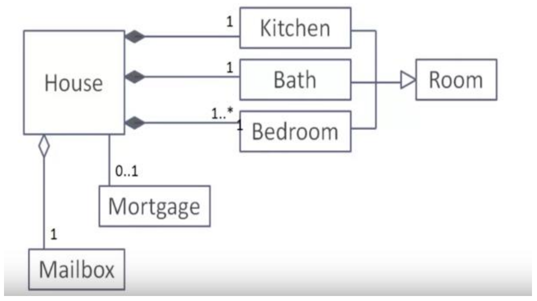
Implementation

A relationship between two model elements, in which one model element implements/executes the behavior that the other model element specifies.

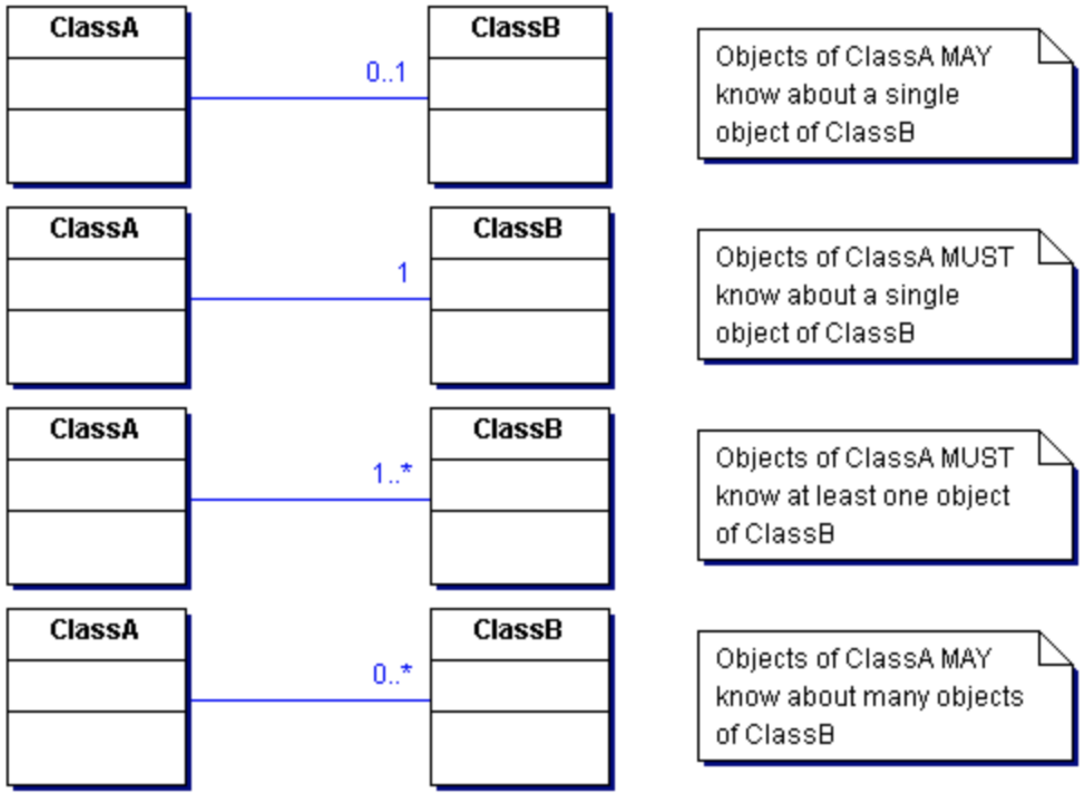


Multiplicity

After specifying the type of association relationship by connecting the classes, you can also declare the cardinality between the associated entities. For example:



The above UML diagram shows that a house has exactly one kitchen, exactly one bath, at least one bedroom (can have many), exactly one mailbox, and at most one mortgage (zero or one).


ClassA
ClassB
0..1
Objects of ClassA MAY know about a single object of ClassB
ClassA
ClassB
1
Objects of ClassA MUST know about a single object of ClassB
ClassA
ClassB
1..*
Objects of ClassA MUST know at least one object of ClassB
ClassA
ClassB
0..*
Objects of ClassA MAY know about many objects of ClassB