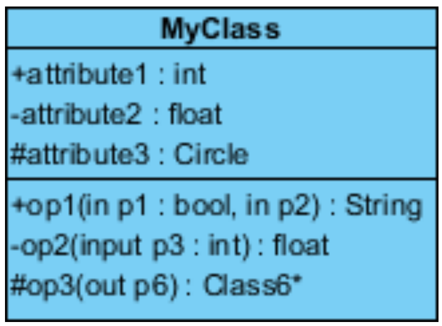
UML Class Diagram Tutorial

Class Notation

A class notation consists of three parts:

1. **Class Name**
   - The name of the class appears in the first partition.
2. **Class Attributes**
   - Attributes are shown in the second partition.
   - The attribute type is shown after the colon.
   - Attributes map onto member variables (data members) in code.
3. **Class Operations** (Methods)
   - Operations are shown in the third partition. They are services the class provides.
   - The return type of a method is shown after the colon at the end of the method signature.
   - The return type of method parameters are shown after the colon following the parameter name.
   - Operations map onto class methods in code



The graphical representation of the class - MyClass as shown above:

- MyClass has 3 attributes and 3 operations
- Parameter p3 of op2 is of type int
- op2 returns a float
- op3 returns a pointer (denoted by a *) to Class6

Class Relationships

A class may be involved in one or more relationships with other classes. A relationship can be one of the following types: (Refer to the figure on the right for the graphical representation of relationships).

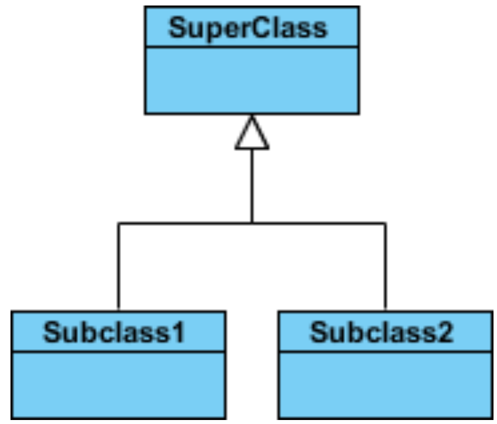
<table>
<tr>
<td>

Inheritance (or Generalization):

- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of Super Class.

A solid line with a hollow arrowhead that point from the child to the parent class

</td>
<td>



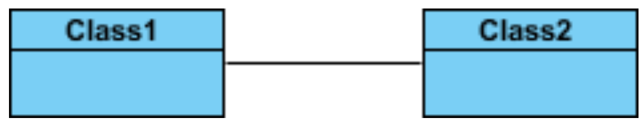
</td>
</tr>
<tr>
<td>

Simple Association:

- A structural link between two peer classes.
- There is an association between Class1 and Class2

A solid line connecting two classes

</td>
<td>



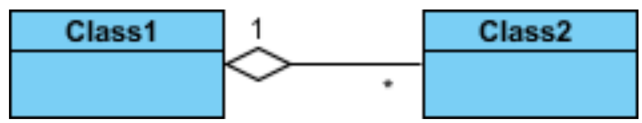
</td>
</tr>
<tr>
<td>

Aggregation:
A special type of association. It represents a "part of" relationship.

- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.

A solid line with a unfilled diamond at the association end connected to the class of composite

</td>
<td>



</td>
</tr>
<tr>
<td>

Composition:
A special type of aggregation where parts are destroyed when the whole is destroyed.

- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

A solid line with a filled diamond at the association connected to the class of composite

</td>
<td>

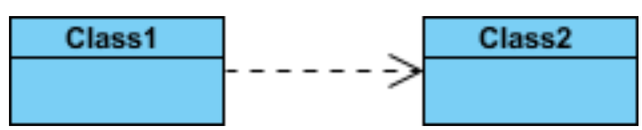
</td>
</tr>
<tr>
<td>

Dependency:

- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2

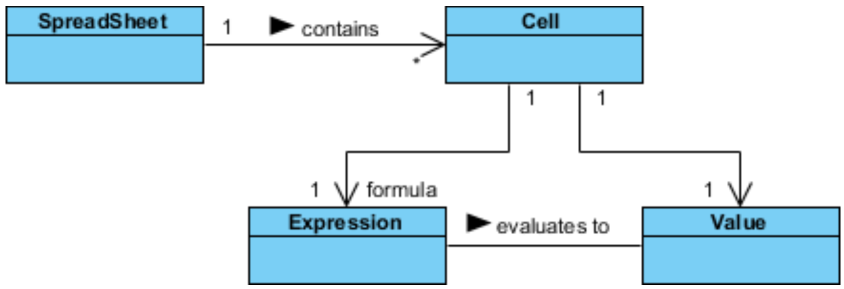
A dashed line with an open arrow

</td>
<td>



</td>
</tr>
</table>

Navigability



The arrows indicate whether, given one instance participating in a relationship, it is possible to determine the instances of the other class that are related to it.
The diagram above suggests that,

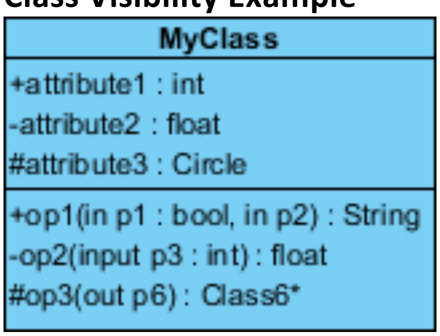
- Given a spreadsheet, we can locate all of the cells that it contains, but that
  - we cannot determine from a cell in what spreadsheet it is contained.
- Given a cell, we can obtain the related expression and value, but
  - given a value (or expression) we cannot find the cell of which those are attributes.

Visibility of Class attributes and Operations

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.

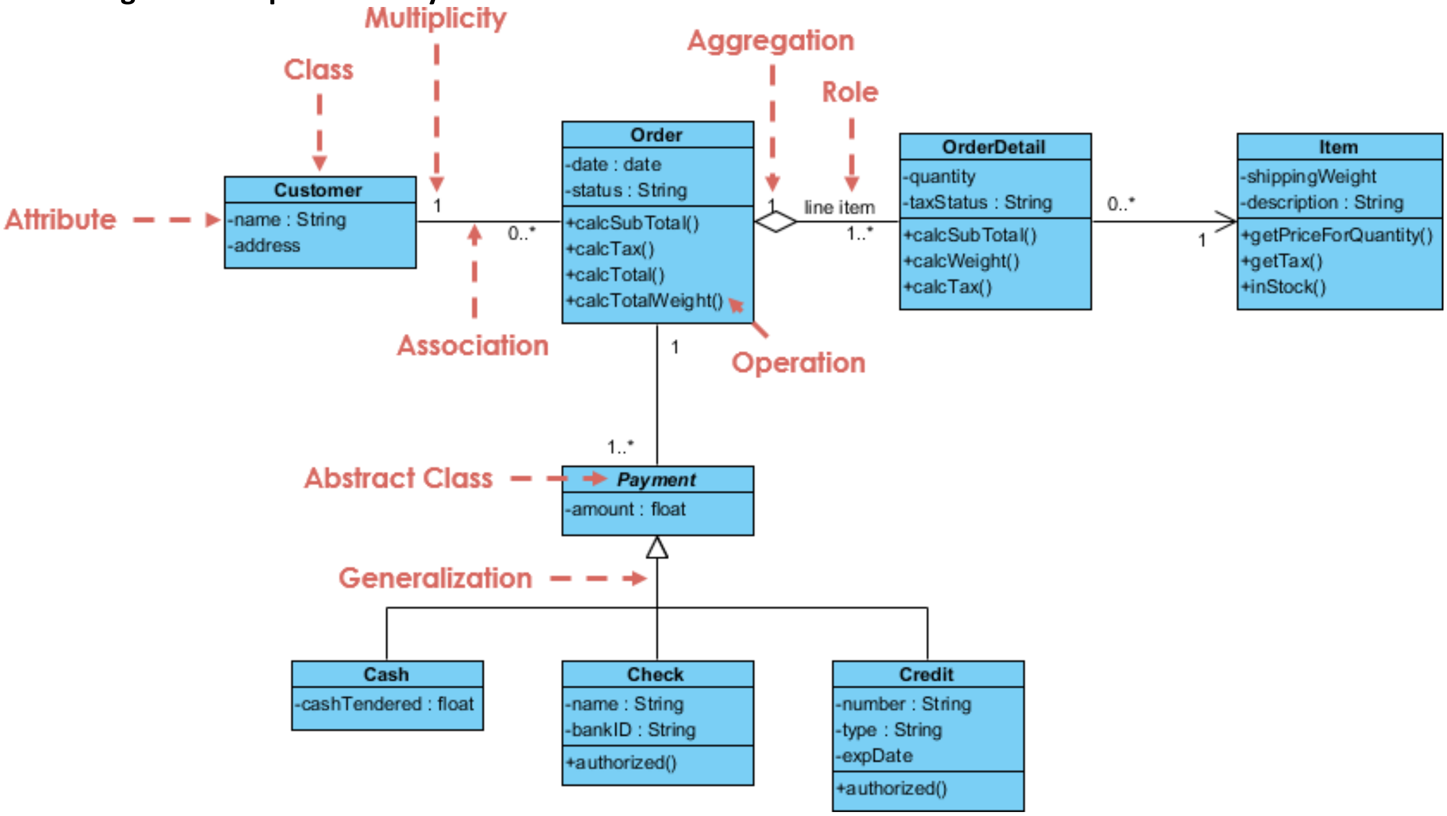
- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations
- ~ denotes package attributes or operations

Class Visibility Example



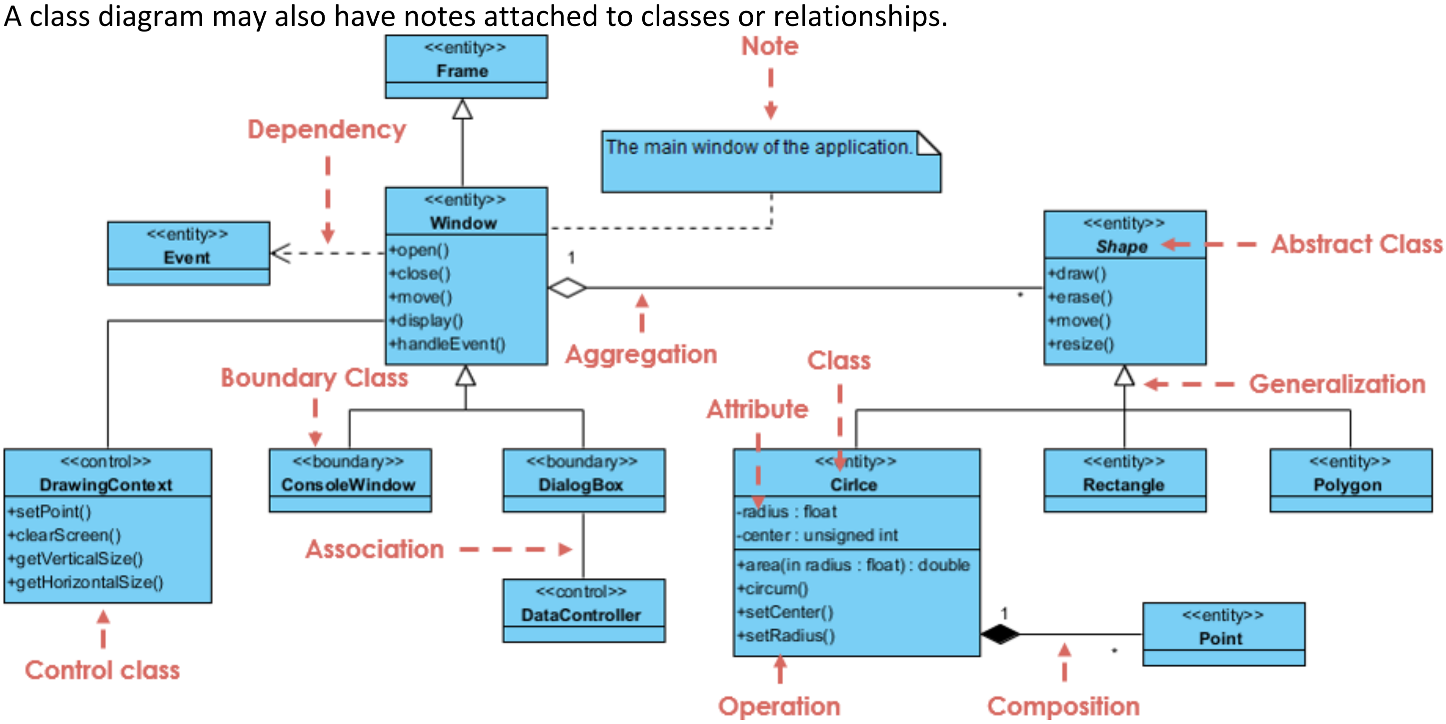
	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Class Diagram Example: Order System



Class Diagram Example: GUI

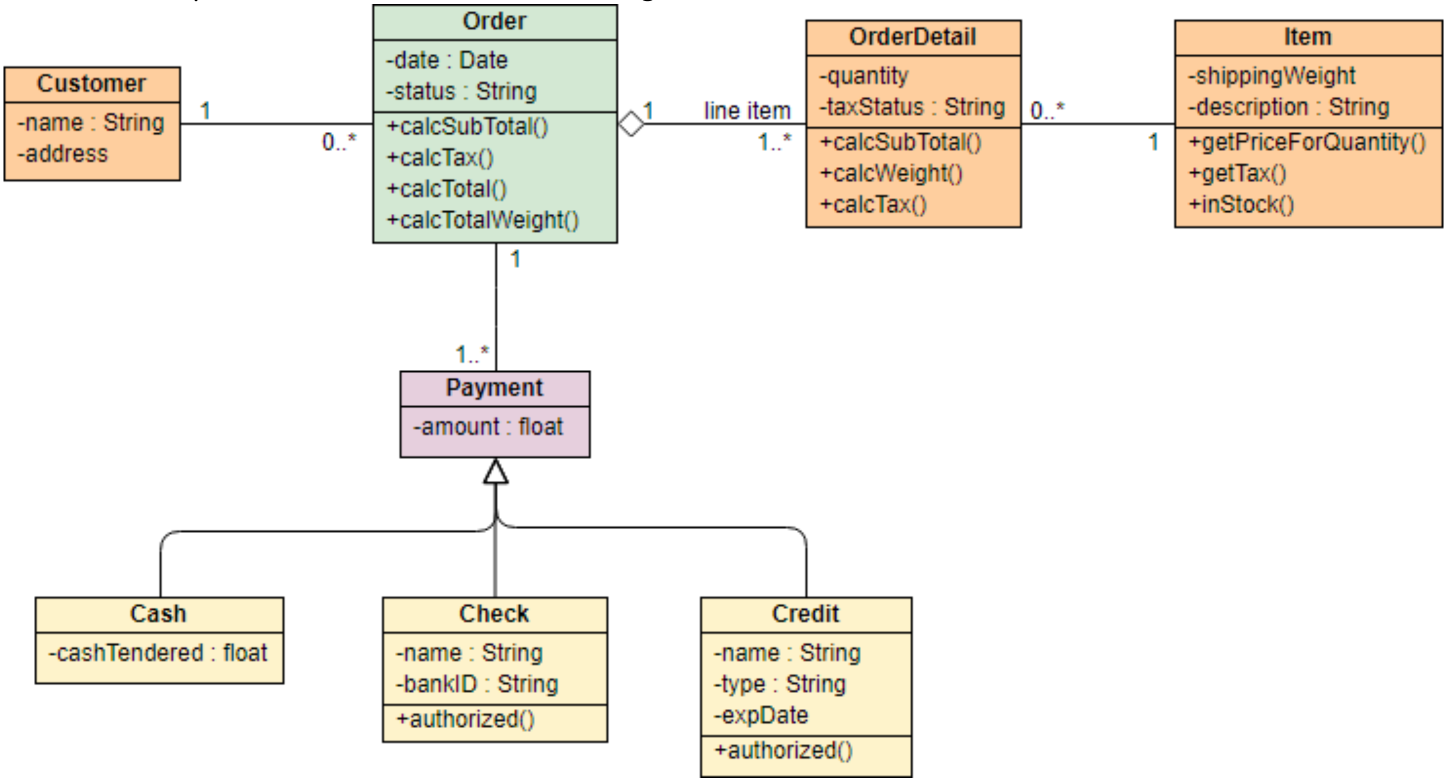
A class diagram may also have notes attached to classes or relationships.



UML Class Diagram Tutorial

What is a Class Diagram in UML?

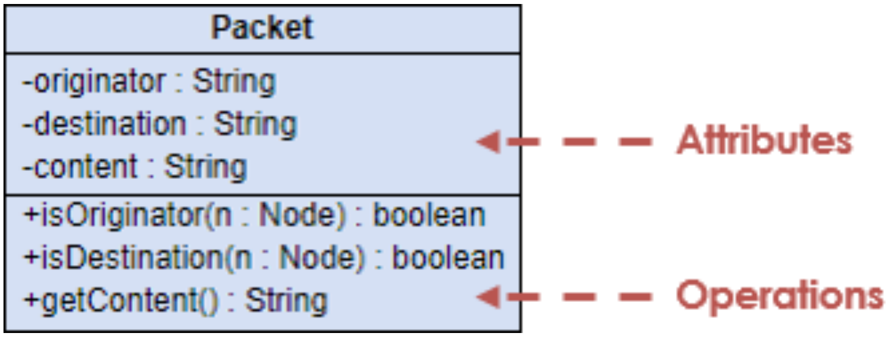
A class diagram describes the structure of an object-oriented system by showing the classes in that system and the relationships between the classes. A class diagram also shows constraints, and attributes of classes.



Class Diagram Notations

Class

The UML representation of a class is a rectangle containing three compartments stacked vertically, as shown in the Figure:



Attribute

The attribute section of a class lists each of the class's attributes on a separate line. The attribute section is optional, but when used it contains each attribute of the class displayed in a list format. The line uses this format: *name : attribute type* (e.g. cardNumber : Integer).

Operation

The operations are documented in the bottom compartment of the class diagram's rectangle, which also is optional. Like the attributes, the operations of a class are displayed in a list format, with each operation on its own line. Operations are documented using this notation: *name (parameter list) : type of value returned* (e.g. calculateTax (Country, State) : Currency).

Relationships

<u>Association</u>

Some objects are made up of other objects. Association specifies a "has-a" or "whole/part" relationship between two classes. In an association relationship, an object of the whole class has objects of part class as instance data.

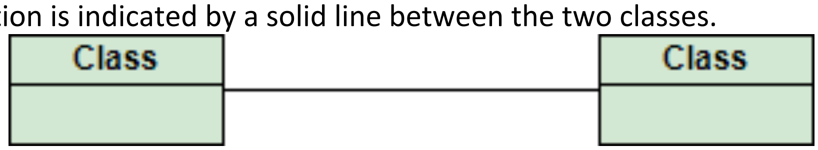
In a class diagram, an association relationship is rendered as a directed solid line.

Unidirectional association - In a unidirectional association, two classes are related, but only one class knows that the relationship exists.

A unidirectional association is drawn as a solid line with an open arrowhead pointing to the known class.

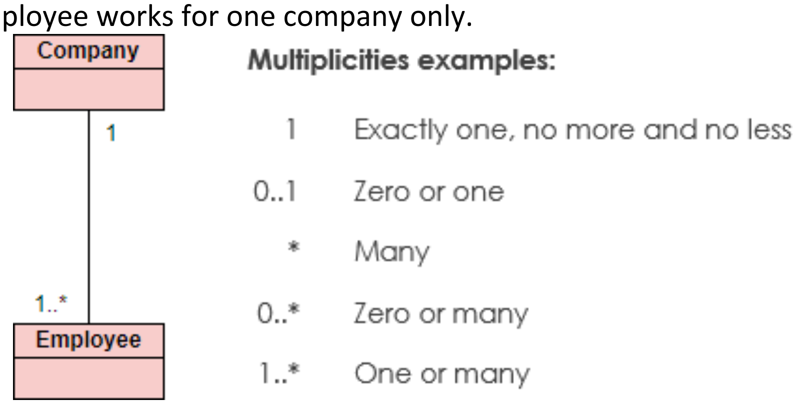
Bidirectional (standard) association - An association is a linkage between two classes. Associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship, unless you qualify the association as some other type.

A bi-directional association is indicated by a solid line between the two classes.



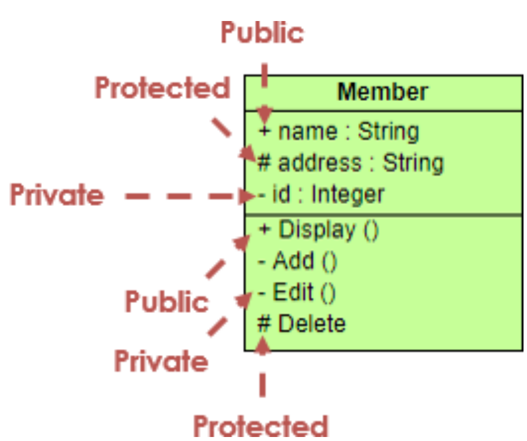
<u>Multiplicity</u>

Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.



<u>Visibility</u>

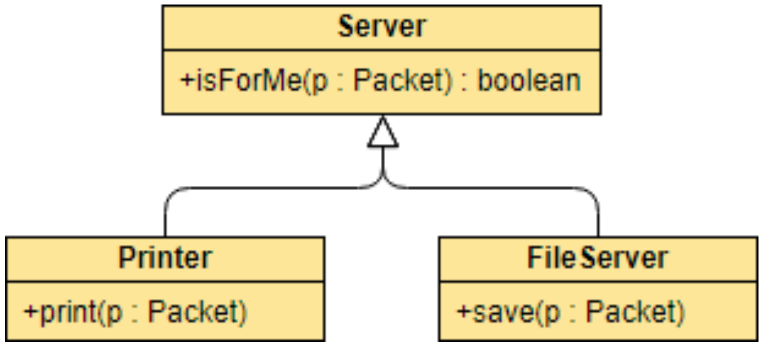
Visibility is used to signify who can access the information contained within a class denoted with +, -, # and ~ as show in the figure:



Generalization

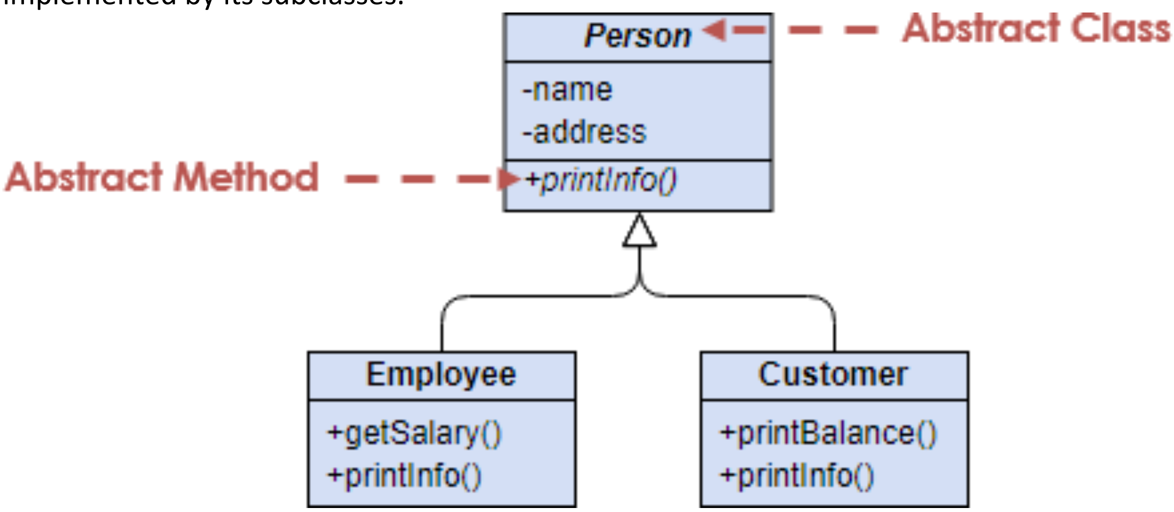
A generalization is a relationship between a general thing (called the superclass) and a more specific kind of that thing (called the subclass). Generalization is sometimes called an "is a kind of" relationship and is established through the process of inheritance.

In a class diagram, generalization relationship is rendered as a solid directed line with a large open arrowhead pointing to the parent class.



Abstract Classes and methods

In an inheritance hierarchy, subclasses implement specific details, whereas the parent class defines the framework its subclasses. The parent class also serves a template for common methods that will be implemented by its subclasses.

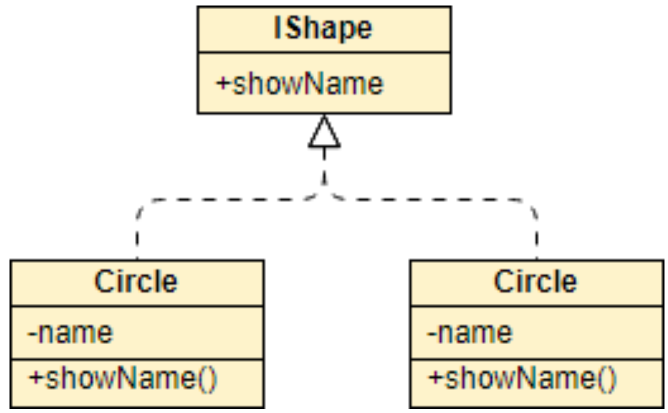


The name of an **abstract Class** is typically shown in italics; alternatively, an abstract Class may be shown using the textual annotation, also called stereotype {abstract} after or below its name.

An **abstract method** is a method that do not have implementation. In order to create an abstract method, create a operation and make it italic.

Realization

A realization is a relationship between two things where one thing (an interface) specifies a contract that another thing (a class) guarantees to carry out by implementing the operations specified in that contract. In a class diagram, realization relationship is rendered as a dashed directed line with an open arrowhead pointing to the interface.

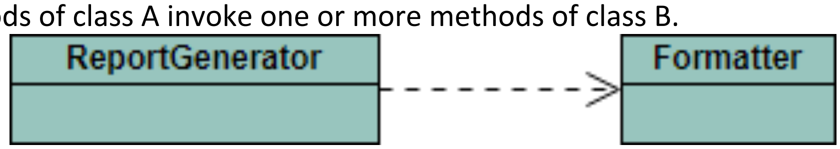


Dependency

Dependency indicates a "uses" relationship between two classes. In a class diagram, a dependency relationship is rendered as a dashed directed line.

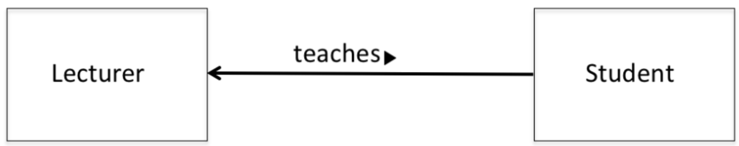
If a class A "uses" class B, then one or more of the following statements generally hold true:

1. Class B is used as the type of a local variable in one or more methods of class A.
2. Class B is used as the type of parameter for one or more methods of class A.
3. Class B is used as the return type for one or more methods of class A.
4. One or more methods of class A invoke one or more methods of class B.

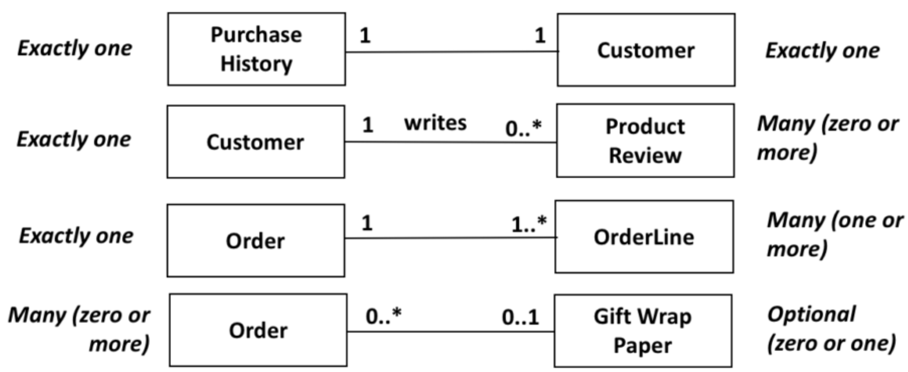


Associations
Associations represent the *relationships* between classes. They are represented as a line connecting the two related classes, and at their simplest, that's all they are.
If the association has a navigability (i.e. an arrow on one end) it indicates that one class knows about the other but not vice versa. It's very handy to introduce navigabilities into your diagram as early as possible. If you're doing analysis, it will help you understand the responsibilities of each class and how they interact, and if you're doing design it'll help you think about the dependencies between your classes and the strength of their coupling.

You can give an association a name if it helps you understand the nature of the relationship. This will require some thought, though – something like ***has*** is not a good name! It'll help you more if you use something more descriptive.



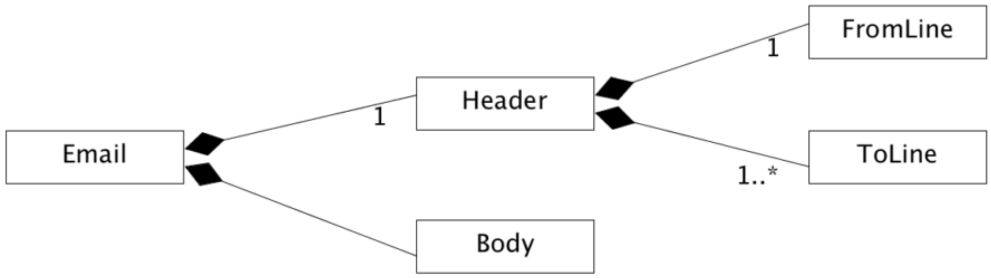
If present, a small triangle next to the name shows you which direction to read the label. In this diagram, Students know about their Lecturer (i.e. each Student is keeping track of its associated Lecturer) but Lecturers don't know their Students. The little arrow next to the word ***teaches*** shows you that Lecturer teaches Student, not the other way around. Getting these notations, the wrong way around is a common mistake.
Each end of an association can has also show its *multiplicity*. This indicates how many objects of each class are involved in the association. For example:



You should steer clear of using variables in multiplicities, e.g. 0..N rather than 0..* – it's not legal UML, and it can be confusing unless you make it very clear what those variables mean.

Aggregate associations
UML 2.4.1 defines two kinds of aggregate association: *composite aggregation* and *shared aggregation*. Composite aggregation is shown as a filled (black) diamond and shared aggregation is shown as an unfilled (white) diamond.
The UML 2.4.1 standard says that composite entities can't share members but aggregate entities can. That is, if an object is part of one composition, it can't be part of another. For the time being, we'll go with this definition – but as we'll see later, there's some controversy around it.

Composite aggregation often appears in software systems, because many composite objects appear in real life: a dog is a composite of a head, a body, a tail and four legs; an email is composed of a header and a text message; the header is composed of a From: line, a To: line, etc.
Composite aggregation is the kind of aggregation that exists *between a whole and its parts.*



The three most important characteristics of composite aggregation as opposed to shared aggregation are:

- the composite object does not exist without its components,
- at any time, each component may be part of only one composite, and
- component objects are *likely* to be of mixed types (although this isn't *always* the case.)

Shared aggregation is also a familiar concept from real life: a forest is an aggregate of trees, and a flock is an aggregate of sheep. It's so common that the word "aggregation", by itself, usually refers to shared aggregation.
An aggregation is the kind of association that exists *between a group and its members.*

The three most important characteristics of shared aggregation are:

- the aggregate object may potentially exist without its constituent objects (although not necessarily in a useful state),
- at any time, each object may be a constituent of more than one aggregate (e.g. a person may belong to several clubs), and
- constituent objects are typically of the same class (but, again, that's not always the case).

Constraints
The elements of a class diagram (associations, attributes and generalisation, etc.) are effectively placing constraints on the system. For example, an association can indicate that an order must have a customer by having a multiplicity of 1 at the customer end. But sometimes, you'll need to specify constraints on the system that can't be covered by these structural elements.
In UML, you can simply write these constraints in natural language and place them inside curly brackets ({ }). At the conceptual level, it's best if you use simple statements, such as {ordered} to show that a particular collection must be ordered in some way.
You can apply these constraints to any part of the diagram: attributes, associations, classes, etc.