

Unified Modeling Language Tutorial

This UML tutorial is designed to provide an overview of the Unified Modeling Language (UML) notation used in the AITP NCC Java Competition Event. Both the 2000 and 2001 event problem statements were documented using [TogetherSoft](#) TogetherJ and the UML version 1.3 notation. The Object Management Group ([OMG](#)) is currently working to finalize UML version 2.0. This tutorial is going to focus on UML notation basics, which will conform to all versions of UML (from 1.3 through 2.0). This tutorial is divided into five parts:

[UML Class Diagram Notation](#)
[Forms of Class Associations](#)
[Class Dependency](#)
[Class Cardinality and Multiplicity Notation](#)
[Class Visibility and Scope](#)

This tutorial is intended as an introduction to basic UML. You are encouraged to explore the Problem Statement UML Documentation to gain a better understanding of how UML is used in industry as a design and modeling tool.

This UML tutorial was prepared by:

Don Baldwin
IT Enterprise Core Architect
Auldenfire Sweden AB

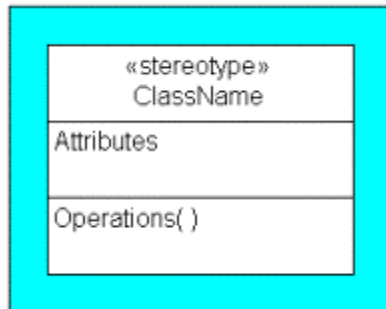
Enterprise Systems Architecture Laboratory
Auldenfire Sweden AB
Forsbackagatan 24
SE - 123 - 43 Farsta
Stockholm, Sweden

+46 (0)8 556 814 00 Phone
+46 (0)8 556 814 05 Fax

UML Class Diagram Notation

A **Class Diagram** provides an overview of a system by showing its classes and the relationships among them. Class diagrams are static - they display what interacts, but not what happens when they do interact.

The Class is represented as a rectangle that is typically divided into three regions. The top region contains the class name, the middle region contains the class attributes, and the bottom region contains the class methods (or operations).

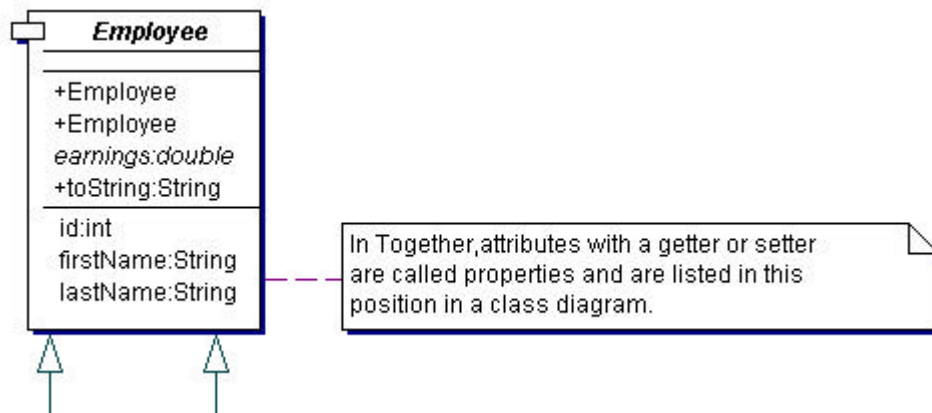


Class Name

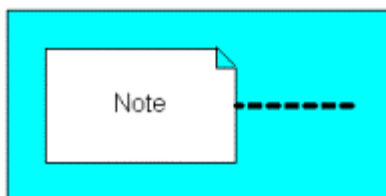
Class Attributes

Class Methods (Operations)

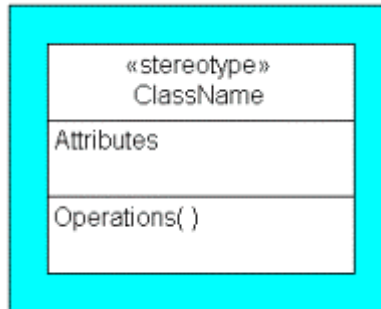
NOTE: TogetherJ adds a fourth region to the bottom that shows the class properties, which are attributes with assessor (getter) and mutator (setter) methods:



A class can have a note associated with it. Notes do not effect the underlying code.



A note contains information to people reading the UML diagram (or model). Notes provide additional context to help explain details that are not apparent in the diagram.

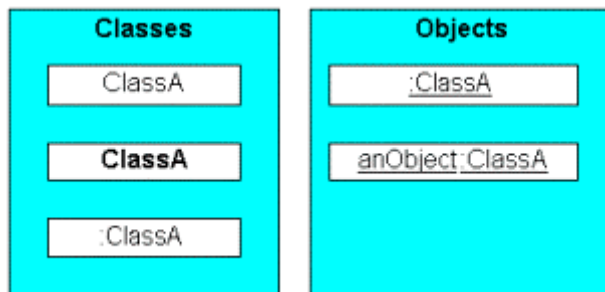


The illustration below shows how TogetherJ represents a note and the corresponding Java source code for the class illustrated above.



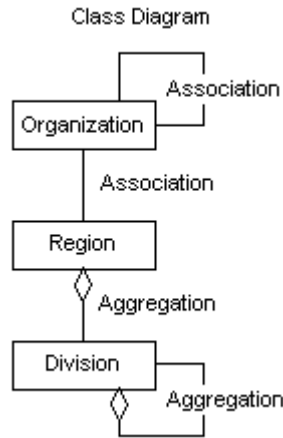
```
public class ClassA
{
}
```

A class is a template from which objects are instantiated (derived from). The act of instantiating an object from a class is similar to building a house from a blueprint. The class is a form of blueprint for building objects. Classes can be represented in a short form by using a box with no internal regions as illustrated below.



AITP National Collegiate Conference Java Competition Event UML Tutorial

Object Diagrams show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships. Objects are indicated by placing the name of the object (instance name) to the left of the class name and separating the two names by a colon. Object names are typically underlined in a UML diagram to visually help in distinguishing them from classes (e.g. [anObject:ClassA](#)). Examples of simplified class and object UML notation are illustrated above.

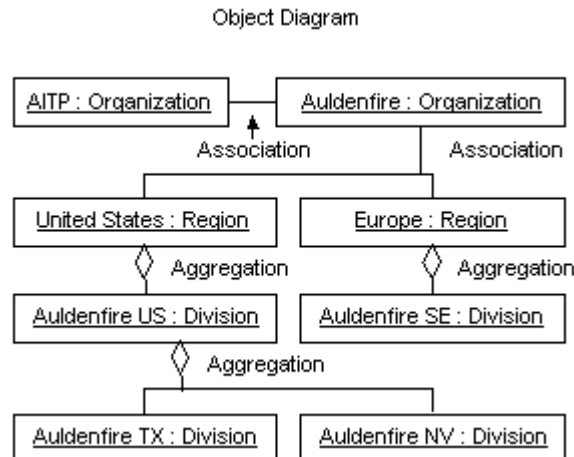


The **class diagram** to the left illustrates an example of an **Organization** that can be **associated** with other **Organizations** (**recursive association**) as well being **associated** with **Regions** (**association**).

A **Region** is **associated** with **Organizations** as well as **Divisions**. A **Region** is made up of **Divisions** (**aggregation**).

A **Division** is part of a **Region** (**aggregation**) and can also be made up of other **Divisions** (**recursive aggregation**).

The **object diagram** below illustrates an example of the **instances** of the underlying classes that have been **instantiated** from the class diagram above.



The **objects** [AITP](#) and [Auldenfire](#) have been **instantiated** from the **class** **Organization**. Both of these objects share a **recursive association** with each other in that [Auldenfire](#) supports the [AITP](#) Java Competition Event.

The **objects** [United States](#) and [Europe](#) have been **instantiated** from the **class** **Region** and are **associated** with the **object** [Auldenfire](#).

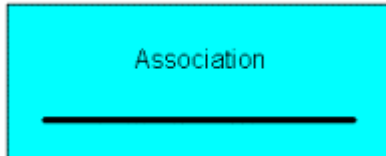
The **objects** [Auldenfire US](#), [Auldenfire SE](#), [Auldenfire TX](#), and [Auldenfire NV](#) have been **instantiated** from the **class** **Division**.

The objects [Auldenfire TX](#) and [Auldenfire NV](#) share a **recursive aggregation** with [Auldenfire US](#) ([Auldenfire TX](#) and [Auldenfire NV](#) are parts of [Auldenfire US](#)).

The **object** [Auldenfire US : Division](#) is a part of the **object** [United States : Region](#). The **object** [Auldenfire SE : Division](#) is part of the **object** [Europe : Region](#).

Forms of Class Association

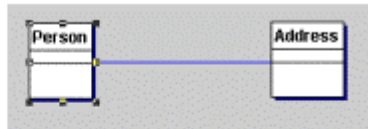
Classes and objects can be associated with each other and a class may be dependent upon another class. There are four forms of association: Association, Aggregation, Composition, and Generalization.



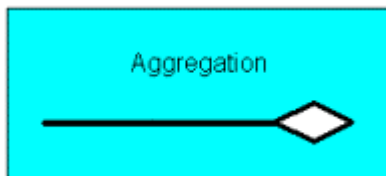
Association is a relationship between the instances of two classes. An association is implemented in Java as:

- One class containing a link to or an instance of another class
- One class creating an instance of another class
- One class sending a message to another class
- One class receiving a message containing another class

In the example illustration below, there is a linked association between Person and Address:



```
public class Person          public class Address
{                            {
  private Address lnkAddress;  }
}
```



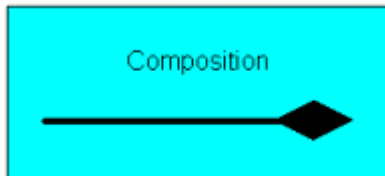
Aggregation is a form of [asymmetric association](#) that specifies a whole-part relationship between the aggregate (whole) class and a subordinate (part) class.

In the example illustration below, there is a whole-part relationship between the Car (whole) class and the Engine (part) class. In English, this would be stated as *Engine is a part of Car*.



```
public class Car          public class Engine
{
    /**
     * @link aggregation
     */
    private Engine lnkEngine;
}
```

There has been considerable confusion in how Aggregation and Composition have been defined in the past and not all tools support both of these special forms of association.



Composition is a form of [symmetric association](#) that specifies a whole-part relationship between the composition (whole) class and a subordinate (part) class in which removing the whole also removes the parts. In relational databases, a cascading delete is a good example of a composition relationship.

Composition is:

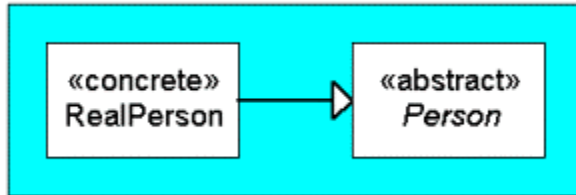
- A form of organization that complements classification
- Aggregation by value

In the example illustration below, there is a whole-part relationship between the Person (whole) class and the Head (part) class. In this example, removing the person also removes the head and removing the head effectively destroys the person - the person and head must exist together in the context of a living person. A Java program can be used to store attributes of the Person and Head.



```
public class Person      public class Head
{
    /**
     * @link aggregationByValue
     */
    private Head lnkHead;
}
```

Generalization is a relationship between classes that allows the sharing of properties defined within one class to be shared within subordinate classes. Generalization is used to build class hierarchies. Although it is a more abstract concept, generalization is also referred to as **inheritance**. The illustration below shows a concrete class inheriting from an abstract class.



Class RealPerson inherits from (extends) Class Person

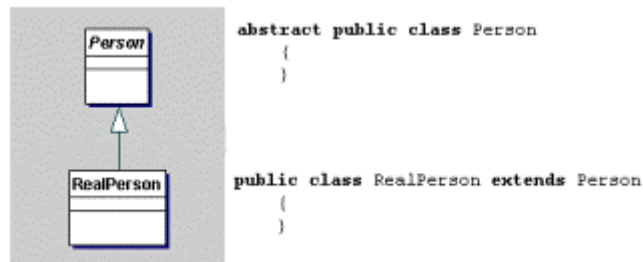
Generalization (or inheritance) can consist of two types of classes:

Abstract Classes cannot be directly instantiated and are only used for specification purposes. A class is abstract if it has no instances. An abstract class is only used to inherit from. Abstract classes are represented by an italicized class name.

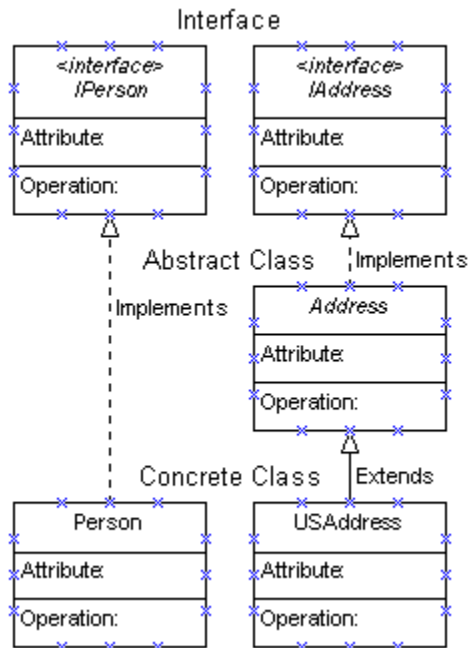
Concrete Classes can be directly instantiated to generate objects.

- An abstract class can inherit from other abstract classes.
- A concrete class can inherit from both abstract and other concrete classes

In Java, generalization is represented by the **extends** keyword as illustrated in the TogetherJ example below.



Interface is a mechanism used by Java to conveniently package and reuse a collection of methods (method signatures) and constants. An interface is an abstract class that only contains method signatures and can also contain constants. There is no underlying source code that defines the methods. Essentially, a Java Interface is a promise to implement a standard package of methods and constants - how the implementation is actually carried out depends on the programmer. The illustration below shows two examples of an interface.



An **interface** contains **method signatures** and may also contain **constants**.
An **interface** is **implemented**.

An **abstract class** may implement an interface. A **concrete class** may also implement an interface.
An **abstract class** is **extended**.

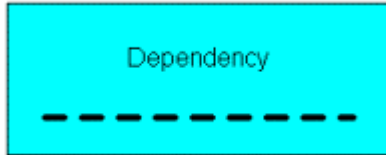
A **concrete class** may **extend** an **abstract class** that in turn **implements** an **interface**.

An interface may be implemented by one or more classes (both abstract and concrete). A class (abstract or concrete) may implement one or more interfaces.

Both **abstract classes** and **concrete classes** can **implement** an **interface**.

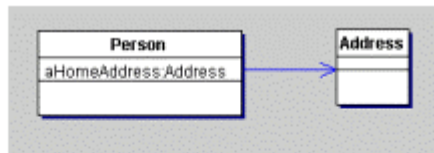
Interfaces provide extensibility to Java. As a side note, interfaces are an alternative to multiple inheritance, which Java does not support. Interfaces provide looser coupling, support design by composition, and provide plug-in points that allow a design to have greater flexibility.

Class Dependency



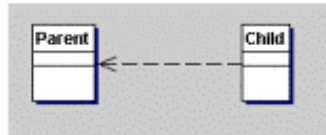
Dependency is a form of **association** that specifies a dependency relationship between two classes. An arrowhead can be used to indicate an asymmetric dependency.

The UML diagram and Java source code example below illustrate **dependency**. In this example, the **Address** class is **associated** with the **Person** class and **Person** **depends** on **Address** for providing **aHomeAddress**. This is an example of an **asymmetric dependency**.



```
public class Person          public class Address
{
    Address aHomeAddress = new Address;
}
}
```

The UML diagram and Java source code example below illustrate **dependency**. In this example, the **Child** class depends on the **Parent** class. A **symmetric dependency** does not use a directional arrowhead. This is an example of an **asymmetric dependency**.



```
public class Parent        public class Child
{
}                          {
                          /** @link dependency */
                          /*#Parent linkParent;*/
                          }
}
```

Class Cardinality and Multiplicity Notation

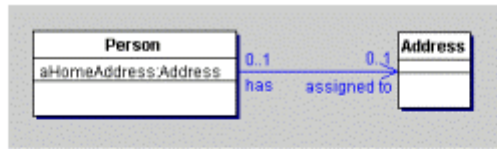
Class Multiplicity notation is shown in the diagram below:

Multiplicity	
n	n
a..z	a..z
+	+
0..*	0..*
0..n	0..n

Class Cardinality is defined in the diagram below:

Cardinality	
n	Exactly n
a..z	Numerically Specified (a to z)
+	Many (zero or more)
0..*	Optional (zero or more)
0..n	Optional (zero or n)
0..1	Optional (zero or 1)

The illustration below shows how class **cardinality** and **multiplicity** are notated in **UML** and in the **Java source code** by TogetherJ. The arrow pointing to the [Address](#) class from the [Person](#) class indicates that [Person](#) **depends** on [Address](#). This notation also indicates that the [Person](#) class has knowledge of the [Address](#) class; however, in this **asymmetric dependency**, [Address](#) has no knowledge of [Person](#).

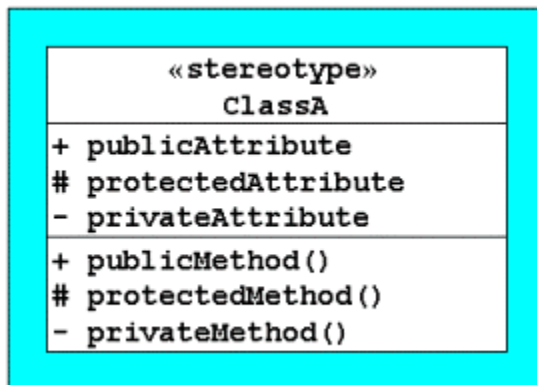


```
public class Person
{
    /**
     * @clientCardinality 0..1
     * @supplierCardinality 0..1
     * @clientRole has
     * @supplierRole assigned to
     */
    Address aHomeAddress = new Address;
}

public class Address
{
}
```

Class Visibility and Scope

A class is divided into three regions: Name, Attributes, and Operations. Each attribute and operation, referred to as a member, is assigned a scope: Public, Protected, or Private. The scope determines the level at which a member can be accessed.

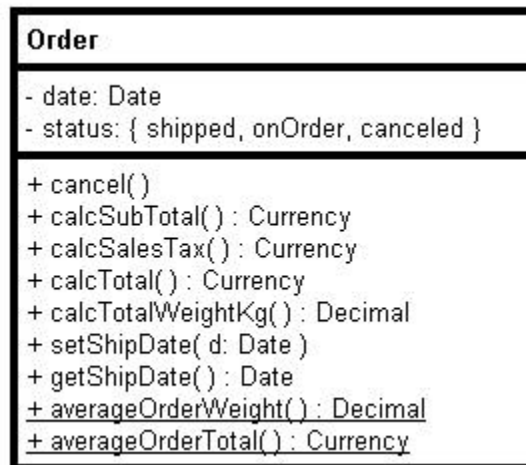


Class Name

Class Attributes

Class Operations

Public members can be accessed by any other class. Protected members can only be accessed by classes that belong to the same package. Private members can only be accessed by the class itself. The class **Order** is used as an example below:



Class Scope or Static members (attributes and operations) are underlined.
Instance members are not underlined.

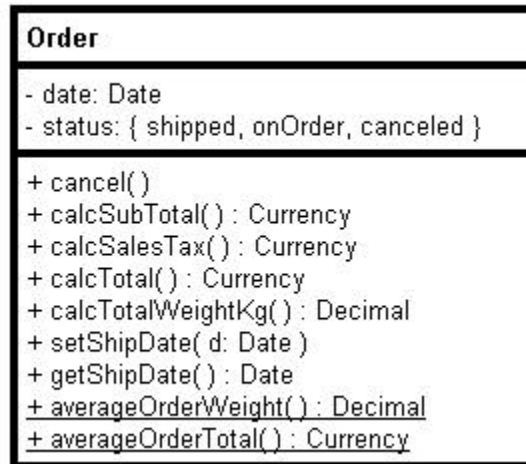
- The attribute **date** is **private**
- The operation **calcTotal** is **public**
- The operation **averageOrderTotal** is **public** with a **class scope (static)**

Static members are not instantiated with the object, they are available through the class itself.

Class Operations

UML Class Diagrams use Class Operators to illustrate class attributes and methods and follow the form:

Access Specifier, Class Name, Parameter List, Return Type



Using the class **Order** illustrated above, three Class Operation examples are provided in the table below.

<access specifier>	<name>	(<parameter list>) :	<return type>
+	setShipDate	(d: Date)	
+	getShipDate		Date
+	calcTotal		Currency

Access specifiers appear in front of each member (+, #, or -).
The parameter list shows each parameter type preceded by a colon.
Static members are underlined. Instance members are not.