UML Class Diagram Summary

A class diagram in UML is a type of static structure diagram that describes the structure of a system by showing the system’s classes, their attributes and the relationships between them.

This document provides a summary of the main features of a class diagram. Please consult a UML textbook for further details.

Class Structure

Each class can have a name, a list of attributes and a list of operations, as shown below. Each row represents a compartment.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>&lt;&lt; stereotype &gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
</tr>
</thead>
</table>
- numDoors : int
- license: String

<table>
<thead>
<tr>
<th>Operations</th>
</tr>
</thead>
</table>
+ Car()
+ getNumDoors() : int
+ setNumDoors(doors: int)
+ getLicense() : String
+ setLicense(theLicense: String)

The top compartment contains the class name, e.g. Car, and an optional stereotype. A stereotype describes an additional characteristic of the UML item. The stereotype is a word enclosed in « and » characters (called guillemets) or if those characters aren’t available, simply << and >>. Example stereotypes that you have seen are <<interface>> and <<enumeration>>. You may also see another property in the top compartment that is used to indicate additional information; an example later in this document uses {abstract}.

The middle compartment contains the attributes (instance variables) for the class. Each attribute has an accessibility operator, a name and a type. Note, the UML way to write attributes is different to how you would write the same thing in a particular programming language. For example, in UML, you could say that there is a private attribute called name that is of type String:

- name : String

In Java, this would be written as:

    private String name;

The bottom compartment contains a list of operations for the class. Each operation has an accessibility operator, a name, a list of parameters and an optional return type. Each parameter has a type.

Sometimes, a simpler view of the class might be drawn to show that other classes in the diagram use the class, but not provide all of the details in the particular diagram. This will contain the top compartment and the other two compartments will be empty.

You might also see examples where only the top compartment is shown.
Accessibility Operators
The class diagram will specify the accessibility for attributes and operations. The most common operators are:

+ public
# protected
- private

Abstract Classes and Methods
If a class contains an abstract method, you can either write the method name in italics or write {abstract} following the method. The same applies to the class name in the top compartment.

```
Pet
{abstract}

+ getSound()
+ otherMethod(one: String)
+ anotherMethod(): String {abstract}
```

Static Methods
Some designs make use of static methods – Java calls these class methods. These are methods that can be called without needing to create an instance first. The main method that starts a Java program is static. There are other times when a method might be static, but most methods/fields will not be static.

To show that a method is static, you underline the method signature. For example:

```
PetShop

+ PetShop()
+ main(args: String[])
+ openShop()
```

Associations
An association shows a link between classes; the main association type is a has-a/has-some relationship. The first type of association indicates that one class holds a reference to another class. The reference is held as an attribute in the original class.

This type of association uses a solid line to connect the two classes. Each end can include multiplicities, explained below, and the role that the class plays in the association.

```
Car
- numDoors: int
- license: String

Person
- name: String

<table>
<thead>
<tr>
<th>Car</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..* cars</td>
<td>1..1 owner</td>
</tr>
</tbody>
</table>
```

In the above example, the Person class includes an attribute called car that refers to the Car object that is referenced from the Person object. Also, the Car class includes an attribute owner that refers to a Person.
If you were to write these out as Java code, you would have the following two definitions:

```java
public class Car {
    private int numDoors;
    private String license;
    public Person owner;
}

public class Person {
    private String name;
    private Car[] cars;
}
```

Some points to note:

1. In the Car class, we have specified that the owner field is public (with the + symbol). This attribute has been set as public only for the purposes of this example. Normally we would not make the fields public, but we would also add get and set methods to the class.

2. The Person class can refer to 0 or more (0..*) Cars. That is, we are saying that a Person can own more than one car. To achieve this in code, we need a way to store more than one reference to a Car. We have used an array of cars in this example, but this isn’t the only way that you might choose to store a collection of items in your Java code.

**Multiplicities**

Common multiplicities are:

- 1 (or 1..1) – There is a one to one relationship. For example, each Car has only one owner.
- 0..1 – There can be 0 or 1 item.
- 1..* - There can be a reference to one or more items.
- * (or 0..*) - There can be a reference to many items or no item is referenced. For example, each Person can have zero or more cars.

**Navigability**

Sometimes you would indicate the navigability on one side of the association by adding an arrow. If you don’t include any navigability then it is assumed that you can navigate in either direction.

In the following example, a Person object can include references to multiple Car objects, but the car does not include any references to the same Person object.

```
<table>
<thead>
<tr>
<th>Car</th>
<th>0..* cars</th>
</tr>
</thead>
<tbody>
<tr>
<td>- numDoors: int</td>
<td>+ cars</td>
</tr>
<tr>
<td>- license: String</td>
<td></td>
</tr>
</tbody>
</table>
```

**Dependency**

A dependency states that somewhere in Class A, there is a reference to Class B. This differs from the main type of association in that the reference is either a local variable or an argument or return type to an operation.
A dependency is drawn using a dashed line. The navigability symbol would be added to one end of the line as appropriate.

**Inheritance**
Inheritance represents the *is-a* relationship, for example, a Student is a special kind of Person.

```
public class Person {
}

public class Student extends Person {
}
```

The same type of diagram is used to show that one class extends an abstract class. For example, a Cat is a special kind of Animal, where an Animal is an abstract class.

```
public abstract class Animal {
}

public class Cat extends Animal {
}
```

**Interfaces – Implementation**
There are two ways that we can represent the situation where a class implements an interface. The first is to use a dashed line with a closed arrowhead at the end that represents the interface.

The following example relates to the Animal interface that we used in the lectures. The Cat implements the Animal interface.
The corresponding code would be:

```java
public interface Animal {
    public String sound();
}

public class Cat implements Animal {
    @Override
    public String sound(){
        return "miaow";
    }
}
```

**Enumerations (enum)**

In Java, an enumeration is a way to have a type that has a defined number of possible values. In the lectures, we had a country code, which we said could be represented as an enum. This had the possible values UK, US, ES, FR.

In UML, we would write it as a class with two compartments. We would add the stereotype <<enumeration>> to the name; you might also see this as <<enum>>. The attributes compartment would list the possible values.

```
<<enumeration>>

CountryCode
UK
US
ES
FR
```

This would correspond to the Java code:

```java
public enum CountryCode {
    UK,
    US,
    ES,
    FR
}
```
Notes:

- There are two more types of association that some of you have seen last semester: aggregation and composition. Aggregation is where an object owns objects, but may share these with other parts of the system. Composition is where an object owns a set of objects and the object is responsible for the lifetime of the objects. Aggregation and Composition are currently outside the scope of this document.

- In the diagrams above, we have generally drawn the associations horizontally. As you will see in the slides for the module, the lines can be drawn in any direction. The key thing is that the line connects two classes. Generally, we try to avoid lines crossing each other, but this is not always possible on more complex diagrams.

---

**Document History**

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2011-02-22</td>
<td>Neil Taylor, Lynda Thomas</td>
<td>Original version, based on original work by Lynda Thomas.</td>
</tr>
<tr>
<td>1.1</td>
<td>2012-03-24</td>
<td>Neil Taylor</td>
<td>Updated to revise information about abstract classes. Added information about static methods.</td>
</tr>
<tr>
<td>1.2</td>
<td>2012-03-30</td>
<td>Neil Taylor</td>
<td>Corrected information about abstract classes.</td>
</tr>
</tbody>
</table>

---