Object Oriented Design

Class Diagrams

Dr. Seán Russell

School of Computer Science,
University College Dublin

Lecture 03

Table of Contents

1. The Object Model
2. Representing Classes in UML
3. Associations
4. Aggregation and Composition
5. Generalisation and Specialisation
6. Class Members
7. Singletons
8. Types of Classes

Section Contents

1. The Object Model
   - Core Concept
   - Objects
   - Object Graph
   - Classes
   - UML Diagrams

- Programming languages and design languages both tell us about programs
- These facts are expressed at different levels of abstraction
- Both are based on the abstract idea of running programs that we will call **the object model**

- The object model is not a specific UML model
  - It refers to how we think about programming in OOP
- The fundamental concept in the object model is that **computation takes place in and between objects**
  - Individual objects are responsible for maintaining part of the systems data
  - Individual objects are responsible for implementing part of the systems functionality

- Objects are often represented as an area in memory containing some data
- One important points is that much data is represented in the relationships between objects
- In the object model, objects are often viewed as a graph of connected objects
- These object communicate by **sending messages**

- It is normally too complicated to write programs by defining individual objects
  - OOP can be done without classes in some languages
- Classes are used to define the common responsibilities of objects
- These classes describe **how an object can be used**

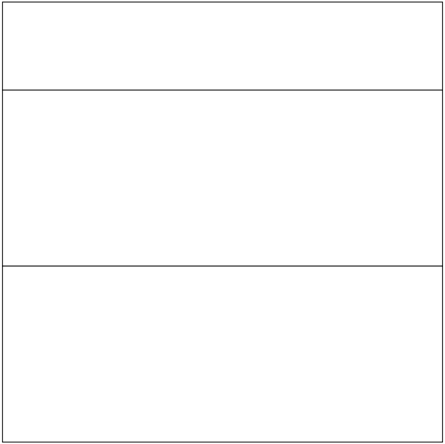
- There are two main types of UML diagram
  - **Static** diagrams describe the properties the object network can have
  - **Dynamic** diagrams describe what happens in the object network

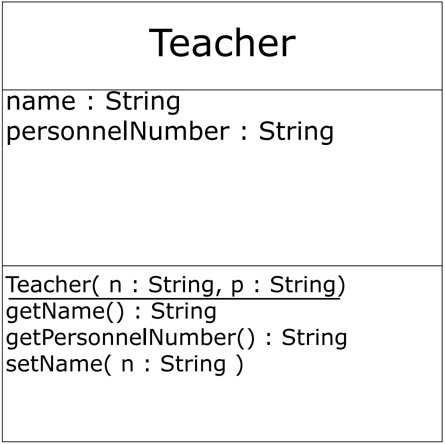
Section Contents

2 Representing Classes in UML

- An Empty Class
- A Teacher Class
- Notes
- Visibility

- Class diagrams in UML are made up of classes, their information and the connections between them
- For every class, we can represent three things:
  1. The **name** of the class
  2. The **attributes** it contains
  3. The **operations** it can perform





Teacher

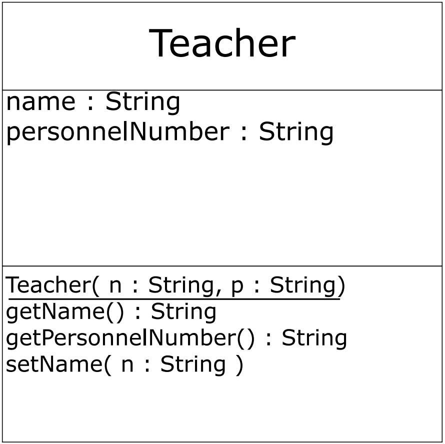
name : String
personnelNumber : String

Teacher(n : String, p : String)
getName() : String
getPersonnelNumber() : String
setName(n : String)

```
public class Teacher {
    String name;
    String personnelNumber;

    Teacher(String n, String p){
        name = n;
        personnelNumber = p;
    }

    String getName(){
        return name;
    }
    String getPersonnelNumber(){
        return personnelNumber;
    }
    void setName(String n){
        name = n;
    }
}
```



Teacher

name : String
personnelNumber : String

Teacher(n : String, p : String)
getName() : String
getPersonnelNumber() : String
setName(n : String)

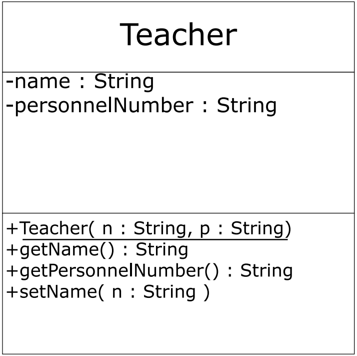
- The attributes section is basically a list of the instance variables in class
  - But this does not include complex objects
- The operations section is a list of the names, return types and parameters for all of the methods in the class
  - But this does not tell us what they do or how
  - Constructors are underlined

- We can represent the visibility of attributes and operations
- There are the same 4 levels as in Java
    - public $+$
    - package $\sim$
    - protected $\#$
    - private -
- The symbol is placed before the name of the attribute or operation

```java
public class Teacher {
    private String name;
    private String personnelNumber;

    public Teacher(String n, String p){
        name = n;
        personnelNumber = p;
    }

    public String getName(){
        return name;
    }
    public String getPersonnelNumber(){
        return personnelNumber;
    }
    public void setName(String n){
        name = n;
    }
}
```



Teacher

-name : String
-personnelNumber : String

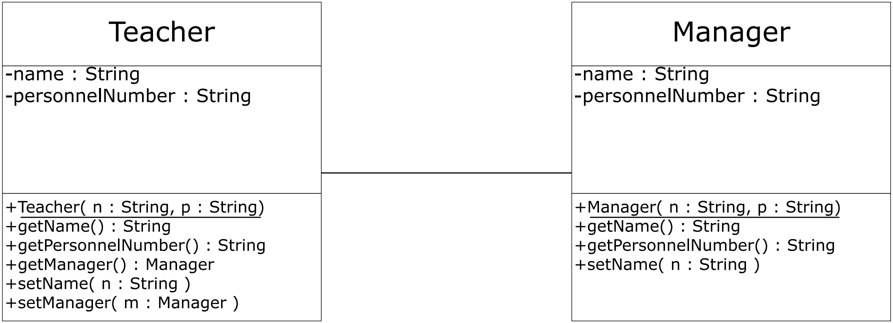
+Teacher(n : String, p : String)
+getName() : String
+getPersonnelNumber() : String
+setName(n : String)

Section Contents

3 Associations

- Instance Variables
- Annotations - Association Name
- Annotation - Role Name
- Annotation - Multiplicity
- Annotation - Navigability
- Example

- Attributes in a UML class do not include the instance variables that are complex objects
- These are instead represented as connections between classes
- The connections are called **associations**
- They are shown as a line connecting two classes
  - Can be annotated with **name**, **role name**, **multiplicity**, and/or **navigability**



Teacher

-name : String
-personnelNumber : String

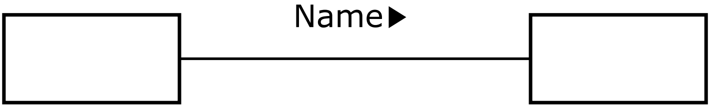
+Teacher(n : String, p : String)
+getName() : String
+getPersonnelNumber() : String
+getManager() : Manager
+setName(n : String)
+setManager(m : Manager)

Manager

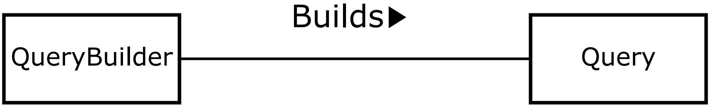
-name : String
-personnelNumber : String

+Manager(n : String, p : String)
+getName() : String
+getPersonnelNumber() : String
+setName(n : String)

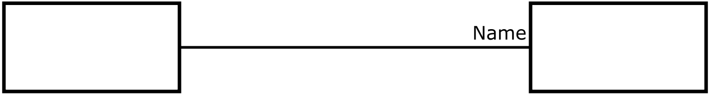
- The name of an association is shown above the line approximately in the middle



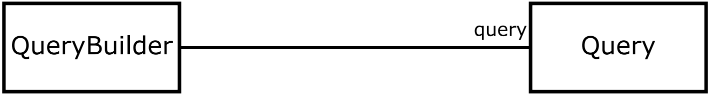
- The name usually explains the association in some way
- The triangle shows the direction the association should be read



- The concept of **role name** or association end name is supposed to describe the role played by classes in the relationship
- In reality, it is most commonly used to show the name of the instance variable used to implement the association

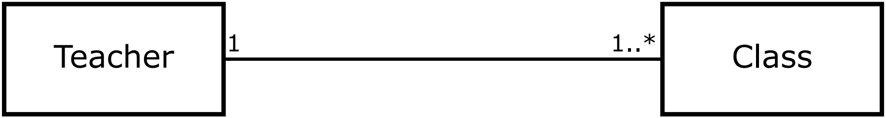


- The name is placed at the opposite end of the association to the class it belongs to



- Role names may also include visibility

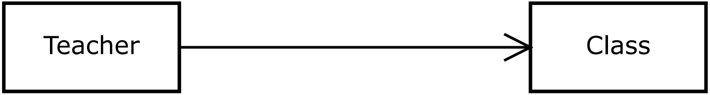
- The multiplicity on an association tells us how many objects are connected
- You will normally see one of the following options
  - Zero or Many **(0..*)** or **(*)**
  - Zero or One **(0..1)**
  - Only One **(1)**
  - One or Many **(1..*)**
- The multiplicity is placed at the opposite end of the association from the class it applies to



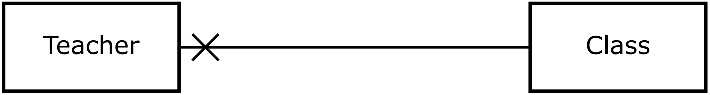
- The multiplicities above give us the following facts
  - Each teacher may be teaching one or more classes
  - Each class is taught by exactly one teacher

- Navigability is the concept of which directions the association can be used in
- More specifically, it tells us which objects can communicate
    - Communication is done by calling methods
- If navigability is not shown, then we do not know what is possible

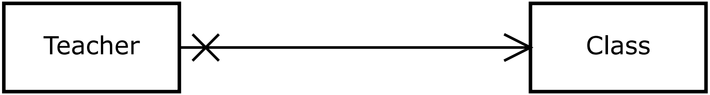
- This diagram shows that Teacher can communicate with Class

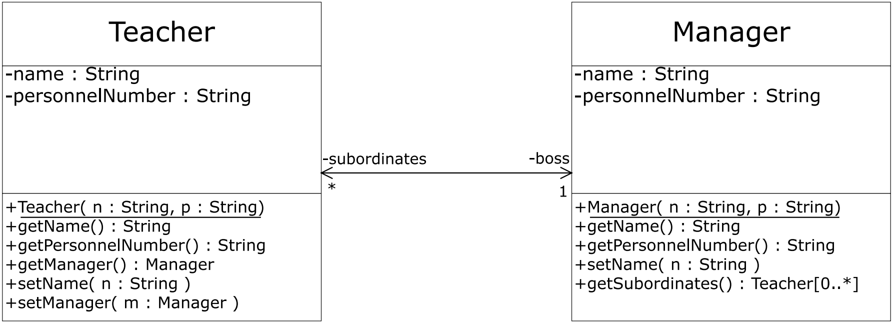


- This diagram shows that Class cannot communicate with Teacher



- This diagram shows both of the above





Teacher

-name : String
-personnelNumber : String

+Teacher(n : String, p : String)
+getName() : String
+getPersonnelNumber() : String
+getManager() : Manager
+setName(n : String)
+setManager(m : Manager)

-subordinates *

-boss 1

Manager

-name : String
-personnelNumber : String

+Manager(n : String, p : String)
+getName() : String
+getPersonnelNumber() : String
+setName(n : String)
+getSubordinates() : Teacher[0..*]

```java
public class Teacher {
    private String name;
    private String personnelNumber;
    private Manager boss;

    public Teacher(String n, String p){ ... }

    public String getName(){ ... }
    public String getPersonnelNumber(){ ... }
    public Manager getManager(){ ... }

    public void setName(String n){ ... }
    public void setManager(Manager m){ ... }
}
```

```java
public class Manager {
    private String name;
    private String personnelNumber;
    private List<Teacher> subordinates;

    public Manager(String n, String p){ ... }

    public String getName(){ ... }
    public String getPersonnelNumber(){ ... }
    public void setName(String n){ ... }
}
```

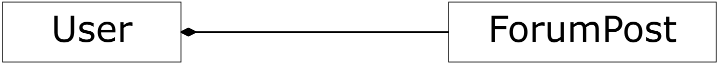
Section Contents

4 Aggregation and Composition

- Composition
- Aggregation
- Example of Composition and Aggregation Relationships

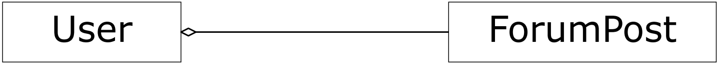
- UML allows us to define some different types of associations
- These will not always result in differences in the actual code
- But they will give some information about how objects in the relationship should be used
- These different types of association are know as **aggregation** and **composition**

- Composition is shown by adding a filled diamond to the association

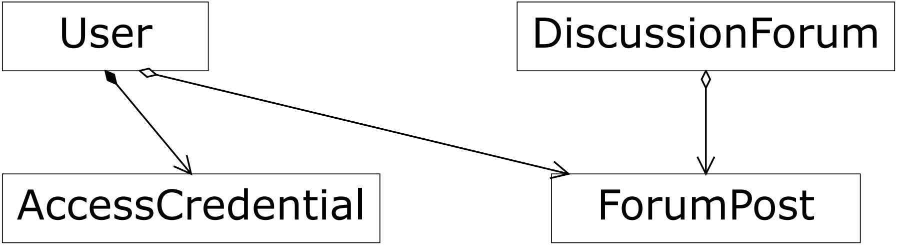


- Composition implies a level of ownership or possession (I.e. the `User` object owns the `ForumPost` object)
- When objects share a composition relationship, they may be **created together** and should be **destroyed together**
- An object may only be part of one composition

- Aggregation is shown by adding a hollow diamond to the association



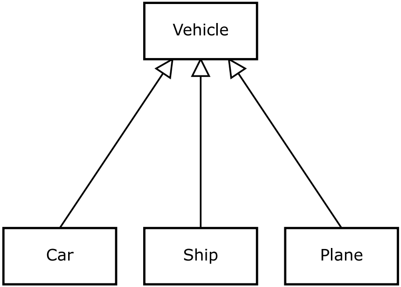
- Aggregation is similar to composition, but weaker and poorly defined
- In principle there is still a relationship where one object is considered part of another
- However, with aggregation an object may be part of multiple aggregations

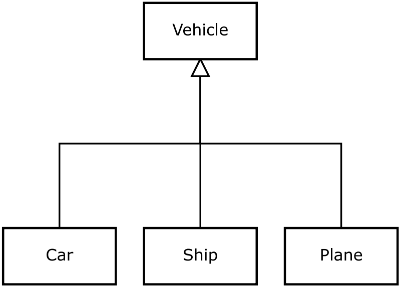

User
DiscussionForum
AccessCredential
ForumPost

Section Contents

5 Generalisation and Specialisation
- Generalisation Example
- Generalisation Implementation

- The relationship between classes we know as inheritance is called generalisation or specialisation in UML
- It is called **generalisation** in the direction of the superclass
  - The superclass is more general than the subclass
- It is called **specialisation** in the direction of the subclass
  - The subclass is more specialised than the superclass


Vehicle
Car
Ship
Plane


Vehicle
Car
Ship
Plane

```
public class Vehicle {
    ...
}
```

```
public class Car extends Vehicle {
    ...
}
```

```
public class Ship extends Vehicle {
    ...
}
```

Section Contents

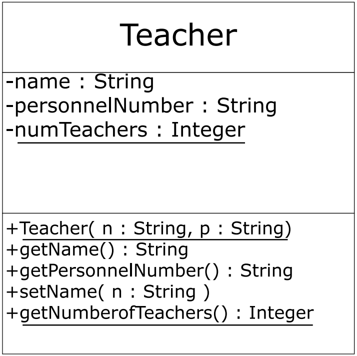
6 Class Members
- Static Members

- UML allows us to show when variables or methods belong to the class
- This is done by underlining the attribute or method
- Constructors are also considered as belonging to the class in UML

```java
public class Teacher {
    private String name;
    private String personnelNumber;
    private static int numTeachers;

    public Teacher(String n, String p){ ... }

    public String getName(){ ... }
    public String getPersonnelNumber(){ ... }
    public void setName(String n){ ... }
    public static int getNumberofTeachers(){ ... }
}
```



Teacher

-name : String
-personnelNumber : String
-numTeachers : Integer

+Teacher(n : String, p : String)
+getName() : String
+getPersonnelNumber() : String
+setName(n : String)
+getNumberofTeachers() : Integer

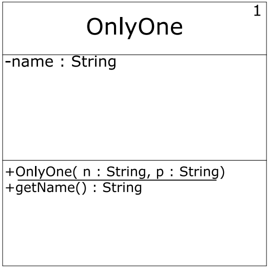
Section Contents

7 Singletons

- Class Multiplicity
- Showing a Class is a Singleton
- Implementing Class as a Singleton
- Using A Singleton Object
- When to use a Singleton

- Multiplicity can also be applied to an individual class
- This can describe how many objects may be created based on this class
- The only important case is where classes are limited to a single instance
- Classes like this are known as **singletons**

- A class that is limited to a single instance is shown by adding a 1 to the top right hand corner



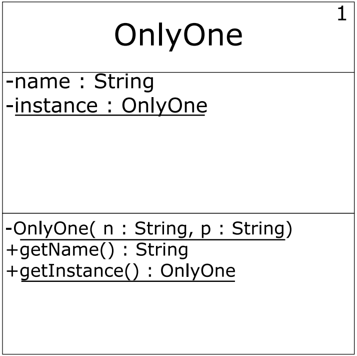
The Singleton Pattern

```java
public class OnlyOne {
    private String name;
    private static OnlyOne instance;

    private OnlyOne(){

    }

    public static OnlyOne getInstance(){
        if (instance == null){
            instance = new OnlyOne();
        }
        return instance;
    }
    public String getName(){
        return name;
    }
}
```



1

OnlyOne

-name : String
-instance : OnlyOne

-OnlyOne(n : String, p : String)
+getName() : String
+getInstance() : OnlyOne

- When we want to use a singleton object, we cannot construct it like a normal object
- Instead, we must use the static `getInstance` method
- E.g. `OnlyOne oo = OnlyOne.getInstance();`

- Having an object that can be accessed from everywhere in our code can simplify things, but it also can have down sides
- Disadvantages:
  - We need to consider concurrency
  - Singletons make testing harder
  - They can hide dependencies in our code
- Typical uses:
  - Logging
  - Caches
  - Read-only application state

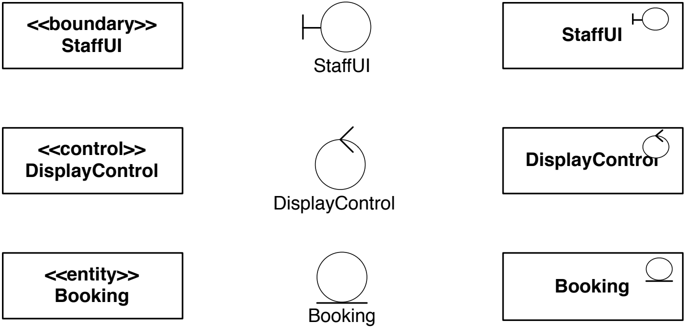
Section Contents

8 Types of Classes

- Boundary, Control and Entity objects

- UML distinguishing between **boundary**, **control** and **entity** objects
  - Entity objects are responsible for maintaining data
  - Boundary objects are those which interact with external users
  - Control objects ensure the interactions in the program happen correctly
- These are ordinary classes with an additional description of their role in the system
  - Represented in UML by stereotypes

- The stereotype can be written in the class icon, but distinctive icons are also defined for the three types of class


<<boundary>>
StaffUI
StaffUI
StaffUI
<<control>>
DisplayControl
DisplayControl
DisplayControl
<<entity>>
Booking
Booking
Booking