UML 1/2
Class Diagrams & Associations

Object Modelling

Let us consider the real world:
It consists of entities (i.e. objects) which inter-relate with each other.

- A problem in the real world can be modelled as a set of inter-relating objects.
- Such a model should make it straightforward to accurately capture user requirements.

A central belief in object-oriented development is that the objects identified when analysing a problem can be used when we are creating a solution (i.e. creating a design and an eventual implementation).

This is an over-statement.

However, a major advantage of object-oriented development is that we use the same approach (i.e. a set of communicating objects) when trying to understand the problem, when designing a solution and when implementing the solution in a programming language.

Other Models

Object-oriented modelling is not the only way in which we can do modelling:

- You may, for example, hear about *SSADM*.
- Database people use *entity-relationship diagrams* (close to class diagrams).
- Also there are *dataflow diagrams*.
- And others…

One reason that object modelling is popular is that the models can be close to/based on reality:

- They can therefore be understood by non-experts.

Unified Modelling Language Models

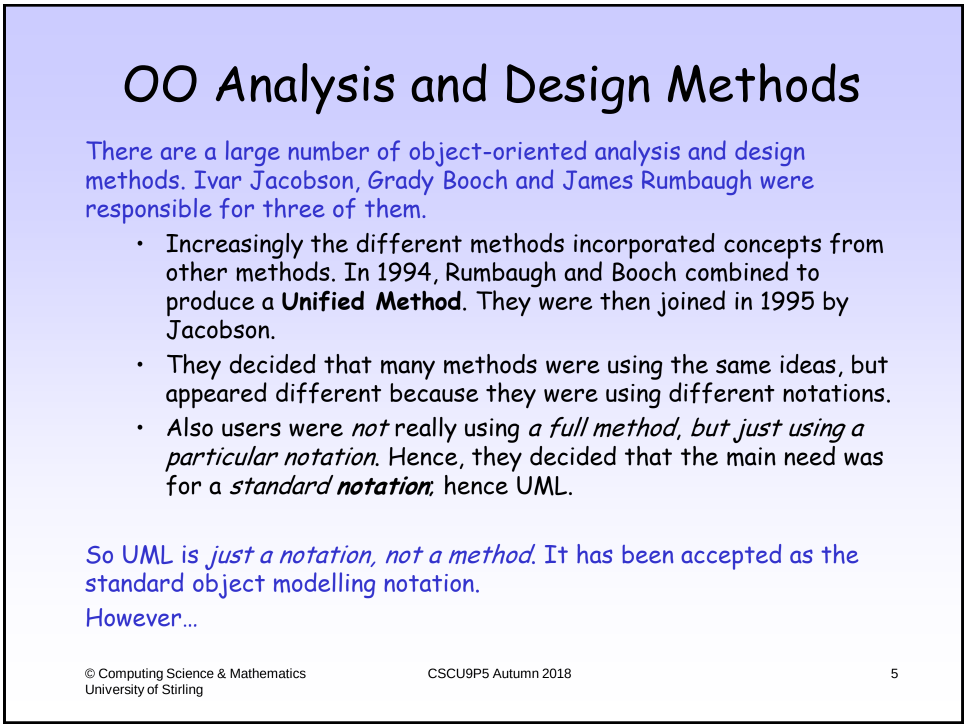
In UML, we do not create *one* kind of model, but a *set of models:*

- Each gives a different view of the problem or design
- The different models are then represented in diagrams

We have **structural models** which show how the different components fit together statically and **dynamic models** which describe behaviour during execution.

UML Diagrams

- The main UML structural diagram is the **class diagram**.
- Requirements are collected together using a **use case diagram**.
- Dynamic models are represented using **state diagrams** and **interaction diagrams** (**sequence diagrams** and **collaboration diagrams**)
- UML supports various other diagrams that we will not look at.



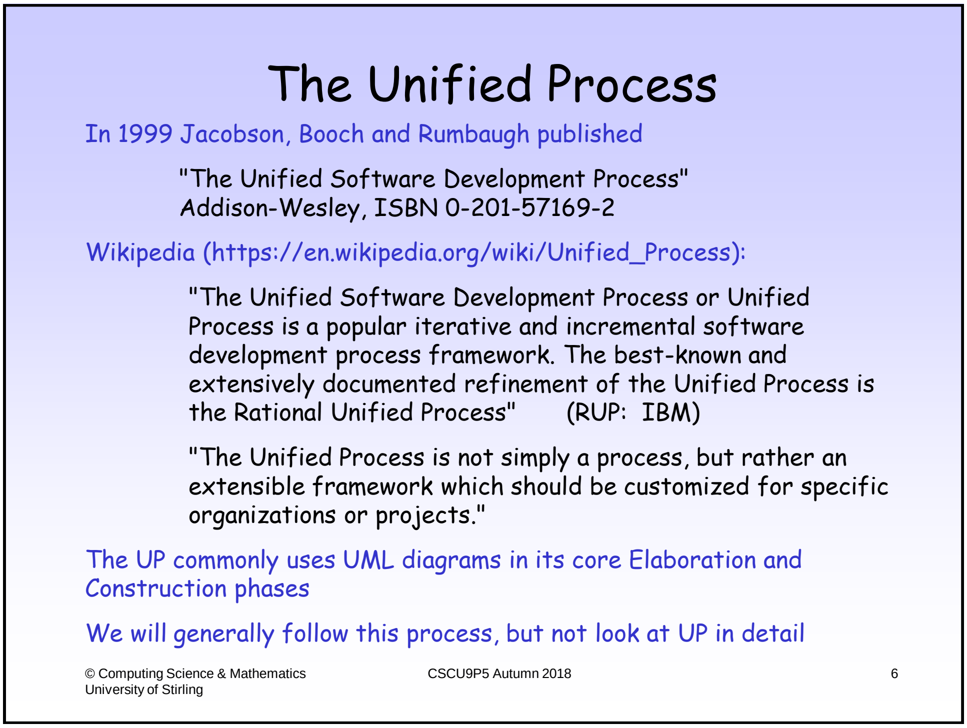
OO Analysis and Design Methods

There are a large number of object-oriented analysis and design methods. Ivar Jacobson, Grady Booch and James Rumbaugh were responsible for three of them.

- Increasingly the different methods incorporated concepts from other methods. In 1994, Rumbaugh and Booch combined to produce a **Unified Method**. They were then joined in 1995 by Jacobson.
- They decided that many methods were using the same ideas, but appeared different because they were using different notations.
- Also users were *not* really using *a full method, but just using a particular notation*. Hence, they decided that the main need was for a *standard* ***notation***; hence UML.

So UML is *just a notation, not a method*. It has been accepted as the standard object modelling notation.

However…



The Unified Process

In 1999 Jacobson, Booch and Rumbaugh published

"The Unified Software Development Process"
Addison-Wesley, ISBN 0-201-57169-2

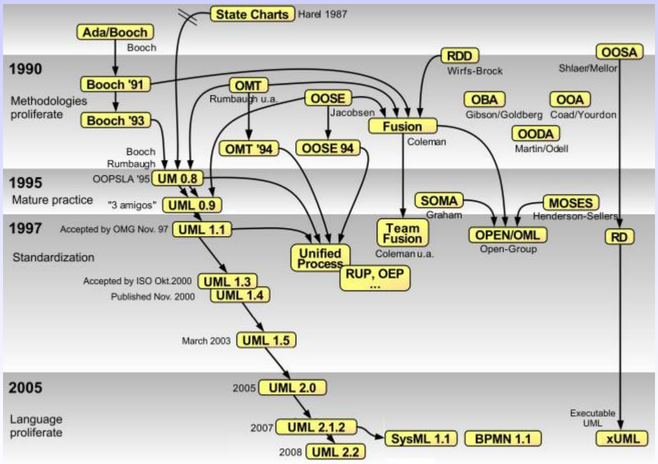
Wikipedia (https://en.wikipedia.org/wiki/Unified_Process):

> "The Unified Software Development Process or Unified Process is a popular iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process" (RUP: IBM)
>
> "The Unified Process is not simply a process, but rather an extensible framework which should be customized for specific organizations or projects."

The UP commonly uses UML diagrams in its core Elaboration and Construction phases

We will generally follow this process, but not look at UP in detail

UML Timeline



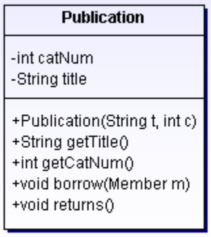
Representing a class in UML

A class is represented in a **class diagram** as a rectangular box divided into three parts

- Name
- Attributes (variables, fields, data)
- Operations for offered services

Access to *attributes* is normally *restricted* (*private*) and they are therefore hidden from external objects. This is represented in UML by the prefix '-'.

The *operations* are *visible* to external objects (public) and so are prefixed with a '+'. For example:

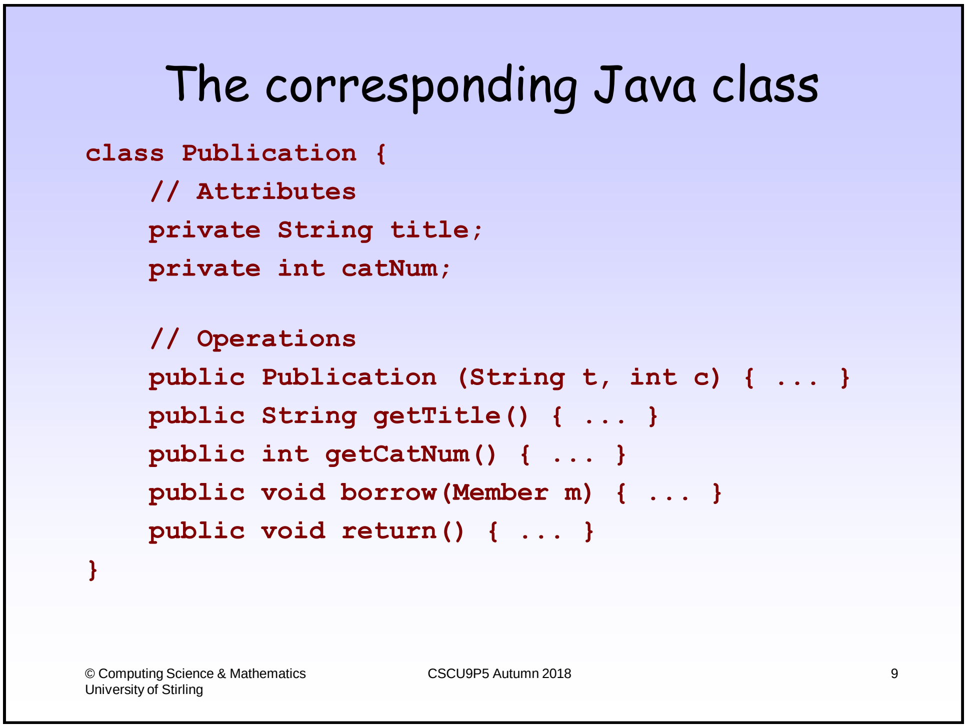


Publication

-int catNum
-String title

+Publication(String t, int c)
+String getTitle()
+int getCatNum()
+void borrow(Member m)
+void returns()

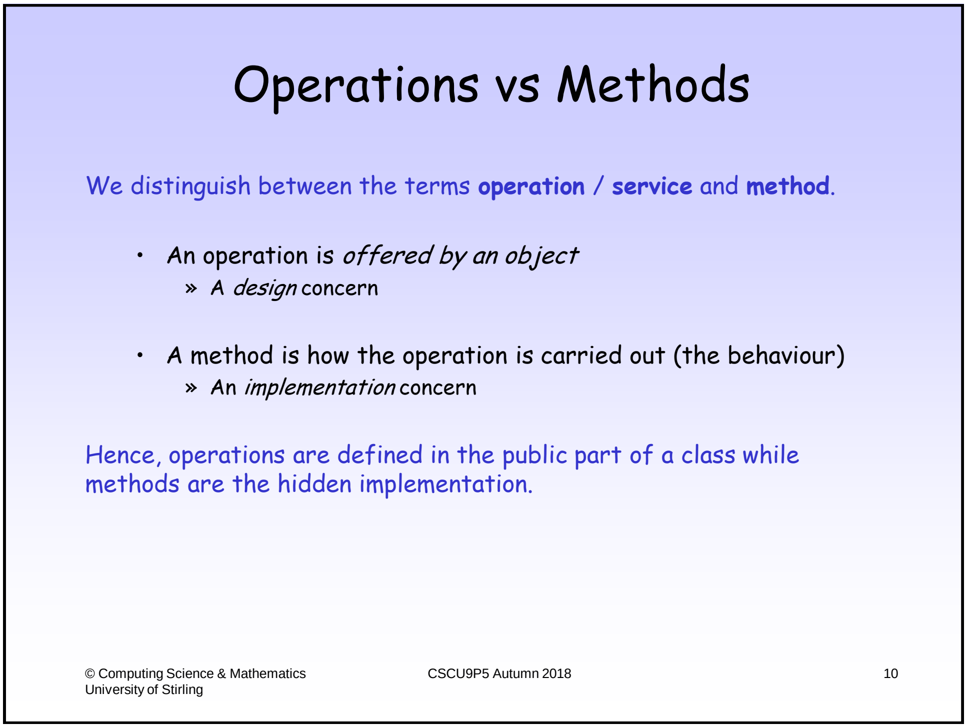
Note: There may also be private operations, but these are an *implementation* and *not a design* issue



The corresponding Java class

```
class Publication {
    // Attributes
    private String title;
    private int catNum;

    // Operations
    public Publication (String t, int c) { ... }
    public String getTitle() { ... }
    public int getCatNum() { ... }
    public void borrow(Member m) { ... }
    public void return() { ... }
}
```



Operations vs Methods

We distinguish between the terms **operation** / **service** and **method**.

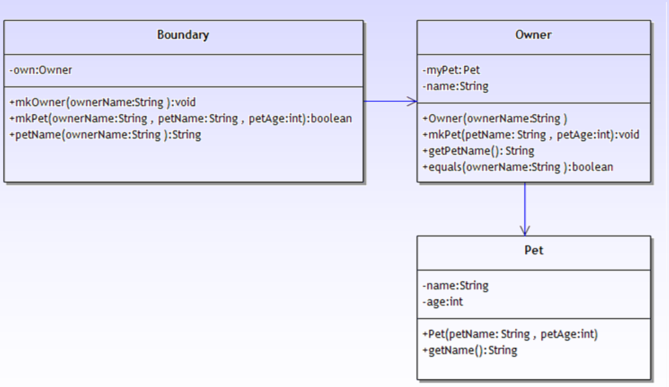
- An operation is *offered by an object*
  - » A *design* concern
- A method is how the operation is carried out (the behaviour)
  - » An *implementation* concern

Hence, operations are defined in the public part of a class while methods are the hidden implementation.

The OnePet Example - Class Diagram

Individual classes are combined in a UML *class diagram* to show the structure of the *overall program*:

"Associations" (relationships) between nodes are represented by arrows



Boundary

-own:Owner

+mkOwner(ownerName:String):void
+mkPet(ownerName:String , petName: String , petAge:int):boolean
+petName(ownerName:String):String

Owner

-myPet: Pet
-name:String

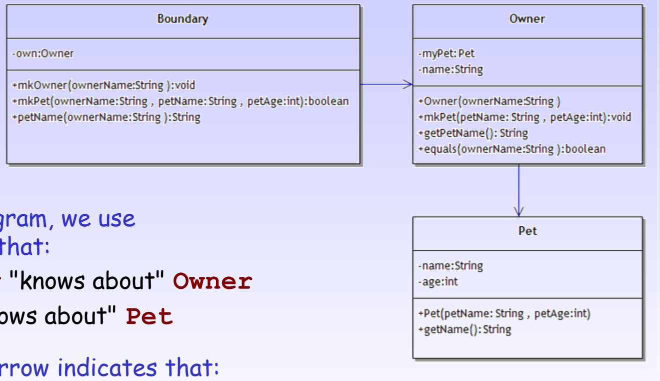
+Owner(ownerName:String)
+mkPet(petName: String , petAge:int):void
+getPetName(): String
+equals(ownerName:String):boolean

Pet

-name:String
-age:int

+Pet(petName: String , petAge:int)
+getName():String

The OnePet Example - Class Diagram



Boundary

-own:Owner

+mkOwner(ownerName:String):void
+mkPet(ownerName:String , petName: String , petAge:int):boolean
+petName(ownerName:String):String

Owner

-myPet: Pet
-name:String

+Owner(ownerName:String)
+mkPet(petName: String , petAge:int):void
+getPetName(): String
+equals(ownerName:String):boolean

Pet

-name:String
-age:int

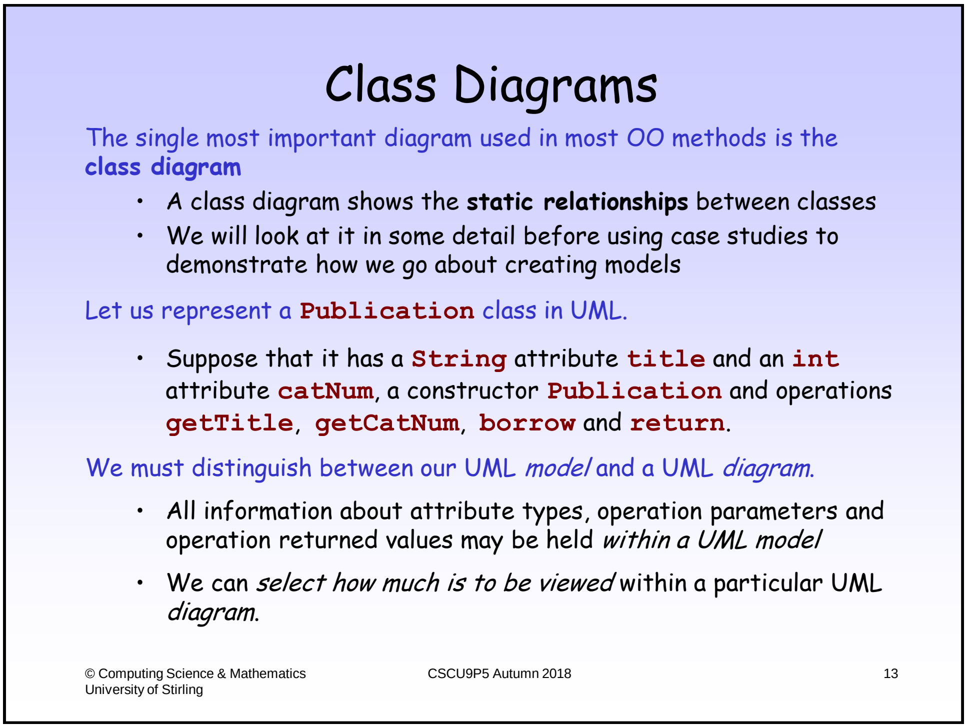
+Pet(petName: String , petAge:int)
+getName():String

In the class diagram, we use arrows to show that:

- `Boundary` "knows about" `Owner`
- `Owner` "knows about" `Pet`

Concretely, an arrow indicates that:

- A `Boundary` object has an attribute (`own`) referencing an `Owner` object and can send messages *to* that object
- An `Owner` object has an attribute (`myPet`) referencing a `Pet` object and can send messages *to* that object



Class Diagrams

The single most important diagram used in most OO methods is the **class diagram**

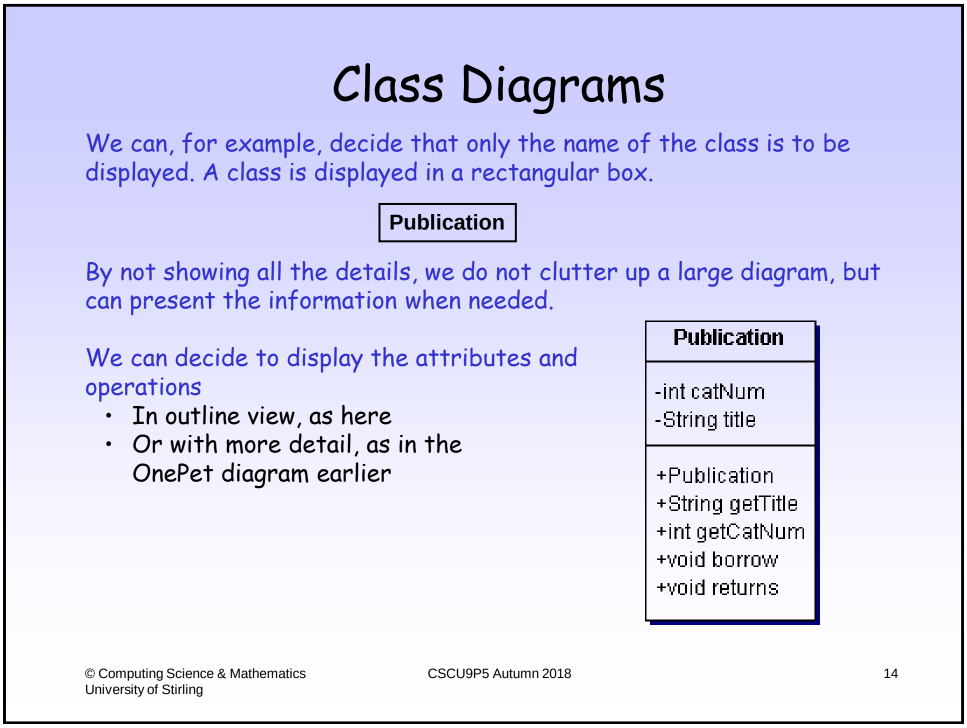
- A class diagram shows the **static relationships** between classes
- We will look at it in some detail before using case studies to demonstrate how we go about creating models

Let us represent a **`Publication`** class in UML.

- Suppose that it has a **`String`** attribute **`title`** and an **`int`** attribute **`catNum`**, a constructor **`Publication`** and operations **`getTitle`**, **`getCatNum`**, **`borrow`** and **`return`**.

We must distinguish between our UML *model* and a UML *diagram*.

- All information about attribute types, operation parameters and operation returned values may be held *within a UML model*
- We can *select how much is to be viewed* within a particular UML *diagram*.



Class Diagrams

We can, for example, decide that only the name of the class is to be displayed. A class is displayed in a rectangular box.

Publication

By not showing all the details, we do not clutter up a large diagram, but can present the information when needed.

We can decide to display the attributes and operations

- In outline view, as here
- Or with more detail, as in the OnePet diagram earlier

Publication

-int catNum
-String title

+Publication
+String getTitle
+int getCatNum
+void borrow
+void returns

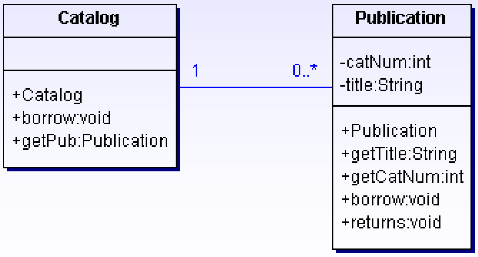
Associations

A class diagram shows the static structure of a system by showing the classes and their *associations*

- Indications that they "work together" or are *related* in some way.

Let us extend our example: for example a library system could have an on-line catalogue containing *a number of different publications*

- We can show that there is an association/relationship between objects in a new **`Catalog`** class and our **`Publication`** class by drawing a line:



Catalog

+Catalog
+borrow:void
+getPub:Publication

1

0..*

Publication

-catNum:int
-title:String

+Publication
+getTitle:String
+getCatNum:int
+borrow:void
+returns:void

Associations - Multiplicity

We can optionally give a *multiplicity* to an association

- Here, the **`Catalog`** object is associated with zero or more **`Publication`** objects (shown by `0..*`)
- And each **`Publication`** object is associated with exactly one **`Catalog`** object.

Common multiplicities are:

1	One instance
0..1	Zero or one instance
0..*	Zero or more instances
1..*	One or more instances

When no multiplicity is given, it is assumed to be 1.

Implementation of Associations

In a programming language, an association can be **implemented** as an attribute

- For example, an implementation of `Catalog` could have a list of `Publication` objects as an attribute
- However, in our *design model*, we might just show the association graphically (with an implicit attribute)
- A UML IDE (eg Together Architect) might allow us to enable display of the attribute as well

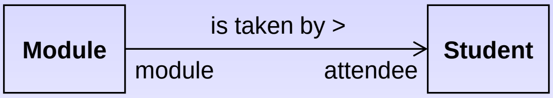
Or perhaps a `Publication` has an attribute referring to the `Catalog`??

- At some point we must decide whether `Catalogs` "know about" `Publications`, and/or `Publications` "know about" `Catalogs`
- This is known as **navigability**, i.e. in which direction do objects refer or send messages from one object to another?
- *We often delay making such decisions until later*

Navigability suggests in which class an implementing attribute should be located

Other annotations on associations

We may add more information to associations to describe/document our evolving design more clearly:



An association may have a *label* describing the relationship, e.g. "is taken by"

- Clearest if navigability has been specified, but this is not necessary
- May have < or > to indicate direction of relationship
  (or implicit if navigability is shown)

Each end of an association may indicate the *role* of the class at that end, e.g. "attendee"

Navigability of Associations

At some point we must decide whether `Catalogs` "know about" `Publications`, and/or `Publications` "know about" `Catalogs`

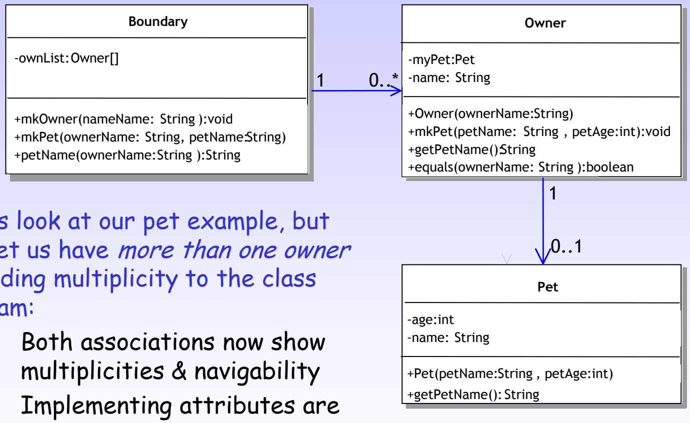
Once it has been decided:

Navigability is represented in a class diagram by adding an arrow to the line representing an association.

- We have the meaning that if no arrow is present then we are saying nothing (yet) about navigability
- If one arrow is present then we are saying that the association is navigable only in that direction

Non-navigability may be indicated by a X on the end of the association to which messages *may not* be sent

Multiplicity & navigability - Example



Boundary
-ownList:Owner[]
+mkOwner(nameName: String):void
+mkPet(ownerName: String, petName:String)
+petName(ownerName:String):String

Owner
-myPet:Pet
-name: String
+Owner(ownerName:String)
+mkPet(petName: String , petAge:int):void
+getPetName():String
+equals(ownerName: String):boolean

Pet
-age:int
-name: String
+Pet(petName:String , petAge:int)
+getPetName(): String

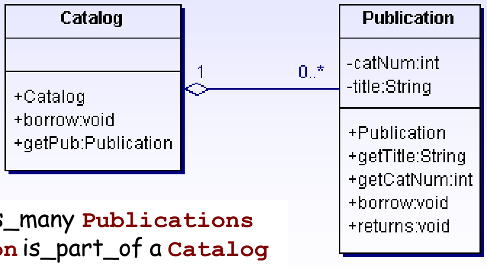
Let us look at our pet example, but now let us have *more than one owner* by adding multiplicity to the class diagram:

- Both associations now show multiplicities & navigability
- Implementing attributes are also displayed

Associations - Aggregation

Aggregation is a special kind of association representing a *structural relationship* between a whole and its parts.

- This can be thought of as a '**has_a**' or '**is_part_of**' relationship.
- It is not essential to use aggregation, but it can help us understand and give added meaning to a model.
- We could suggest that the relationship between `Catalog` and `Publication` is an aggregation in which case we would show:



A `Catalog` has_many `Publications`
A `Publication` is_part_of a `Catalog`

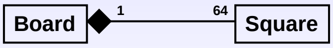
Associations - Composition

Composition is a *strong kind of aggregation*

- The parts can only belong to a single composition
- And if the composition is copied or deleted then *all the parts are copied or deleted with it.*

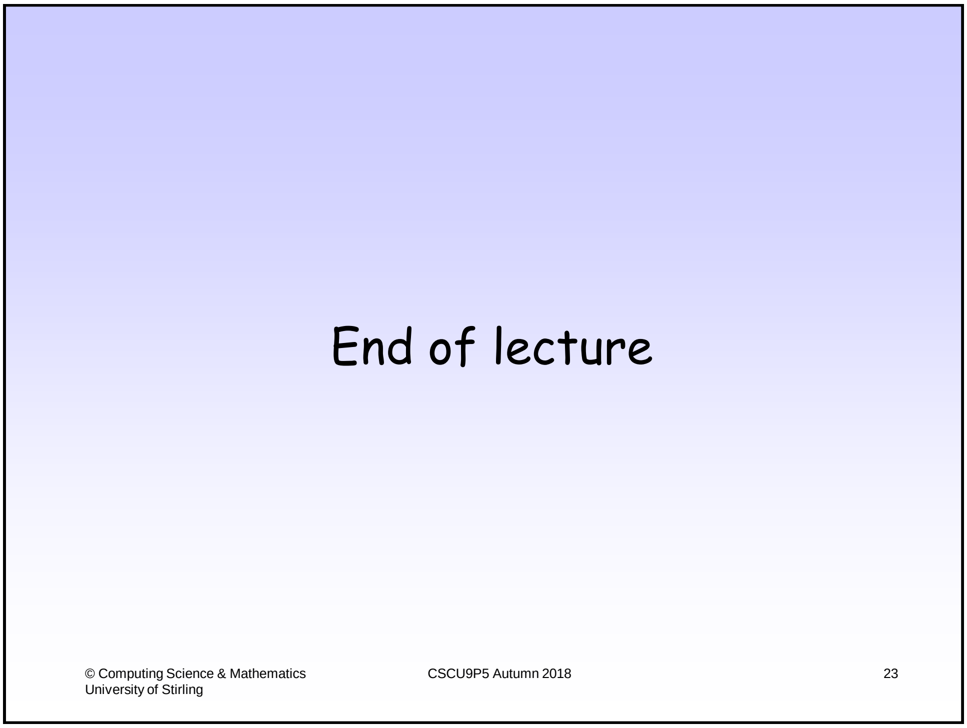
So, if we have a chess board, we could show that its squares make up the board by representing this as a composition relationship

- In composition, the diamond symbol is filled in:



This diagram states 'A board is composed of 64 Squares'.

- Note that here only the class name has been shown in each node.
- UML tools allow us to select how much information about a class is to be displayed.



End of lecture