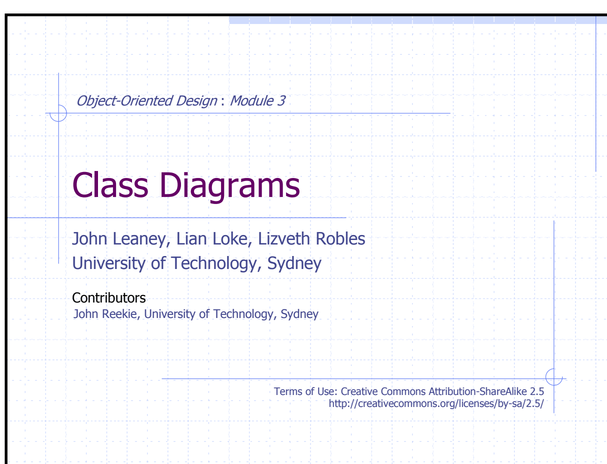
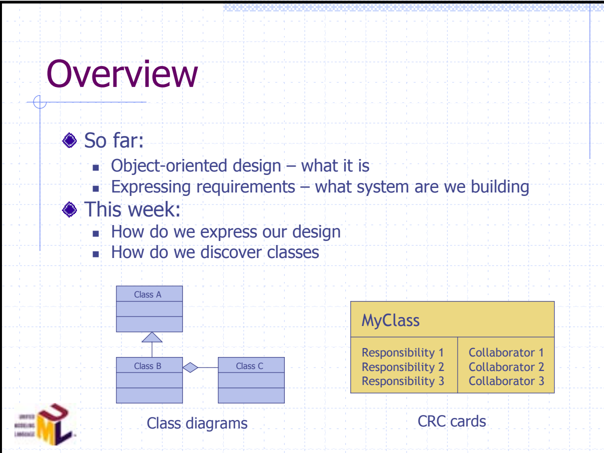
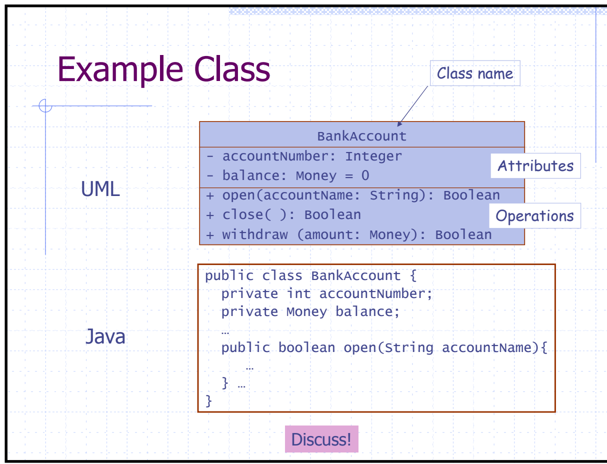




Class Diagram

- UML structural diagram for presenting the **static model** of a system
  - *Use cases* express a dynamic relationship (this, then that ..)
- The static model is first constructed in the requirements analysis phase, (for example with CRC cards) and greater detail is added in design, and as the design progresses.
- Typically shows:
  - Classes with attributes, operations, invariants.
  - Relationships between classes



More info

- Class names are capitalized
- Attributes and operations have visibility, type, name
- Operations represent:
  - common behaviours of all instances of a class
  - actions that can be carried out by, or on, an object
  - services offered by a class
  - specification for some aspect of behaviour of a class
- For example:
  - change an instance's data (i.e. attributes)
  - respond to simple queries
  - perform calculations
  - send messages or events to other objects

Visibility

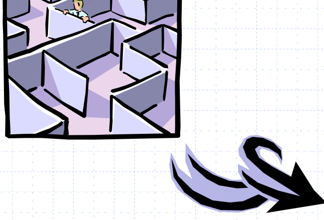
- Public (+)
  - member of class is directly accessible to other objects; part of public interface
- Private (-)
  - member is hidden from public use; only accessible to other members within the class
- Protected (#)
  - special visibility scope applicable to inheritance only

My advice: Unless you have a good reason to do otherwise, make all operations public, and all attributes private

Java vs UML

- You will note that the syntax is different
  - UML is a design language. It does not depend on any particular implementation language
  - Java is an implementation language. It has nothing whatsoever to do with UML
  - You have to know both!
- Assumed or implicit operations may not be explicitly shown on class diagram:
  - create instances
  - get and set methods
  - destroy object
- Java code naturally "grows" data members and operations that aren't in the UML
  - This is the old problem of keeping design specifications in sync with implementations

Solving a complex problem



Don't get stuck!
Start somewhere,
and then iterate…

How do we start finding classes?

- CRC Cards:
  - An informal approach to OO modelling.
  - Created through scenarios, based on the system requirements, that model the behaviour of the system.
- CRC stands for
  - Class
    - A set of objects that share common structure and behaviour
  - Responsibility
    - Some behaviour for which an object is held accountable
  - Collaboration
    - Process whereby several objects cooperate to provide some higher-level behaviour

CRC Cards

Class Name:	
Responsibilities	Collaborations
Responsibilities of a class are listed in this section.	*Collaborations with other classes are listed here, together with a brief description of the purpose of the collaboration.*

Class Name	*Client*
Responsibilities	Collaborations
Provide client information.	
Provide list of campaigns.	*Campaign provides campaign details.*

Class Name	*Campaign*
Responsibilities	Collaborations
Provide campaign information.	
Provide list of adverts.	*Advert provides advert details.*
Add a new advert.	*Advert constructs new object.*

Class Name	*Advert*
Responsibilities	Collaborations
Provide advert details.	
Construct adverts.	

Types of Responsibilities

- Knowing
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate
- Doing
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- Examples
  - a *Sale* is responsible for knowing its total (POS example)
  - a *Sale* is responsible for creating *SalesLineItems*

A CRC Session

- The participants are users (ideally…) and designers.
- A designer runs the session, records the data and resolves any disputes.
- The users contribute most of the ideas, as they are the people who know the current system.
- As classes, responsibilities and collaborators are discovered, they are written down, one class per card.

Running a CRC Session (1)

1. Brainstorm the classes.
   Ask the users to suggest possible classes.
   At this point, the key is to get people involved so no decision is made about whether a class is "right" or not; they are possible classes.
2. Filter the classes.
   - eliminate redundant classes
   - identify missing classes
   - recognise related classes
3. Define the data in each class.
   - All participants should have the same meaning
     - for the words used in the session. The definition can be a simple phrase, or attributes can be added to the class. This is usually done on the back of the card.

Running a CRC Session (2)

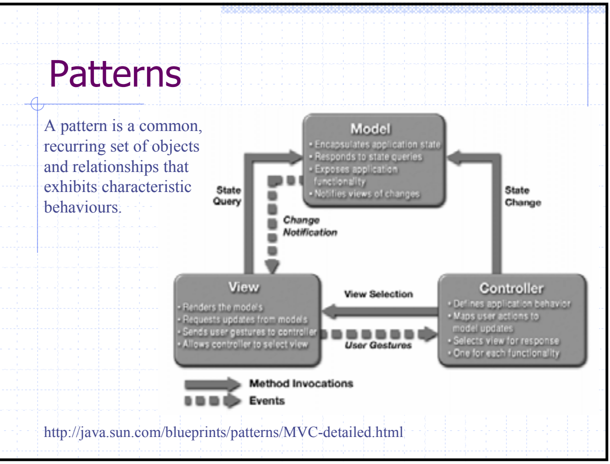
4. Assign classes to participants.
5. Run the scenarios.
   - Step through each scenario.
   - For each event or action, decide which class should have the responsibility to fulfil it.
   - Write down the name of the action under the Responsibilities column for the chosen card.
   - Decide which classes collaborate to fulfil this responsibility, and write down their names in the Collaborators column.

Other techniques

- Users, experts, commonsense, imagination
- Noun phrase identification
  - Extract the nouns from a written description of the vision for the system (or from similar kinds of texts, eg. brief, system narrative, etc.)
  - Classes are named by nouns
- Stereotype classes
- Analysis patterns

Stereotype classes

- Classes can have stereotypical, or often found, roles.
- 3 main stereotypes in UML
  - Entity
    - representative of key entities or things in the application domain
  - Boundary
    - user or external interfaces
  - Control
    - coordinate or delegate to other objects
- Other:
  - Collection classes
    - provide storage and management of a set of objects of the one type.
    - e.g. array, linked list, vector

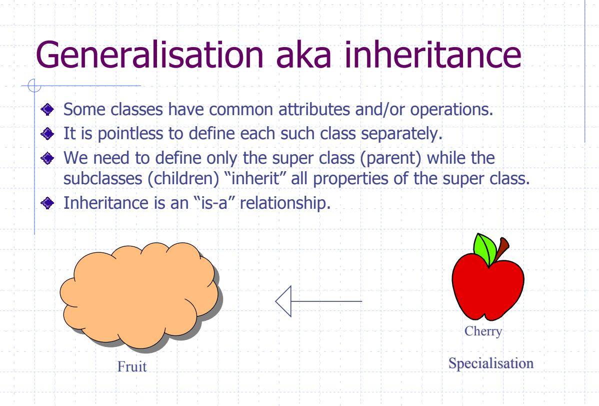


Reasonability Checks for Candidate Classes

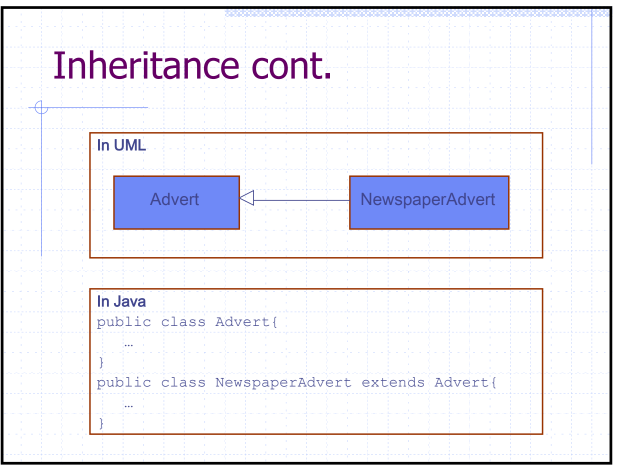
- A number of tests help to check whether a candidate class is reasonable
  - Is it beyond the scope of the system?
  - Does it refer to the system as a whole?
  - Does it duplicate another class?
  - Is it too vague?
  - Is it too tied up with physical inputs and outputs?
  - Is it really an attribute?
  - Is it really an operation?
  - Is it really an association?
- If any answer is 'Yes', consider modelling the potential class in some other way (or do not model it at all)

Generalisation aka inheritance

- Some classes have common attributes and/or operations.
- It is pointless to define each such class separately.
- We need to define only the super class (parent) while the subclasses (children) "inherit" all properties of the super class.
- Inheritance is an "is-a" relationship.



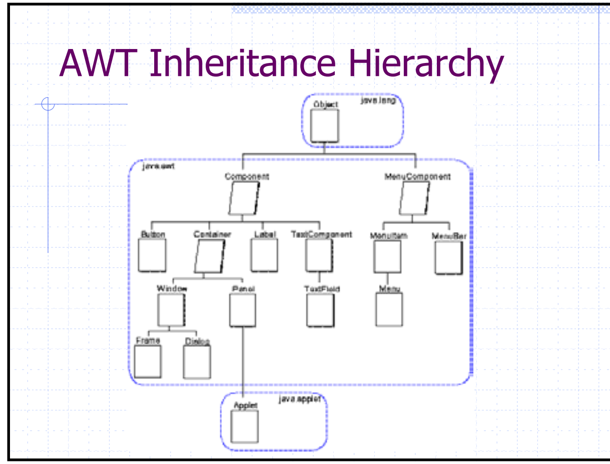
Inheritance cont.

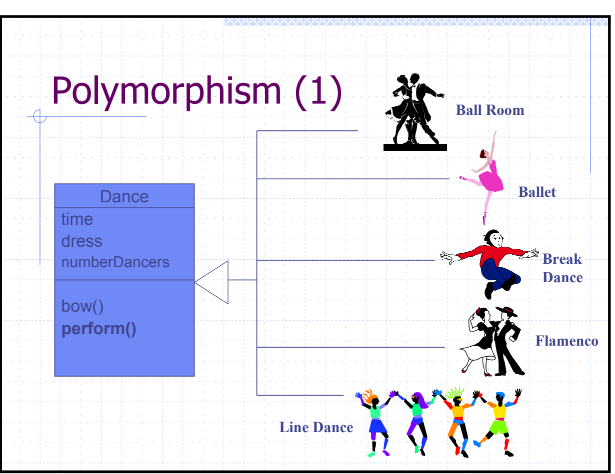



```
public class Advert{
    …
}
public class NewspaperAdvert extends Advert{
    …
}
```

When to use inheritance?

- Ask the question: *Is A a B?*
- Designing a more *specialised* class from an existing class.
- Capturing common information in a super (base) class
  - and specialised functionality in sub (derived) classes.
- Inheritance leads to creation of a *class hierarchy*
  - The AWT (Abstract Windowing Toolkit) in Java has an inheritance hierarchy of many levels
  - At the top of the hierarchy of every class in Java is the **Object** class.





Polymorphism (2)

- Each child class has the operation "perform".
- *Perform* has a different meaning for each class.
  - The dance is performed in a completely different way.
- Each subclass must define its own *perform* operation.
- It should be possible to use any of the subclasses where the base class can be used (the Liskov substitution principle)

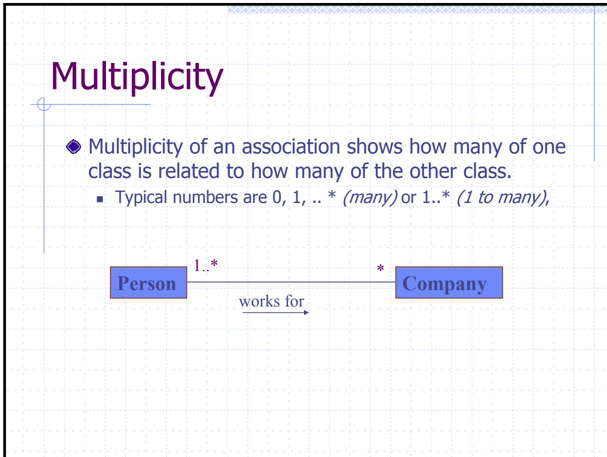
poly = many, morph = shape

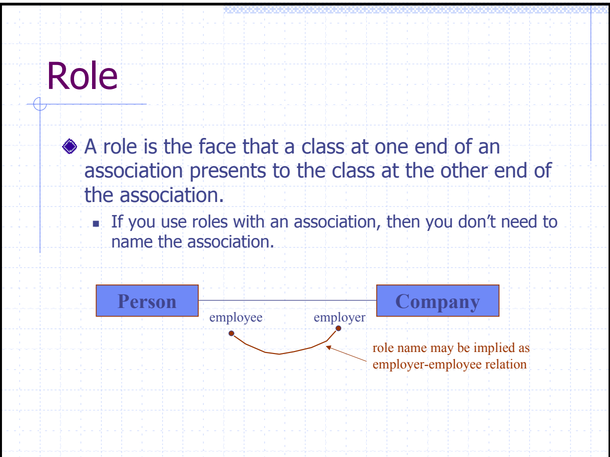
Pros and Cons

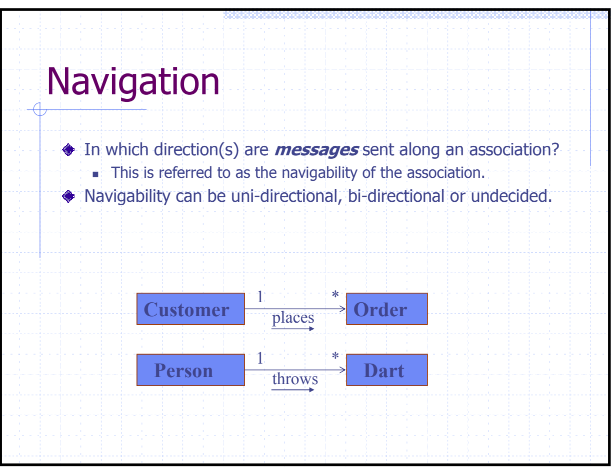
- Pros
  - Conceptually straight-forward translation from modeling to implementation ("is-a" -> "extends")
  - Can be used to improve reusability
  - Can be used to improve extensibility (the open-closed principle)
- Cons
  - Excessively deep hierarchies become brittle and hard to understand
  - There is on-going controversy on whether implementation inheritance is good (or whether we should just have interface inheritance)
- Regardless, inheritance is firmly established in object-oriented analysis, design, and implementation. You **must** understand and be able to use it well.

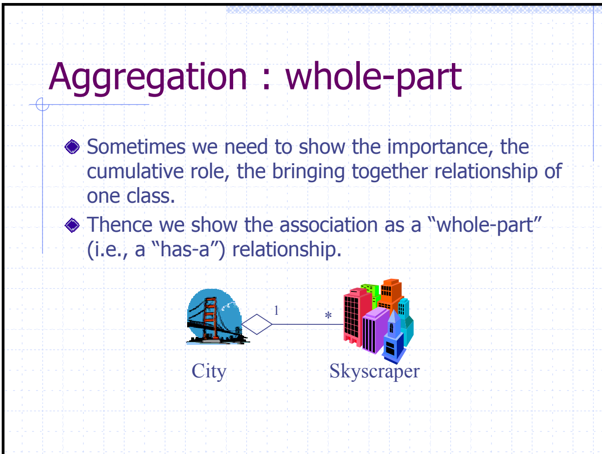
Association

- This relationship specifies the *structural* relationship between two classes (it is a "*needs-a*" relationship).
- Both classes are conceptually at the same level (i.e., none is more important than the other).
- An association can be named, for clarity.

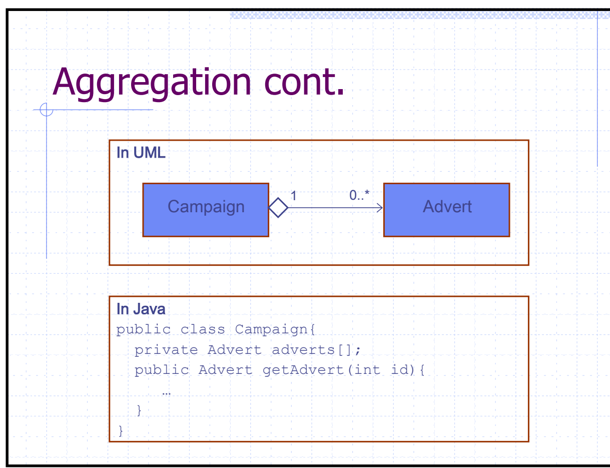

Multiplicity
Multiplicity of an association shows how many of one class is related to how many of the other class.
Typical numbers are 0, 1, .. * (many) or 1..* (1 to many),
Person
1..*
*
Company
works for


Role
A role is the face that a class at one end of an association presents to the class at the other end of the association.
If you use roles with an association, then you don't need to name the association.
Person
Company
employee
employer
role name may be implied as
employer-employee relation

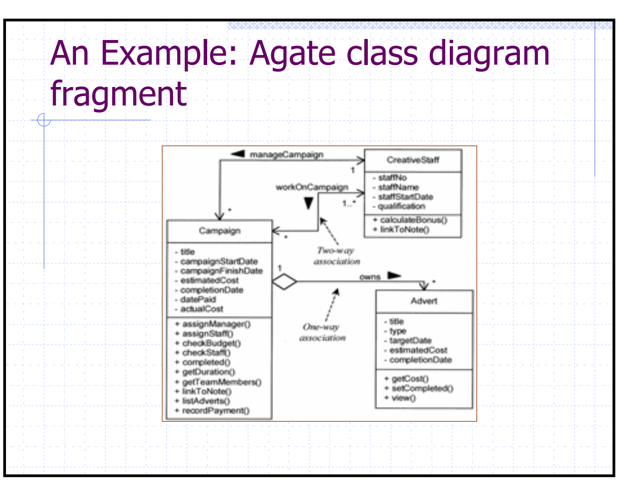

Navigation
In which direction(s) are messages sent along an association?
This is referred to as the navigability of the association.
Navigability can be uni-directional, bi-directional or undecided.
Customer
1
*
Order
places
Person
1
*
Dart
throws



Aggregation : whole-part
Sometimes we need to show the importance, the cumulative role, the bringing together relationship of one class.
Thence we show the association as a "whole-part" (i.e., a "has-a") relationship.
1
*
City
Skyscraper



Aggregation cont.
In UML
Campaign
1
0..*
Advert
In Java
public class Campaign{
private Advert adverts[];
public Advert getAdvert(int id){
…
}
}



An Example: Agate class diagram fragment
manageCampaign
CreativeStaff
1
- staffNo
- staffName
- staffStartDate
- qualification
+ calculateBonus()
+ linkToNote()
workOnCampaign
1..*
*
*
Campaign
- title
- campaignStartDate
- campaignFinishDate
- estimatedCost
- completionDate
- datePaid
- actualCost
+ assignManager()
+ assignStaff()
+ checkBudget()
+ checkStaff()
+ completed()
+ getDuration()
+ getTeamMembers()
+ linkToNote()
+ listAdverts()
+ recordPayment()
Two-way association
1
owns
*
One-way association
Advert
- title
- type
- targetDate
- estimatedCost
- completionDate
+ getCost()
+ setCompleted()
+ view()

What we haven't done…

- Dependency
  - An instance of class A *depends* on an instance of class B
- Interfaces
  - A class *realizes* an interface
- So: **read** (at least) the references here:
  http://www.softwarepractice.org/courseware/index.php/design/class_diagrams
- And **practice** creating your class diagrams!
  - Start with CRC cards to identify objects
  - Identify attributes and operations
  - Identify inheritance relationships
  - Identify associations
  - As you get more used to the notation, you can start making your diagrams more sophisticated and *precise*.