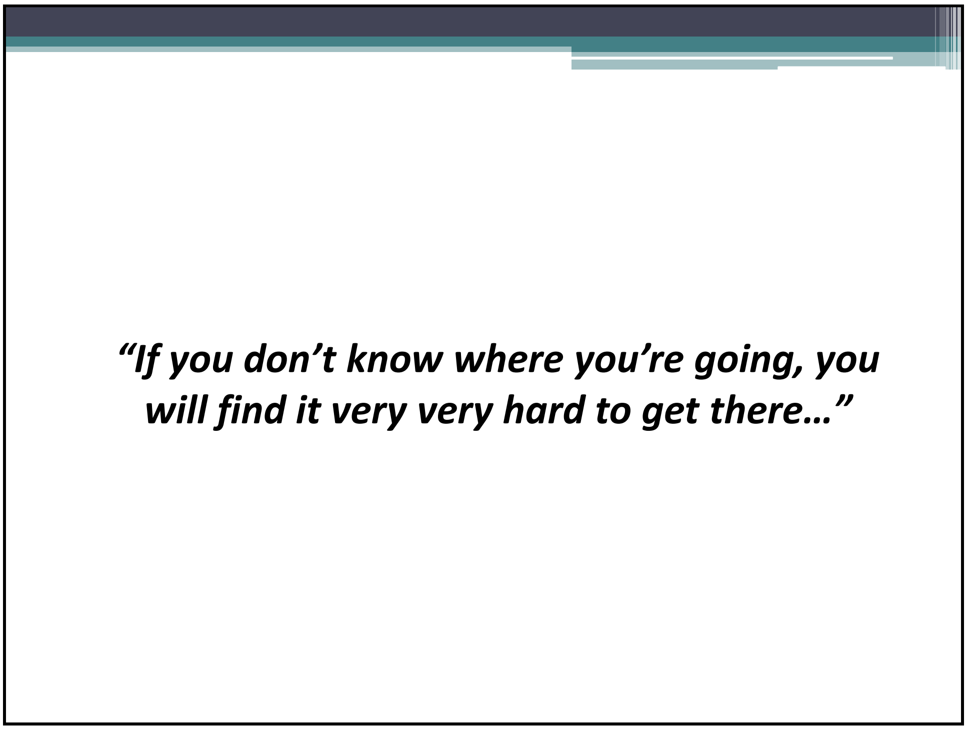
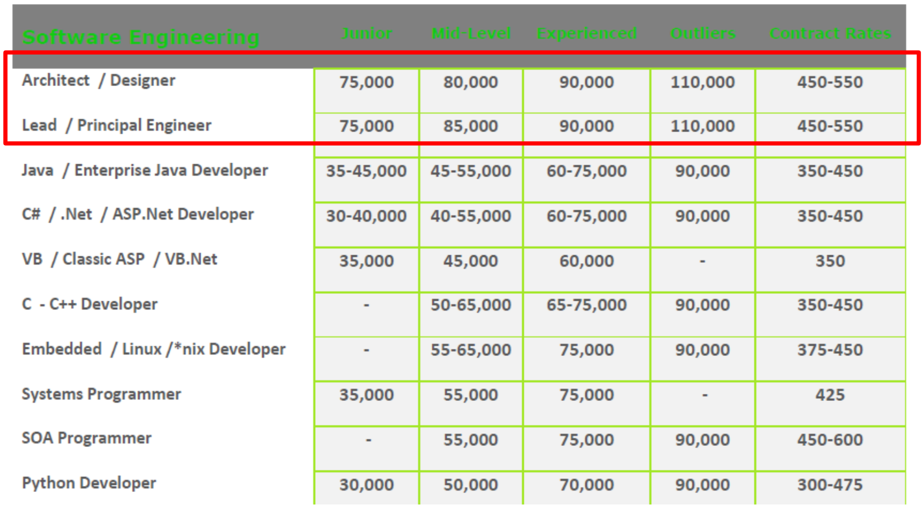
Software Engineering III

Ciaran Cawley
KE-G-026

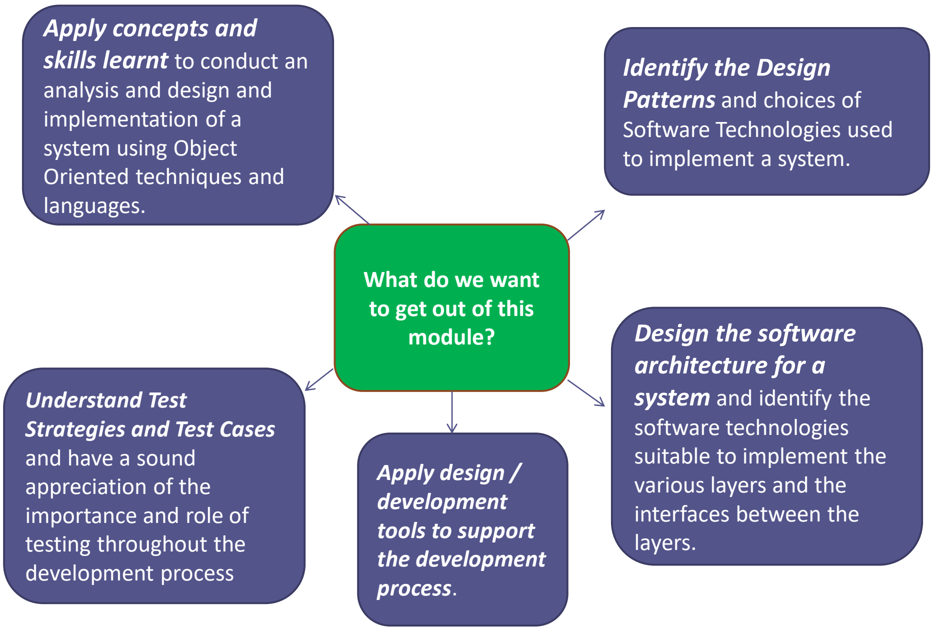
ciaran.cawley@dit.ie



"If you don't know where you're going, you will find it very very hard to get there…"



Software Engineering	Junior	Mid-Level	Experienced	Outliers	Contract Rates
Architect / Designer	75,000	80,000	90,000	110,000	450-550
Lead / Principal Engineer	75,000	85,000	90,000	110,000	450-550
Java / Enterprise Java Developer	35-45,000	45-55,000	60-75,000	90,000	350-450
C# / .Net / ASP.Net Developer	30-40,000	40-55,000	60-75,000	90,000	350-450
VB / Classic ASP / VB.Net	35,000	45,000	60,000	-	350
C - C++ Developer	-	50-65,000	65-75,000	90,000	350-450
Embedded / Linux /*nix Developer	-	55-65,000	75,000	90,000	375-450
Systems Programmer	35,000	55,000	75,000	-	425
SOA Programmer	-	55,000	75,000	90,000	450-600
Python Developer	30,000	50,000	70,000	90,000	300-475



What do we want to get out of this module?

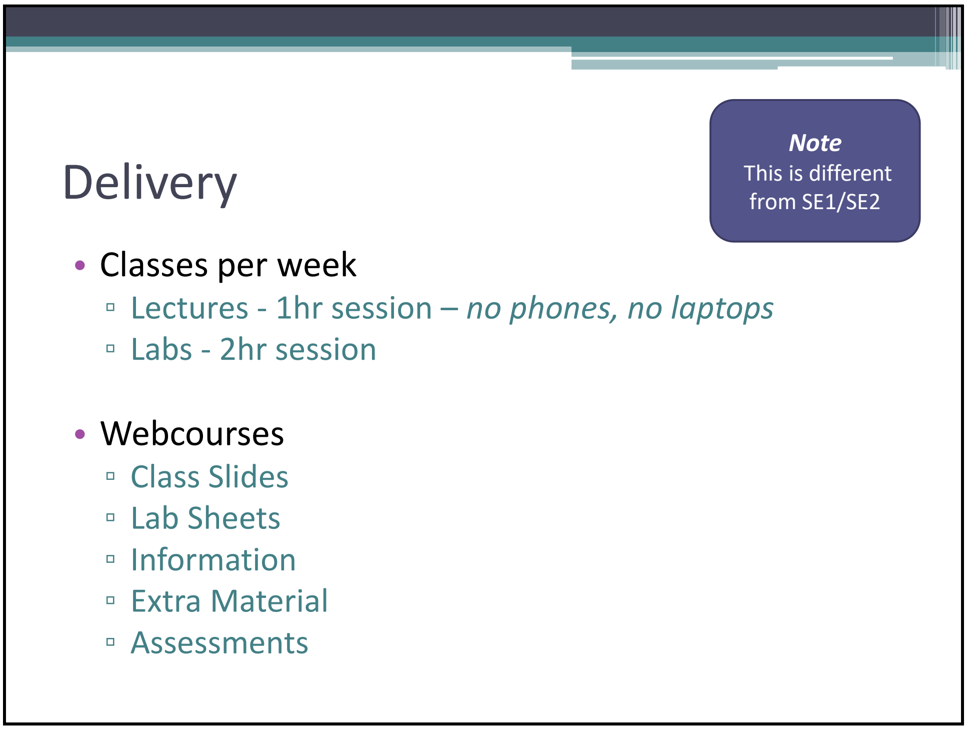
Apply concepts and skills learnt to conduct an analysis and design and implementation of a system using Object Oriented techniques and languages.

Identify the Design Patterns and choices of Software Technologies used to implement a system.

Design the software architecture for a system and identify the software technologies suitable to implement the various layers and the interfaces between the layers.

Apply design / development tools to support the development process.

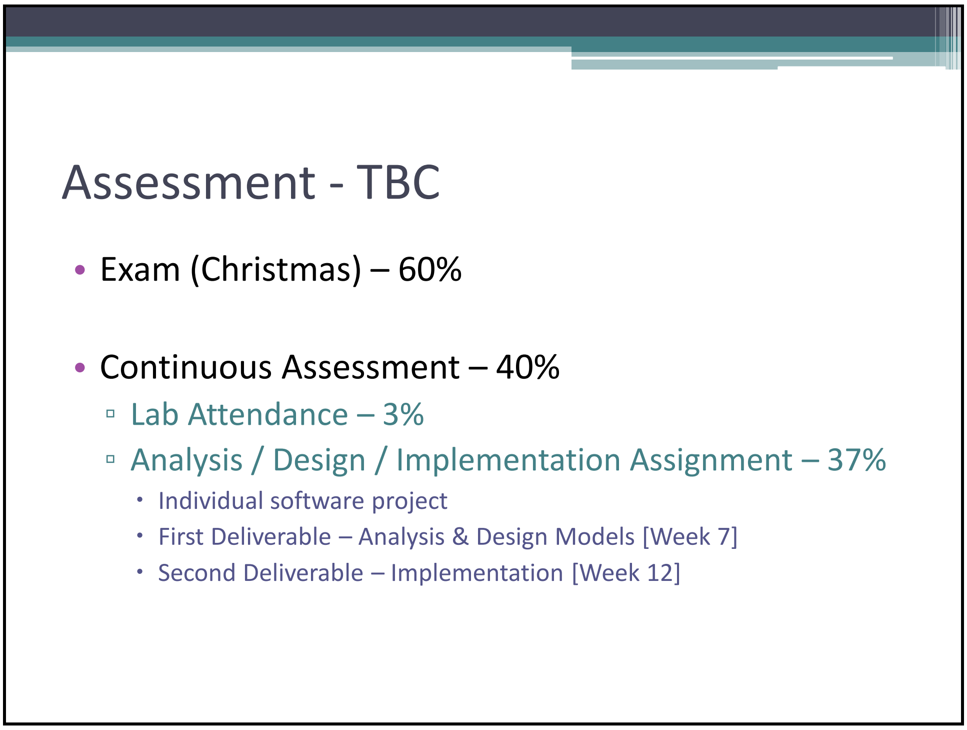
Understand Test Strategies and Test Cases and have a sound appreciation of the importance and role of testing throughout the development process



Delivery

Note
This is different from SE1/SE2

- Classes per week
  - Lectures - 1hr session – *no phones, no laptops*
  - Labs - 2hr session
- Webcourses
  - Class Slides
  - Lab Sheets
  - Information
  - Extra Material
  - Assessments



Assessment - TBC

- Exam (Christmas) – 60%
- Continuous Assessment – 40%
  - Lab Attendance – 3%
  - Analysis / Design / Implementation Assignment – 37%
    - Individual software project
    - First Deliverable – Analysis & Design Models [Week 7]
    - Second Deliverable – Implementation [Week 12]

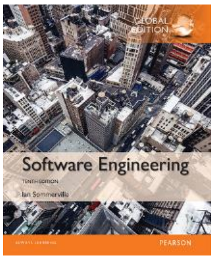
Books

Ian Sommerville, 2016, Software Engineering, 10th Edition, Pearson.

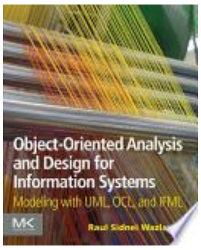
S. Bennett, S. McRobb and R. Farmer, 2010, Object-Oriented Systems Analysis and Design using UML, 4th Ed , McGraw-Hill.

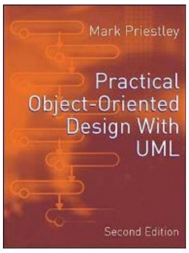
Wazlawick, 2014, Object-Oriented Analysis and Design for Information Systems, 1st Edition Modeling with UML, OCL, and IFML.

Mark Priestley, 2003, Practical Object-Oriented Design with UML, McGraw Hill.



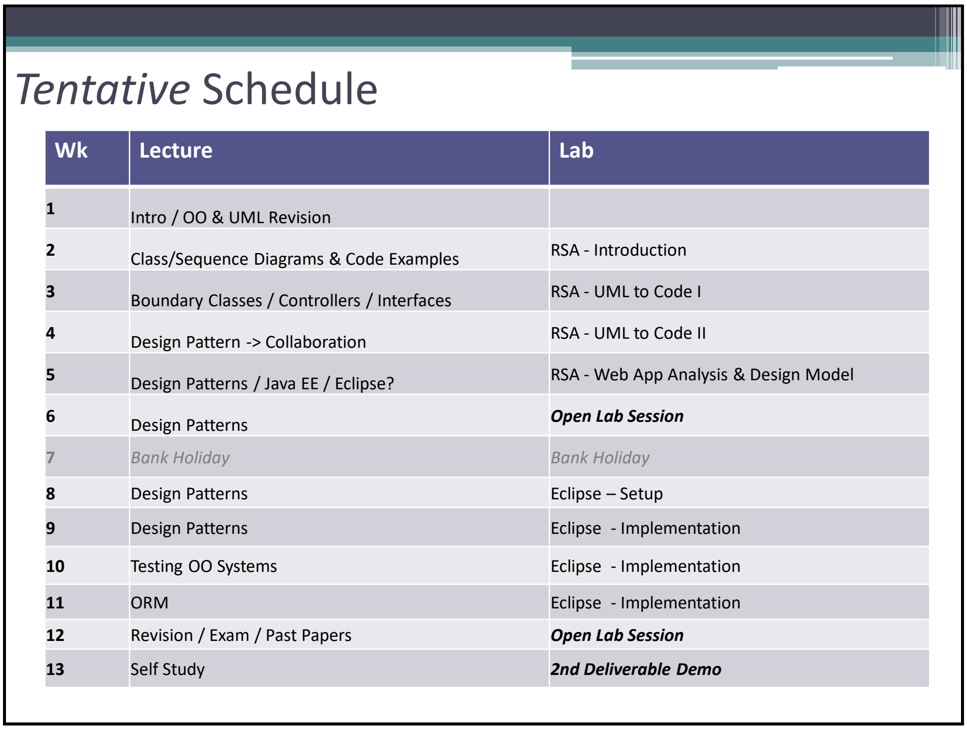
Object-Oriented Systems Analysis and Design Using UML
Simon Bennett, Steve McRobb and Ray Farmer


Object-Oriented Analysis and Design for Information Systems
Modeling with UML, OCL, and IFML
MK
Raul Sidnei Wazlawick



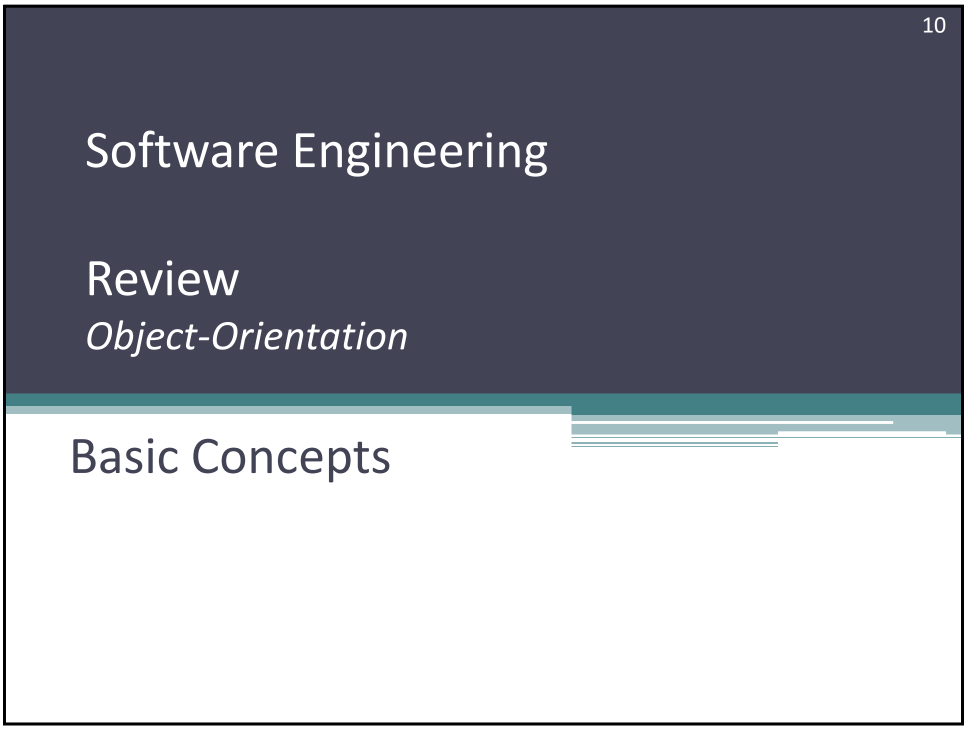
Development Environment

- **Tools**
  - **R**ational **S**oftware **A**rchitect (RSA) [Designer]
    - On lab machines (look for IBM Software Delivery Platform in *Programs)*
    - ***Optional – If you want to use RSAD on your own machine***
      - Commercial Software but *Academic* licence available (fully functional)
      - Link to the software on webcourses
      - I will provide the key (activation kit) on webcourses also
  - **Eclipse IDE for *Java EE* Developers**
- **Back up your work…**



Tentative Schedule

Wk	Lecture	Lab
1	Intro / OO & UML Revision	
2	Class/Sequence Diagrams & Code Examples	RSA - Introduction
3	Boundary Classes / Controllers / Interfaces	RSA - UML to Code I
4	Design Pattern -> Collaboration	RSA - UML to Code II
5	Design Patterns / Java EE / Eclipse?	RSA - Web App Analysis & Design Model
6	Design Patterns	***Open Lab Session***
7	*Bank Holiday*	*Bank Holiday*
8	Design Patterns	Eclipse – Setup
9	Design Patterns	Eclipse - Implementation
10	Testing OO Systems	Eclipse - Implementation
11	ORM	Eclipse - Implementation
12	Revision / Exam / Past Papers	***Open Lab Session***
13	Self Study	***2nd Deliverable Demo***



Software Engineering

Review

Object-Orientation

Basic Concepts

Object Oriented Programming

- A software development paradigm
- A methodology (way) of writing computer programmes
- Others approaches / ways are

▫ Procedural	e.g. C
▫ Functional	e.g. Haskell
▫ Logical	e.g. Prolog

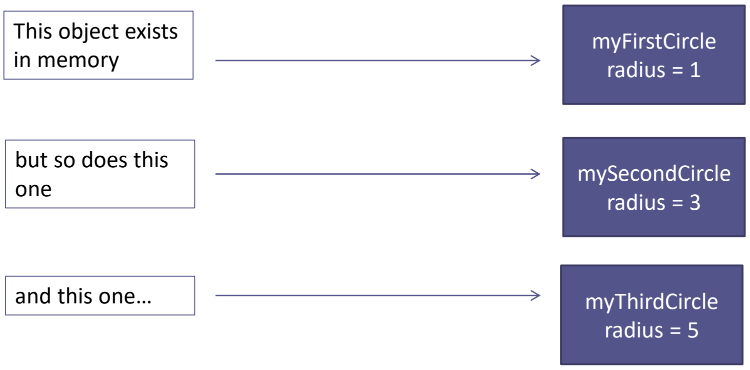
Objects

- Definition: an ***abstraction*** of something in the problem domain, reflecting the capabilities of the system to keep information about it, interact with it, or both.
- Abstraction
  - A form of representation that includes only what is important or interesting from a particular viewpoint, hiding the features deemed irrelevant for the current study
  - Example: a map, engineering drawing, analyst representation of data...

Objects

- Purpose of Objects
  - They promote understanding of the real world
  - Provide a practical basis for computer implementation.
- Characteristics of Objects
  - State
    - Particular condition that an object may be in at a moment in its life
  - Behaviour
    - What an object can do, how it can respond to events and stimuli
    - Things that an object can do (methods) that are relevant to the view or abstraction of interest.
  - Identity
    - Every object is unique

From Objects to Classes



This object exists in memory

myFirstCircle
radius = 1

but so does this one

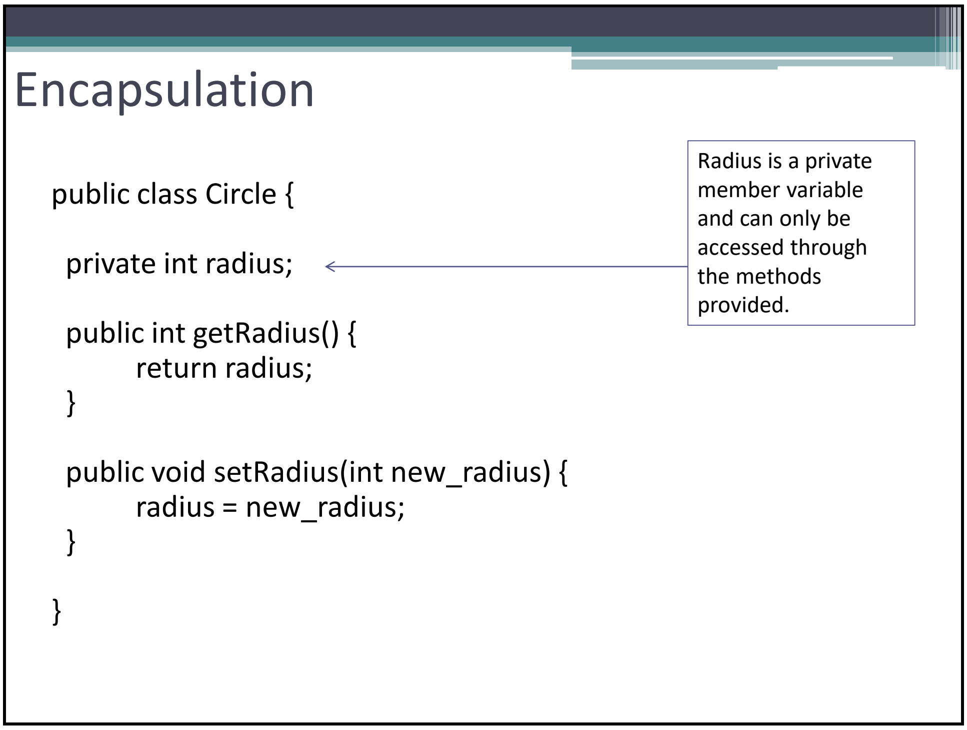
mySecondCircle
radius = 3

and this one…

myThirdCircle
radius = 5

These objects are all circles…
And so we say their ***type*** or ***class*** is ***Circle***

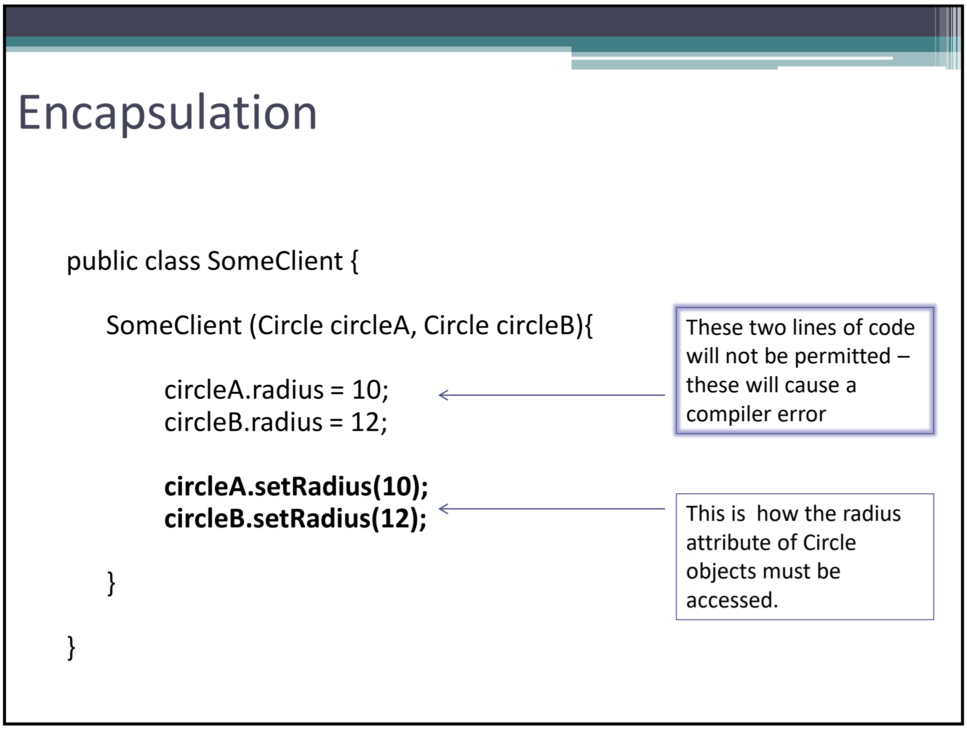
Encapsulation, Inheritance and Polymorphism



Encapsulation

```
public class Circle {

  private int radius;

  public int getRadius() {
          return radius;
  }

  public void setRadius(int new_radius) {
          radius = new_radius;
  }

}
```

Radius is a private member variable and can only be accessed through the methods provided.

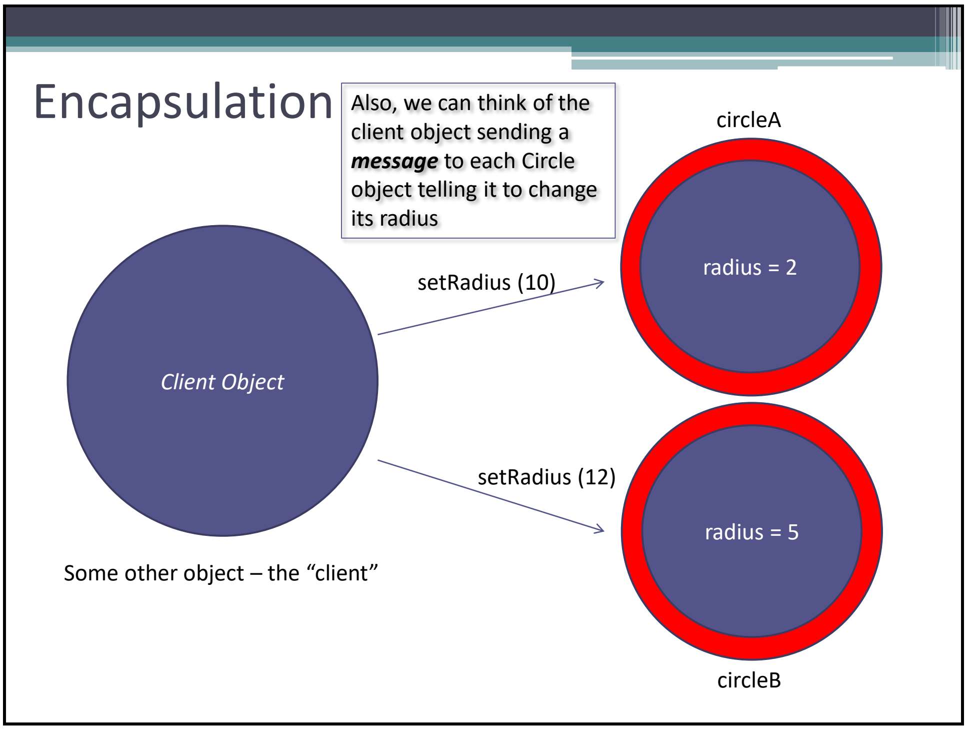


Encapsulation

```
public class SomeClient {

    SomeClient (Circle circleA, Circle circleB){

        circleA.radius = 10;
        circleB.radius = 12;

        circleA.setRadius(10);
        circleB.setRadius(12);

    }

}
```

These two lines of code will not be permitted – these will cause a compiler error

This is how the radius attribute of Circle objects must be accessed.



Encapsulation

Also, we can think of the client object sending a ***message*** to each Circle object telling it to change its radius

Interfaces

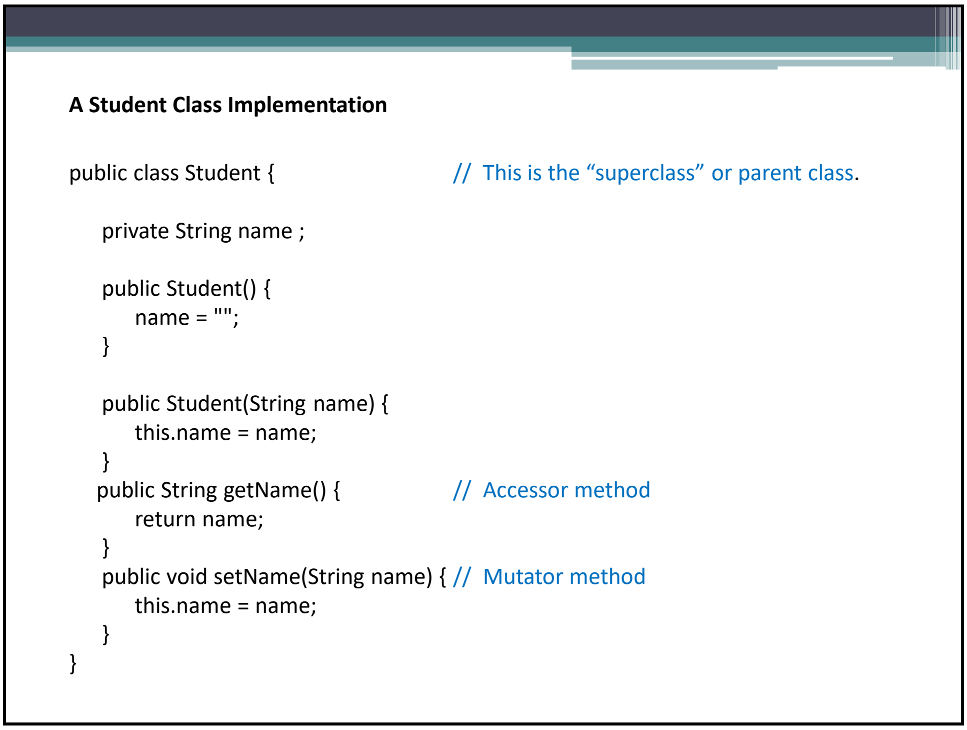
```
public class Circle {

  private int radius;

  public int getRadius() {
         return radius;
  }

  public void setRadius(int new_radius) {
         radius = new_radius;
  }

}
```

Note the two public methods – if these were private or did not exist then there would be no way for an other object to access a Circle’s *radius* attribute.

The *public* methods here define how *other objects can access a Circle object* – in other words they define a Circle’s ***interface.***

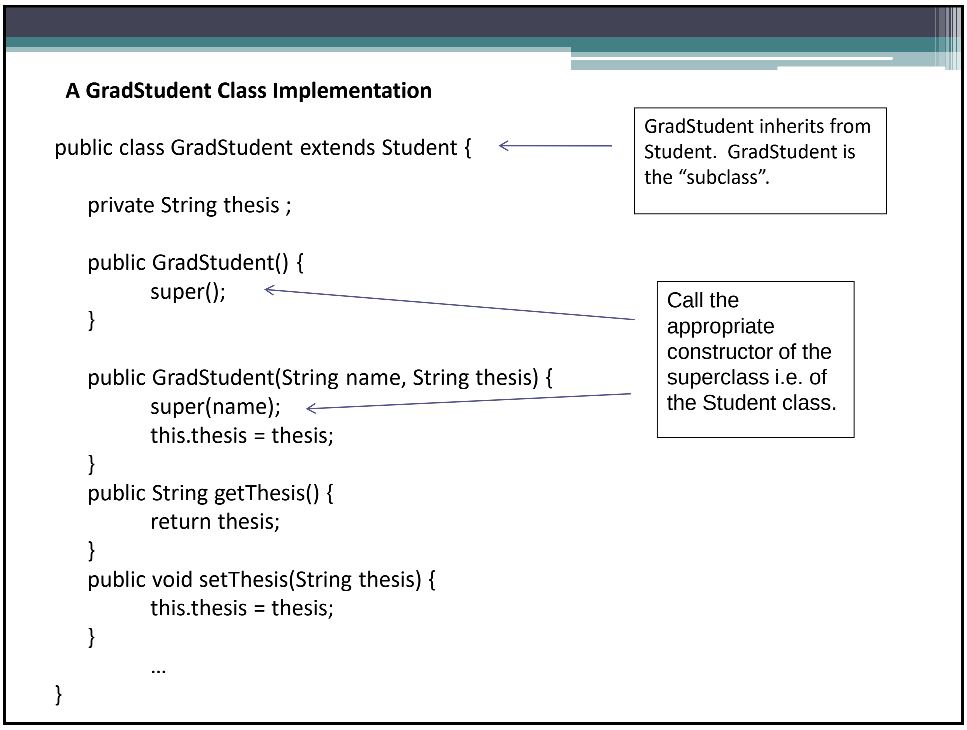
Inheritance

- A class (the ***subclass***) can extend another class (the ***superclass***)
- The subclass ***inherits*** the methods of the superclass
- The subclass may ***override*** a method in the superclass by ***implementing its own version of the method***. The new version may call the method of the superclass



A Student Class Implementation

```
public class Student {                    //  This is the “superclass” or parent class.

     private String name ;

     public Student() {
          name = "";
     }

     public Student(String name) {
          this.name = name;
     }
    public String getName() {             //  Accessor method
          return name;
     }
     public void setName(String name) { //  Mutator method
          this.name = name;
     }
}
```

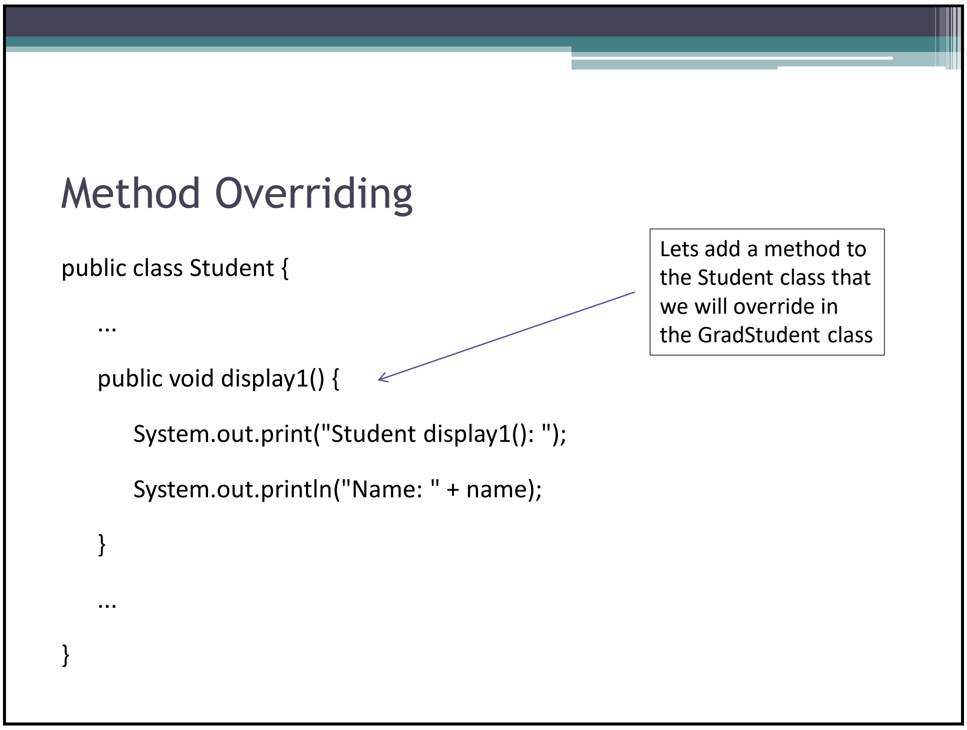


A GradStudent Class Implementation

```
public class GradStudent extends Student {

     private String thesis ;

     public GradStudent() {
               super();
     }

     public GradStudent(String name, String thesis) {
               super(name);
               this.thesis = thesis;
     }
     public String getThesis() {
               return thesis;
     }
     public void setThesis(String thesis) {
               this.thesis = thesis;
     }
               …
}
```

GradStudent inherits from Student. GradStudent is the “subclass”.

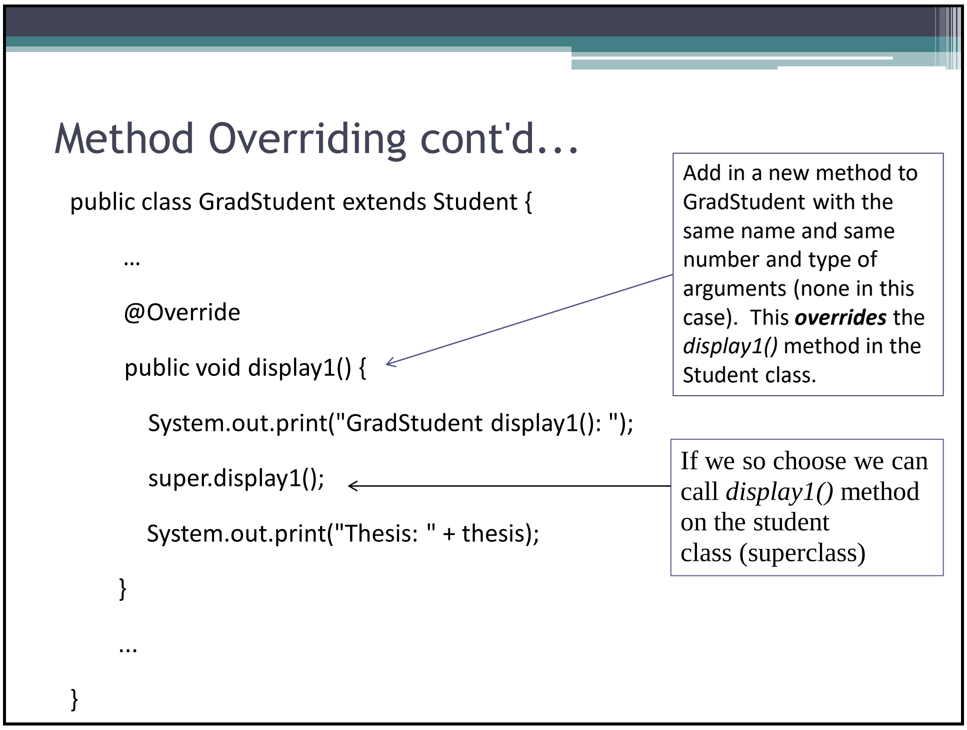
Call the appropriate constructor of the superclass i.e. of the Student class.



Method Overriding

```
public class Student {
    ...
    public void display1() {
        System.out.print("Student display1(): ");
        System.out.println("Name: " + name);
    }
    ...
}
```

Lets add a method to the Student class that we will override in the GradStudent class



Method Overriding cont'd...

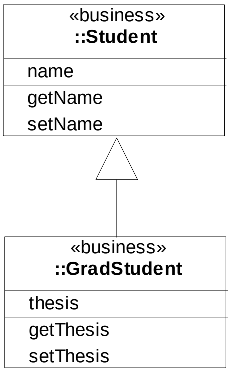
```
public class GradStudent extends Student {
    ...
    @Override
    public void display1() {
      System.out.print("GradStudent display1(): ");
      super.display1();
      System.out.print("Thesis: " + thesis);
    }
    ...
}
```

Add in a new method to GradStudent with the same name and same number and type of arguments (none in this case). This ***overrides*** the *display1()* method in the Student class.

If we so choose we can call *display1()* method on the student class (superclass)

Modelling the Class Hierarchy for *Student* and *GradStudent*

Class Diagram



«business»
::Student

name

getName
setName

«business»
::GradStudent

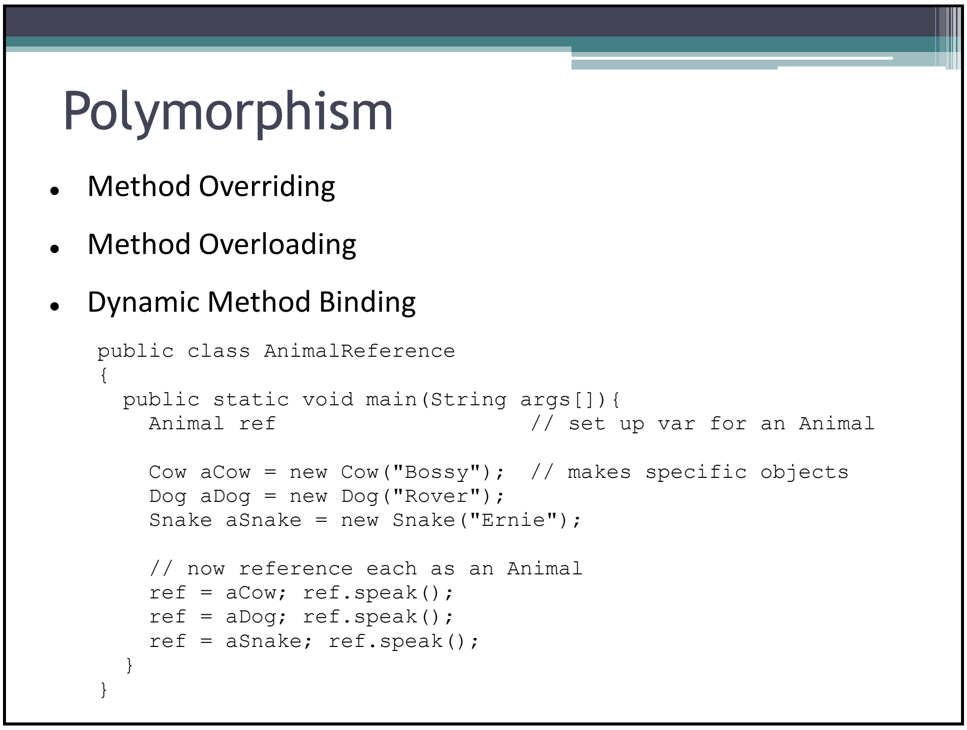
thesis

getThesis
setThesis

The diagram may contain additional information, e.g. the types of attributes, the signatures of methods, etc. Simple methods such as *get* and *set* methods are not usually shown on the diagram. A class diagram is intended to give a clear picture of the classes in a system and the relationships between those

The “is a” relationship

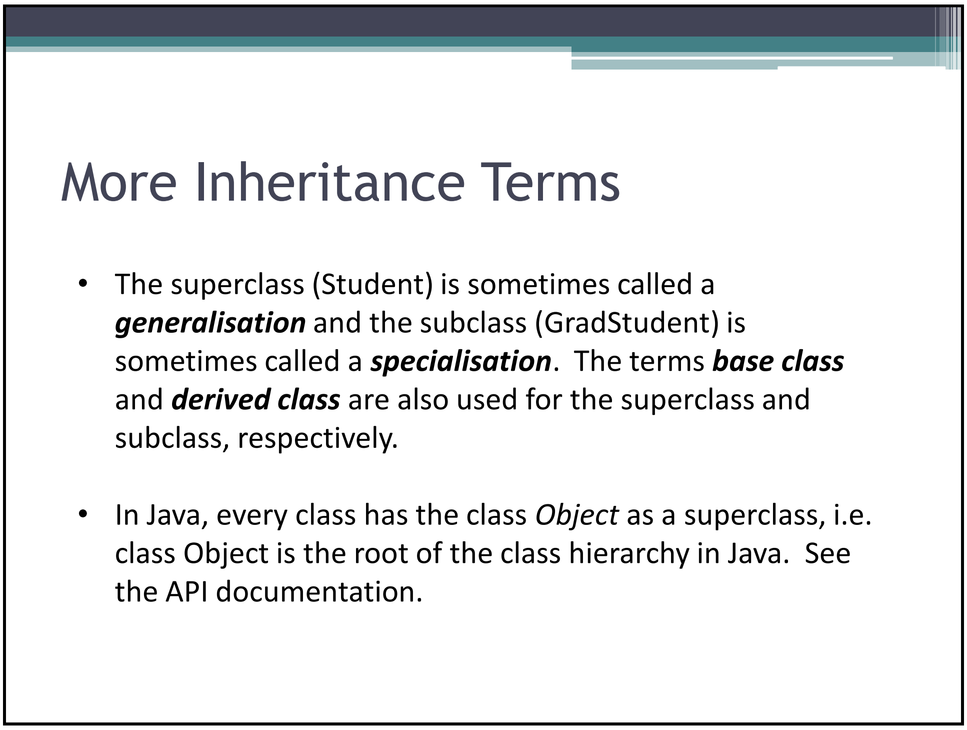
- GradStudent ***extends*** the Student class
- We can say that a GradStudent ***“is a”*** Student
- This means we can refer to GradStudent objects as Student objects
- **This has important implications**



Polymorphism

- Method Overriding
- Method Overloading
- Dynamic Method Binding

```
public class AnimalReference
{
  public static void main(String args[]){
    Animal ref                    // set up var for an Animal

    Cow aCow = new Cow("Bossy");  // makes specific objects
    Dog aDog = new Dog("Rover");
    Snake aSnake = new Snake("Ernie");

    // now reference each as an Animal
    ref = aCow; ref.speak();
    ref = aDog; ref.speak();
    ref = aSnake; ref.speak();
  }
}
```



More Inheritance Terms

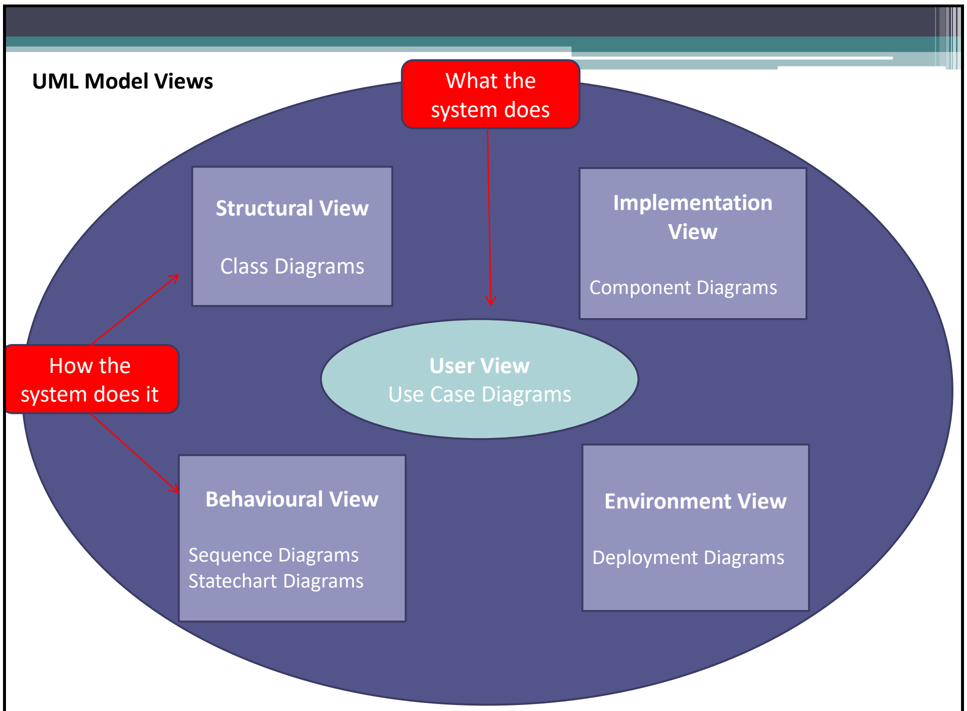
- The superclass (Student) is sometimes called a ***generalisation*** and the subclass (GradStudent) is sometimes called a ***specialisation***. The terms ***base class*** and ***derived class*** are also used for the superclass and subclass, respectively.

- In Java, every class has the class *Object* as a superclass, i.e. class Object is the root of the class hierarchy in Java. See the API documentation.

Review

UML

UML Model Views



What the system does

Structural View

Class Diagrams

Implementation View

Component Diagrams

How the system does it

User View
Use Case Diagrams

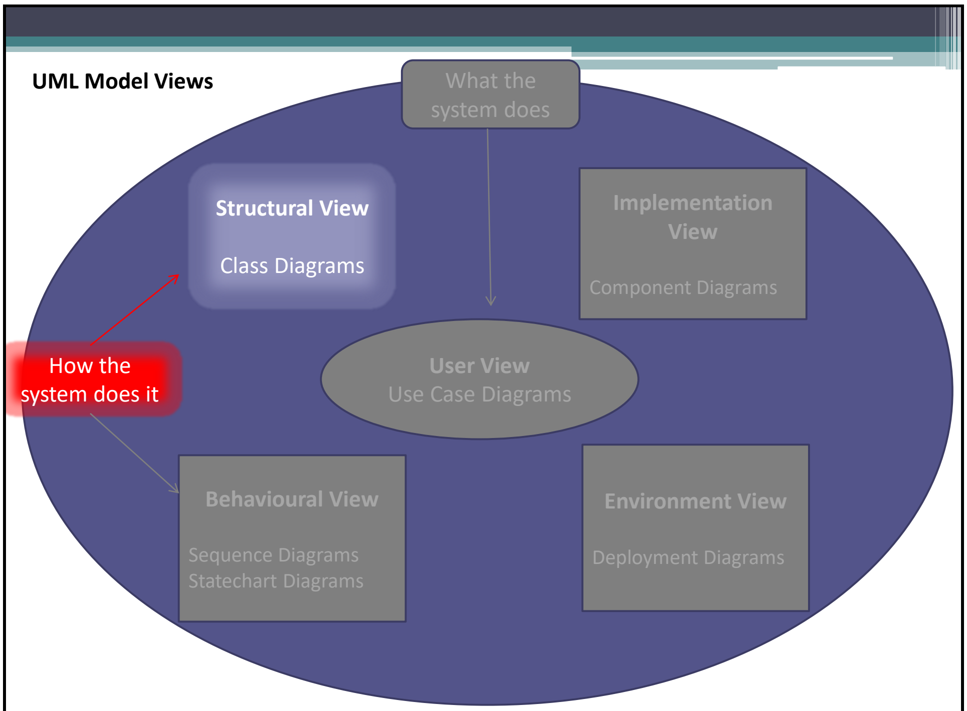
Behavioural View

Sequence Diagrams
Statechart Diagrams

Environment View

Deployment Diagrams

UML Model Views



What the system does

Structural View

Class Diagrams

Implementation View

Component Diagrams

How the system does it

User View

Use Case Diagrams

Behavioural View

Sequence Diagrams
Statechart Diagrams

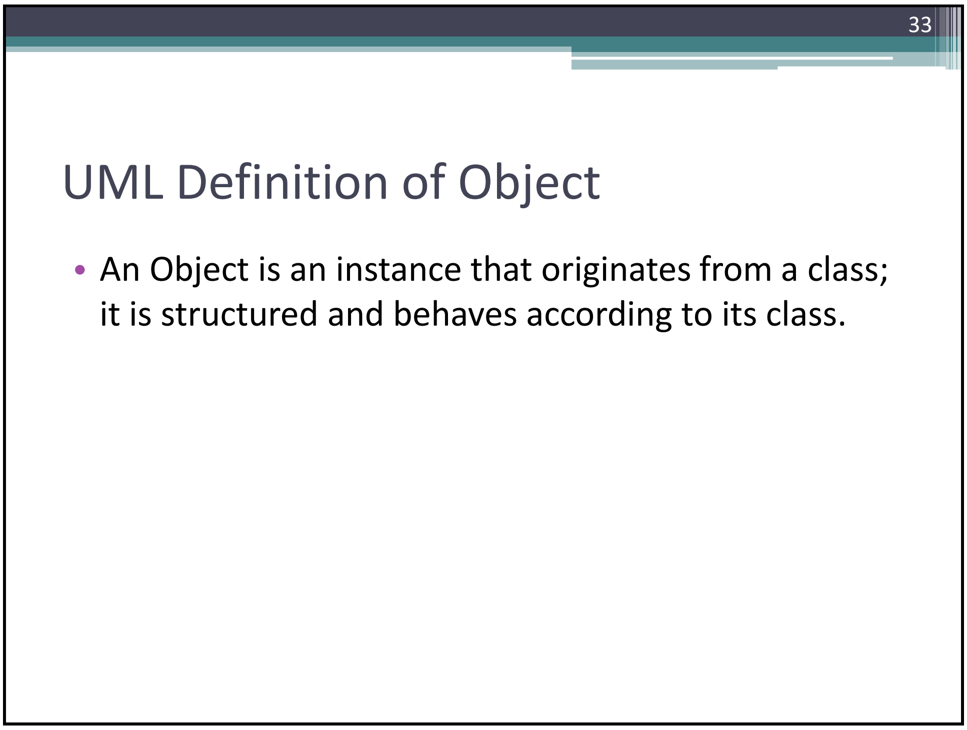
Environment View

Deployment Diagrams

UML Definition of Class

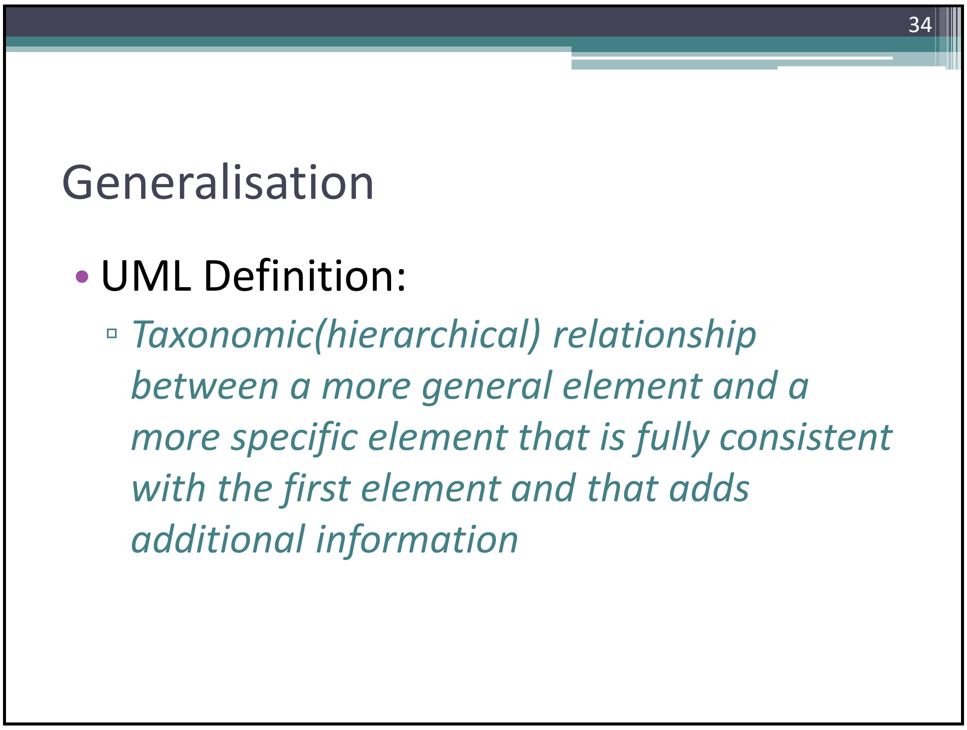
- A Class is a description of a set of objects that share attribute, operations (behaviours), ***relationship with other objects***.

- The purpose of a class is to declare a collection of operations, and attributes that fully describe the structure and behaviour for all objects of that class.



UML Definition of Object

- An Object is an instance that originates from a class; it is structured and behaves according to its class.

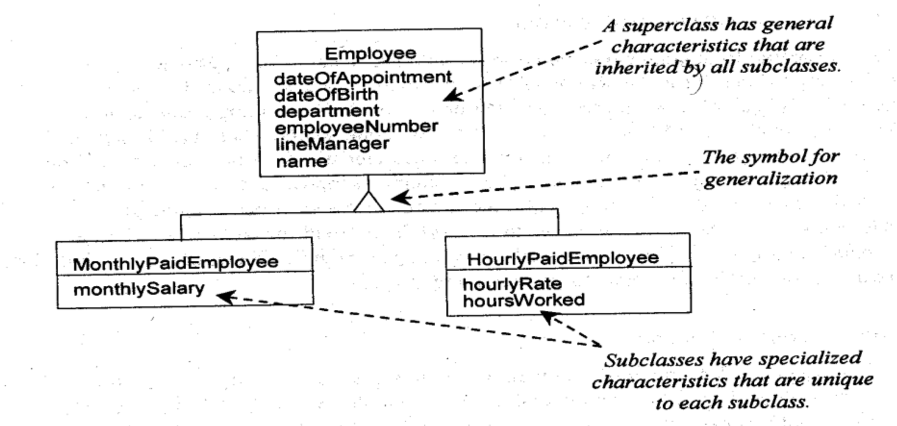


Generalisation

- UML Definition:
  - *Taxonomic(hierarchical) relationship between a more general element and a more specific element that is fully consistent with the first element and that adds additional information*

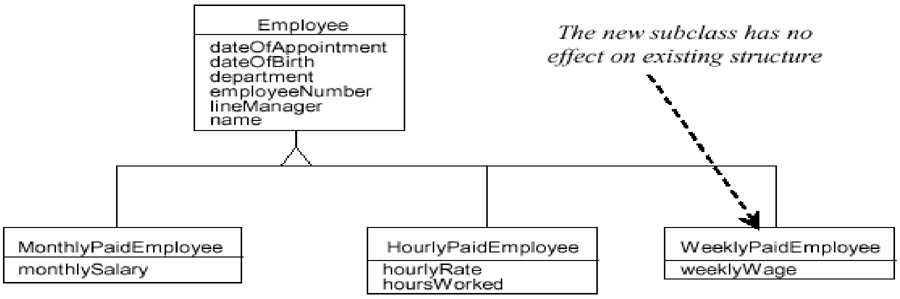
Generalisation in Object Classes

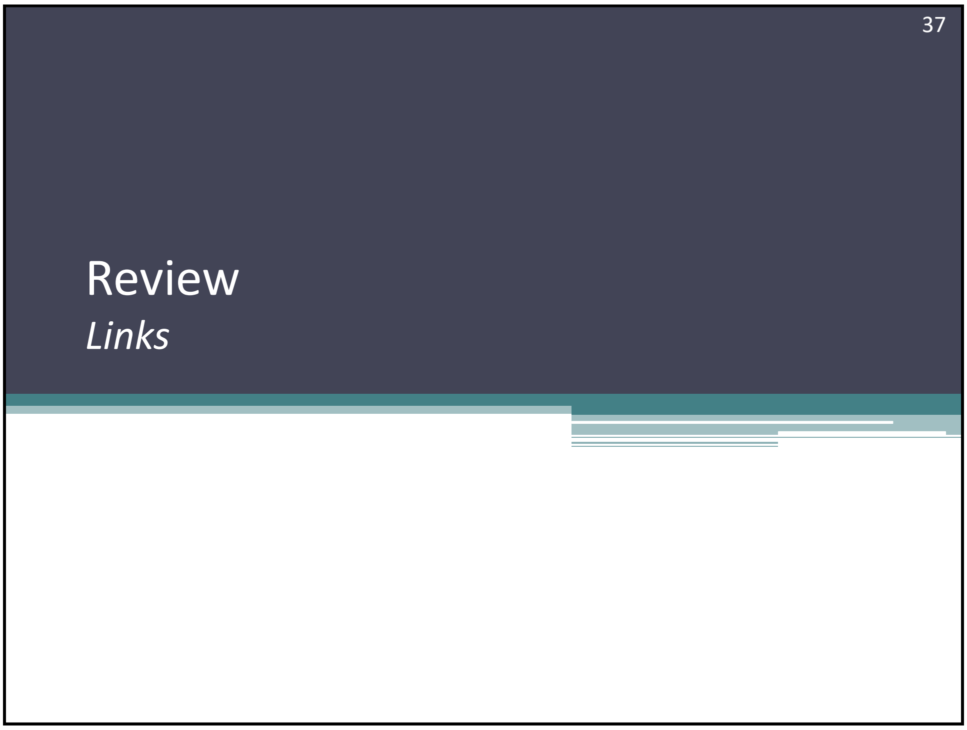
- Arrange real world object classes into hierarchies.
- Models the difference and similarity between classes


Employee
dateOfAppointment
dateOfBirth
department
employeeNumber
lineManager
name
MonthlyPaidEmployee
monthlySalary
HourlyPaidEmployee
hourlyRate
hoursWorked
A superclass has general characteristics that are inherited by all subclasses.
The symbol for generalization
Subclasses have specialized characteristics that are unique to each subclass.

Generalisation in Object Classes - Inheritance

- When two classes are arranged in a hierarchy the more specialised class (subclass) will ***inherit*** all the characteristics from the more generalised class (superclass)
- The subclass will have at least one additional detail not inherited form its superclass

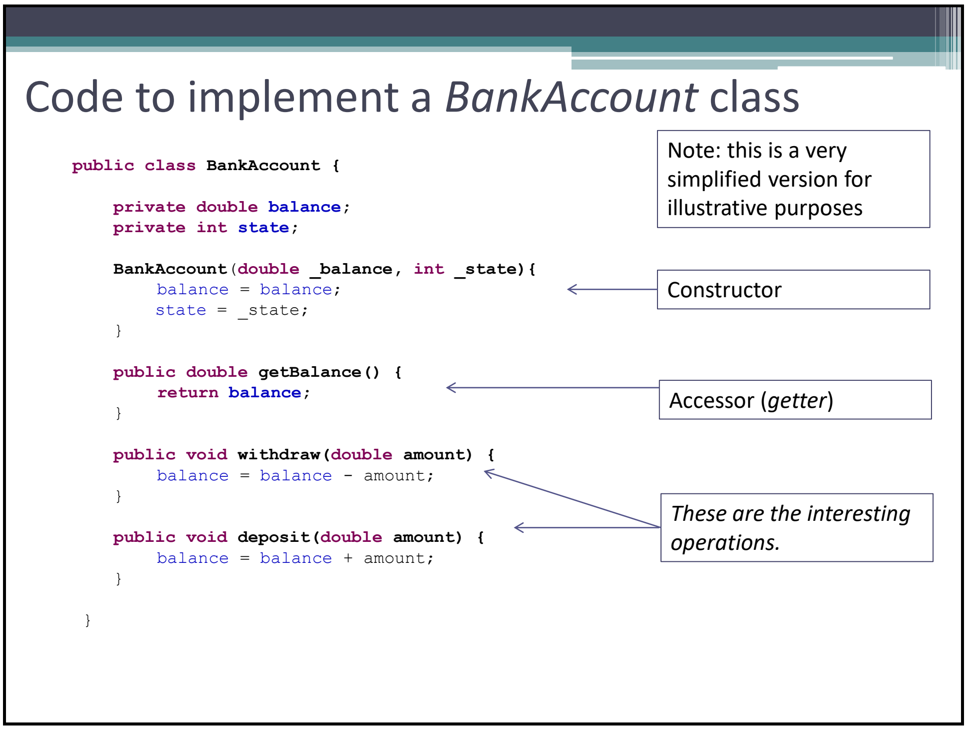




37

Review

Links



Code to implement a *BankAccount* class

```
public class BankAccount {

    private double balance;
    private int state;

    BankAccount(double _balance, int _state){
        balance = balance;
        state = _state;
    }

    public double getBalance() {
        return balance;
    }

    public void withdraw(double amount) {
        balance = balance - amount;
    }

    public void deposit(double amount) {
        balance = balance + amount;
    }

 }
```

Note: this is a very simplified version for illustrative purposes

Constructor

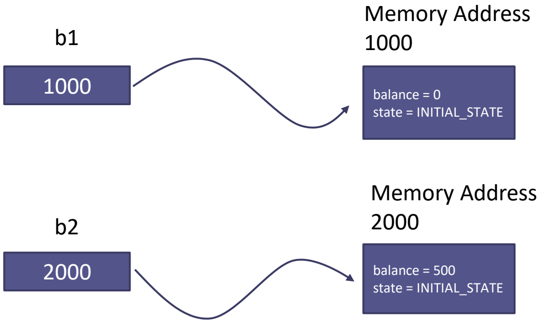
Accessor (*getter*)

These are the interesting operations.

Object References

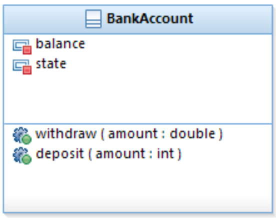
Consider two instances (object) of the *BankAccount* class in memory

BankAccount **b1** = new BankAccount (0, INITIAL_STATE) ;
BankAccount **b2** = new BankAccount (500, INITIAL_STATE) ;



A Simple Class Diagram

Take the simplified example of a *BankAccount* class



First compartment contains the name of the class.

Second compartment contains the attributes (data) that an object of type *BankAccount* can have.

Third compartment contains the operations that an object of type *BankAccount* can perform.

Note1: accessor and mutator operations are not generally modelled.
Note2: In the diagram above, the red squares on the attributes denote that the attributes are *private* (a minus sign, "-", can also denote this). The green circles on the operations denote that they are *public* (a plus sign, "+", can also denotes this).

Object Links – Consider the following classes discussed previously

```
public class SomeClient {

    SomeClient (Circle circleA,
                Circle circleB)
    {
        circleA.setRadius(10);
        circleB.setRadius(12);
    }
}
```

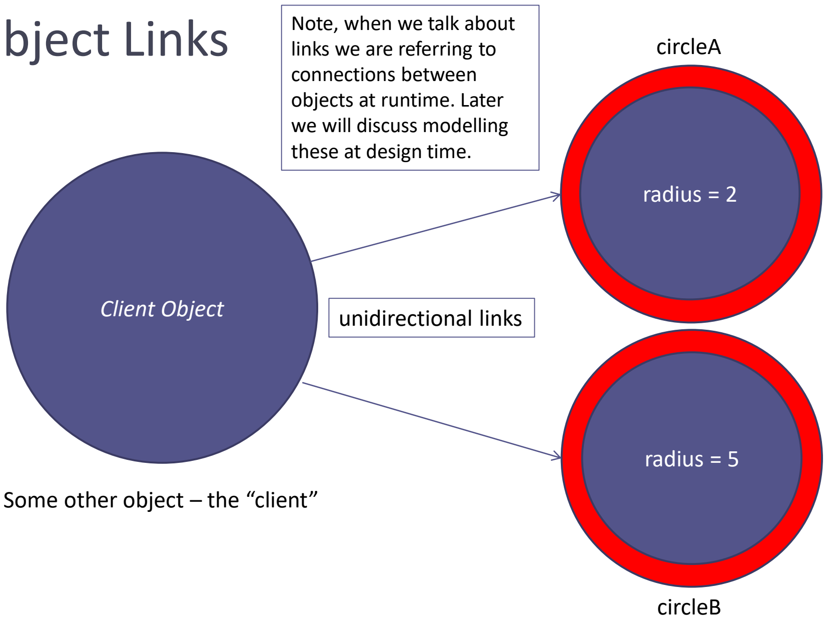
```
public class Circle {

  private int radius;

  public int getRadius() {
         return radius;
  }

  public void setRadius(int new_radius)
  {
         radius = new_radius;
  }

}
```

Consider an instance of the above class - it will contain two *references* to two Circle objects. This means that at runtime it will be *linked* to those objects

Vice versa, the two instances of this class (circleA and circleB) are *linked* to the client object

The actual *link* is provided by the references (*circleA* and *circleB*) within the client object – interestingly, this means that only the client object "knows" about the link. In this sense, the link is *uni-directional*.

Object Links



Note, when we talk about links we are referring to connections between objects at runtime. Later we will discuss modelling these at design time.

Some other object – the "client"

The Object Model

Common Model Shared by UML and OO Languages

- The *object model* is a general way of thinking about the structure of OO programs.
- The fundamental property of the *object model* is that **computation** takes place **in** and **between** *objects*.
- The *object model* is the common computational model shared by UML and object-oriented programming languages.

Role of an Object’s Data and Methods

- Individual objects are responsible for maintaining part of a system’s **data** and for implementing aspects of its overall **functionality**.
- An object supports the data stored by that object and the methods, or functions, to access and update the data that it contains.

Network of Objects

- Relationships between the data stored in individual objects must be recorded.
- The global behaviour of the system emerges from the interaction of many distinct objects.
- These requirements are supported by allowing objects to be **linked** together.
  - This is typically achieved by enabling one object to hold a reference to another.

Network of Objects

- The object model views a running program as a network, or graph of objects.
- The objects form the **nodes** in the graph, and the arcs connecting the objects are known as ***links***.
- The object network represents relationships between data entities.
- Objects can be created and destroyed at run-time and the links between them can also be changed.
  - The structure, or topology, of the object network is therefore highly dynamic, changing as a program runs.

Object Messages

- The links between objects also serve as communication paths, which enable objects to interact by sending *messages* to each other.

- Messages are analogous to function calls:
  - they are typically requests for the receiving object to perform one of its methods and can be accompanied by data parameterizing the message.

- An object's response to a message can be to send messages to other objects and in this way computation can spread and be shared across the network, involving many objects in the response to an initial message.

The Dual Role of the Object Model in Design

- UML (Unified Modelling Language) diagrams play the same role as source code - defining in a general structural way what can happen at run-time.

- These diagrams fall into two main categories:
  - *Static* diagrams:
    - describe the kinds of connections that can exist between objects (one object referencing another).
  - *Dynamic* diagrams:
    - describe the messages that can be passed between objects and the effect on an object of receiving a message.

The Dual Role of the Object Model in Design

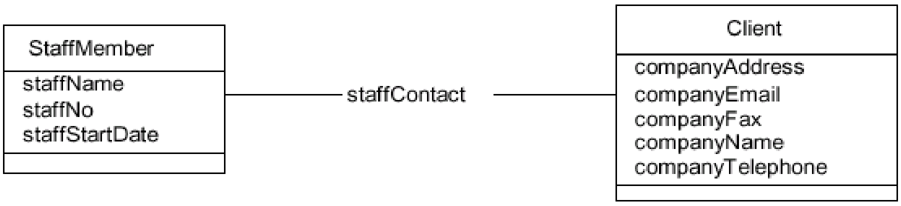
- The dual role of the object model makes it possible to relate UML design notations to actual programs
  - makes UML a suitable language for designing and documenting object-oriented programs.
- Because of the shared (distribution of data and process across the object network) object model:
  - UML diagrams can easily be implemented
  - object-oriented programs can easily be documented in UML
  - Reverse engineering:
    - creating a UML model for existing code
    - often useful in dealing with undocumented legacy systems

Review

Associations

Association of Classes

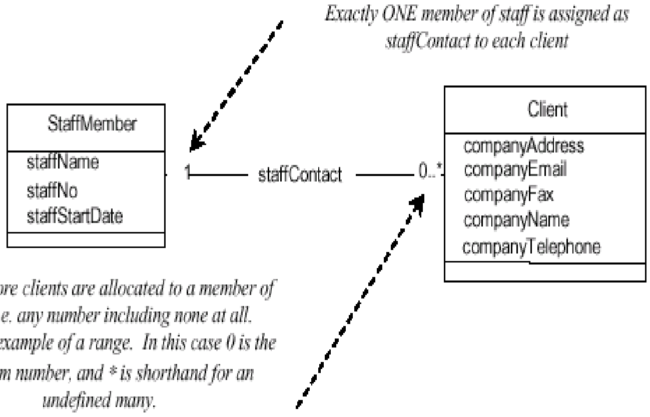
- Just as a ***link*** connects *two instances of a class*, an ***association*** connects *two classes*.
- Modelling all the possibilities of links is not feasible so we model association of classes
- An association between two classes represents the possibility that objects of the classes may be linked.
- Associations are modelled initially and a sample of links may be noted for verification of the model
- Consider the following two classes…



Multiplicity of Association

- A Staff Member may be associated with more than one instance of Client but a client may have only one Staff Member looking after them.
- A Patient may have several Doctors and a Doctor may look after several Patients.
- A Bank Account may be associated with one and only one Account Holder.
- A joint bank account may be associated with two and only two account holders.
- These business rules must be modelled correctly and implemented in the software.

Multiplicity Types 1:M



Exactly ONE member of staff is assigned as staffContact to each client

*Zero or more clients are allocated to a member of staf— i.e. any number including none at all. This is an example of a range. In this case 0 is the minimum number, and * is shorthand for an undefined many.*

Multiplicity Types 1:M

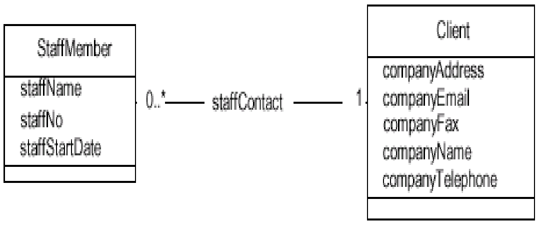
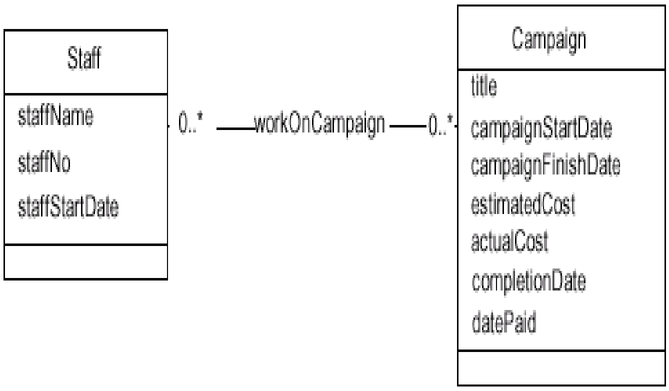


Figure 6.10 *Every* `StaffMember` *must be the contact for exactly one* `Client`*, and a* `Client` *may have no contact, or one or more.*

Multiplicity Types M:N both optional



Multiplicity Types M:N one optional

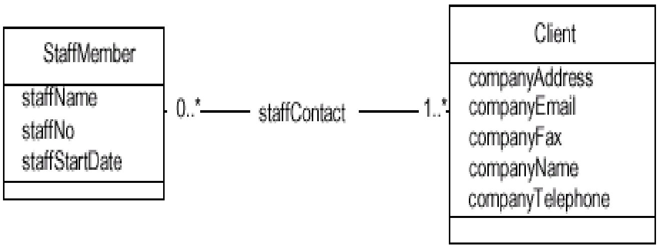


Figure 6.11 *Every* StaffMember *must be the contact for one or more* Clients, *and a* Client *may have no contact, or one or more.*