Object-Oriented Programming and UML

Lecture 3

Based on slides from Leigh Dodds (mostly verbatim)

Chapters 2 and 3 of MSD

Overview

- Principles of Object Oriented Programming
  - What is OOP? Why is it important?
  - Basic principles and advantages
- The Unified Modelling Language
  - UML Class Diagrams

Object-Oriented Programming

- Understanding OOP is fundamental to writing good Java applications
    - Improves design of your code
    - Improves understanding of the Java APIs
- There are several concepts underlying OOP:
    - Abstract Types (Classes)
    - Encapsulation (or Information Hiding)
    - Aggregation
    - Inheritance
    - Polymorphism

What is OOP?

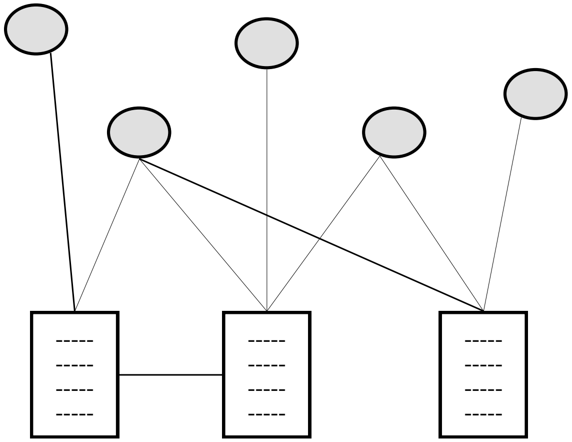
- Modelling real-world objects in software
- Why design applications in this way?
  - We naturally *class*ify objects into different *types*.
  - By attempting to do this with software aim to make it more maintainable, understandable and easier to reuse
- In a conventional programming we typically:
  - decompose it into a series of functions,
  - define data structures that those functions act upon
  - there is no relationship between the two other than the functions act on the data

What is OOP?

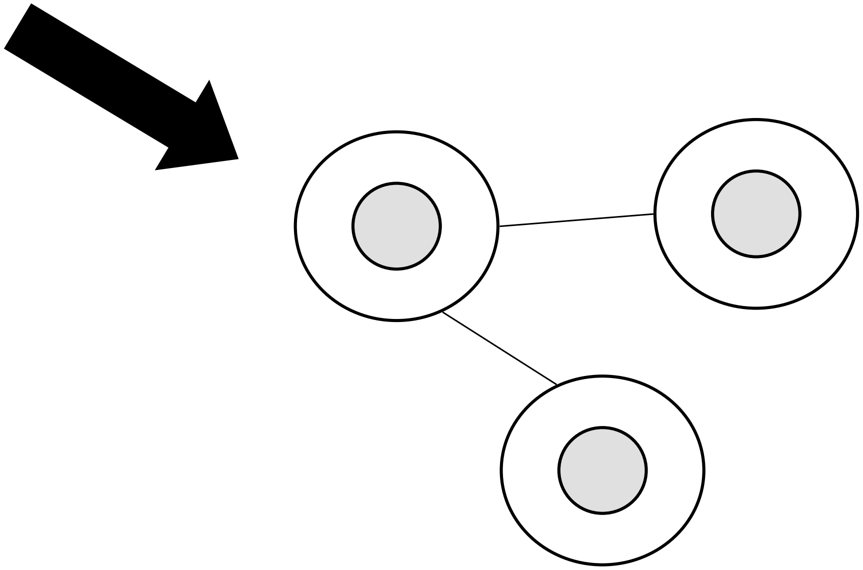
- How is OOP different from conventional programming?
  - Decompose the application into *abstract data types* by identifying some useful entities/abstractions
  - An abstract type is made up of a series of behaviours and the data that those behaviours use.
- Similar to database modelling, only the types have both behaviour and state (data)

Abstract Data Types

- Identifying abstract types is part of the modelling/design process
  - The types that are useful to model may vary according to the individual application
  - For example a payroll system might need to know about Departments, Employees, Managers, Salaries, etc
  - An E-Commerce application may need to know about Users, Shopping Carts, Products, etc
- Object-oriented languages provide a way to define abstract data types, and then create *objects* from them
  - It’s a template (or ‘cookie cutter’) from which we can create new objects
  - For example, a Car class might have attributes of speed, colour, and behaviours of accelerate, brake, etc
  - An individual Car *object* will have the same behaviours but its own values assigned to the attributes (e.g. 30mph, Red, etc)



"Conventional Programming" --
Functions or Procedures operating on independent data

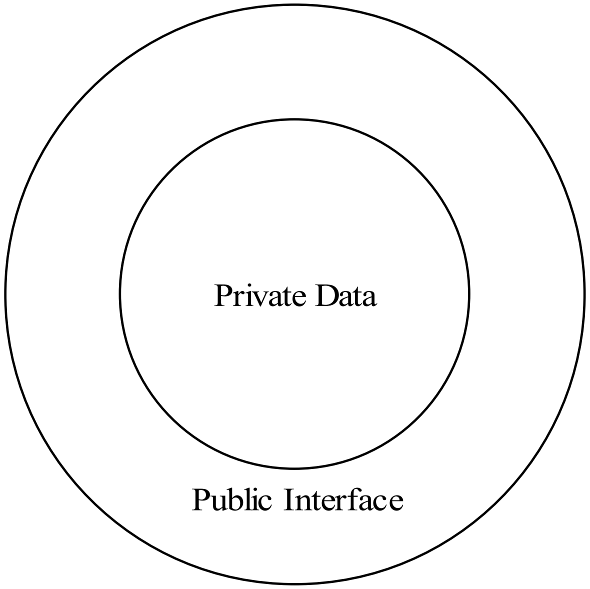


"OO Programming" --
Abstract Types combine data and behaviour

Encapsulation

- The data (state) of an object is private – it cannot be accessed directly.
- The state can only be changed through its behaviour, otherwise known as its public *interface* or *contract*
- This is called *encapsulation*

"The Doughnut Diagram"
Showing that an object has private state and public behaviour. State can only be changed by invoking some behaviour



Encapsulation

- Main benefit of encapsulation
  - Internal state and processes can be changed independently of the public interface
  - Limits the amount of large-scale changes required to a system

What is an OO program?

- What does an OO program consist of?
  - A series of objects that use each others behaviours in order to carry out some desired functionality
  - When one object invokes some behaviour of another it sends it a *message*
  - In Java terms it invokes a *method* of the other object
  - A method is the implementation of a given behaviour.
- OO programs are intrinsically modular
  - Objects are only related by their public behaviour (methods)
  - Therefore objects can be swapped in and out as required (e.g. for a more efficient version)
  - This is another advantage of OO systems

Aggregation

- Aggregation is the ability to create new classes out of existing classes
  - Treating them as building blocks or components
- Aggregation allows reuse of existing code
  - “Holy Grail” of software engineering
- Two forms of aggregation
- Whole-Part relationships
  - Car is made of Engine, Chassis, Wheels
- Containment relationships
  - A Shopping Cart contains several Products
  - A List contains several Items

Inheritance

- Inheritance is the ability to define a new class in terms of an existing class
  - The existing class is the *parent*, *base* or *superclass*
  - The new class is the *child*, *derived* or *subclass*
- The child class inherits all of the attributes and behaviour of its parent class
  - It can then add new attributes or behaviour
  - Or even alter the implementation of existing behaviour
- Inheritance is therefore another form of code reuse

Polymorphism

- Means ‘many forms’
- Difficult to describe, easier to show, so we’ll look at this one in a later lesson
- In brief though, polymorphism allows two different classes to respond to the same message in different ways
- E.g. both a Plane and a Car could respond to a ‘turnLeft’ message,
  - however the means of responding to that message (turning wheels, or banking wings) is very different for each.
- Allows objects to be treated as if they’re identical

Summary!

- In OO programming we
  - Define classes
  - Create objects from them
  - Combine those objects together to create an application
- Benefits of OO programming
  - Easier to understand (closer to how we view the world)
  - Easier to maintain (localised changes)
  - Modular (classes and objects)
  - Good level of code reuse (aggregation and inheritance)

Overview

- Principles of Object Oriented Programming
  - What is OOP?
  - Why is it important?
- The Unified Modelling Language
  - UML Class Diagrams

Unified Modelling Language

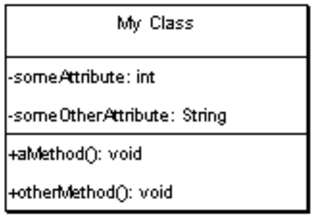
- UML is a diagramming tool for describing and documenting object oriented applications
- Programming language independent
- Used for modelling an application before its engineered
- Twelve different diagrams in all, with many complex details
- Generally though only two of these are used regularly
  - Class diagrams
  - Sequence diagrams

Unified Modelling Language

- Class Diagrams
  - Describe classes and interfaces
  - …their properties
  - …their public interface
  - …and their relationships (e.g. inheritance, aggregation)
- Sequence Diagrams
  - Describe how objects send messages to one another
  - Useful for describing how a particular part of an application works
- We'll be covering just class diagrams
  - Very useful for describing APIs and discussing OO applications

UML -- Classes

- Box with 3 sections
- The top contains the class name
- The middle lists the classes attributes
- The bottom lists the classes methods
- Can indicate parameters and return types to methods, as well as their visibility



My Class

-someAttribute: int

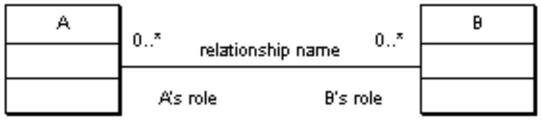
-someOtherAttribute: String

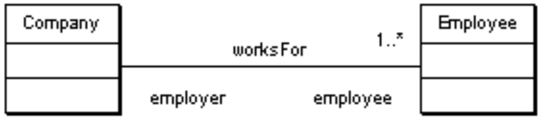
+aMethod(): void

+otherMethod(): void

UML -- Association

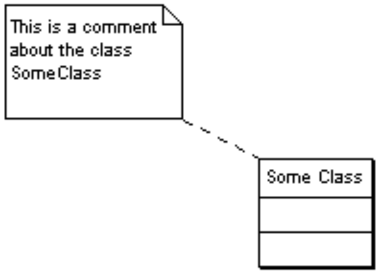
- A line between two classes indicates a relationship
- Extra information can be added to describe the relationship
- Including
  - Its name
  - The roles that the classes play
  - The *cardinality* of the relationship (how many objects are involved)
- E.g. a Person worksFor a Company, which has many employees





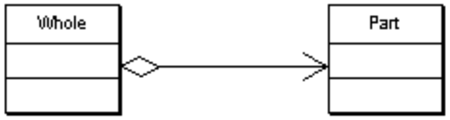
UML -- Comments

- Useful for adding text for the readers of your diagram
- The symbol looks like a little post-it note, with a dotted line joining it to the class or relationship that its describing



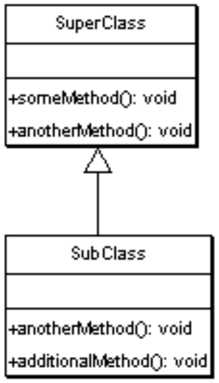
UML -- Aggregation

- Aggregation (a whole-part relationship) is shown by a line with clear diamond.
- As aggregation is a form of relationship you can also add the usual extra information
- I.e.
  - Name
  - Roles
  - Cardinality



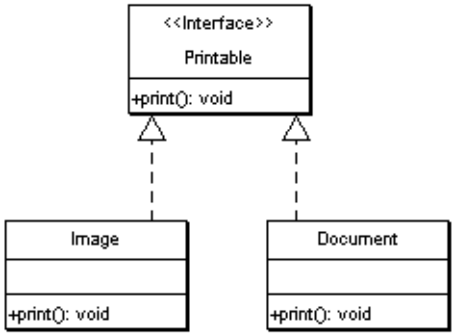
UML -- Inheritance

- Inheritance is shown by a solid arrow from the sub-class to the super-class
- The sub-class doesn’t list its super-class attributes or methods,
- *unless* its providing its own alternate version (I.e. is extending the behaviour of the base class)

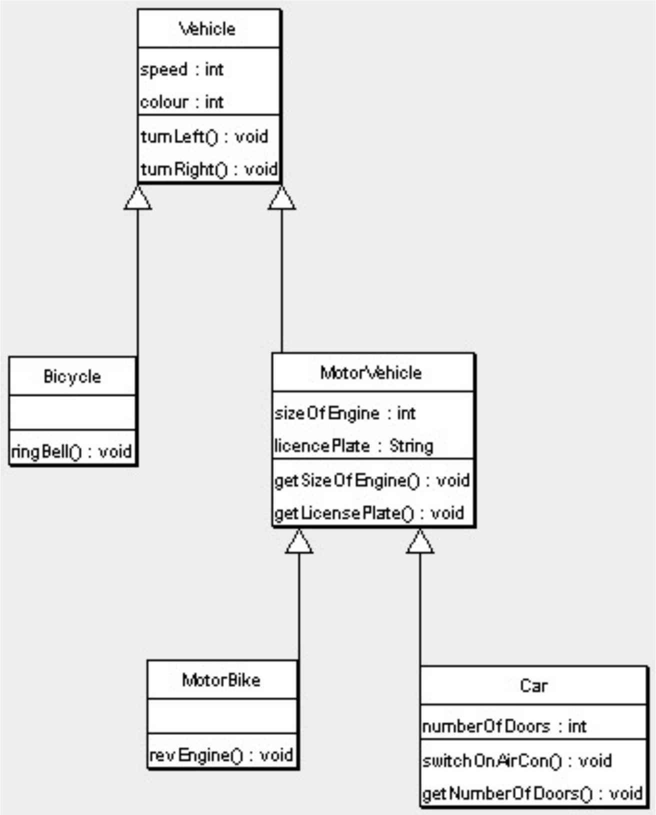


UML -- Interfaces

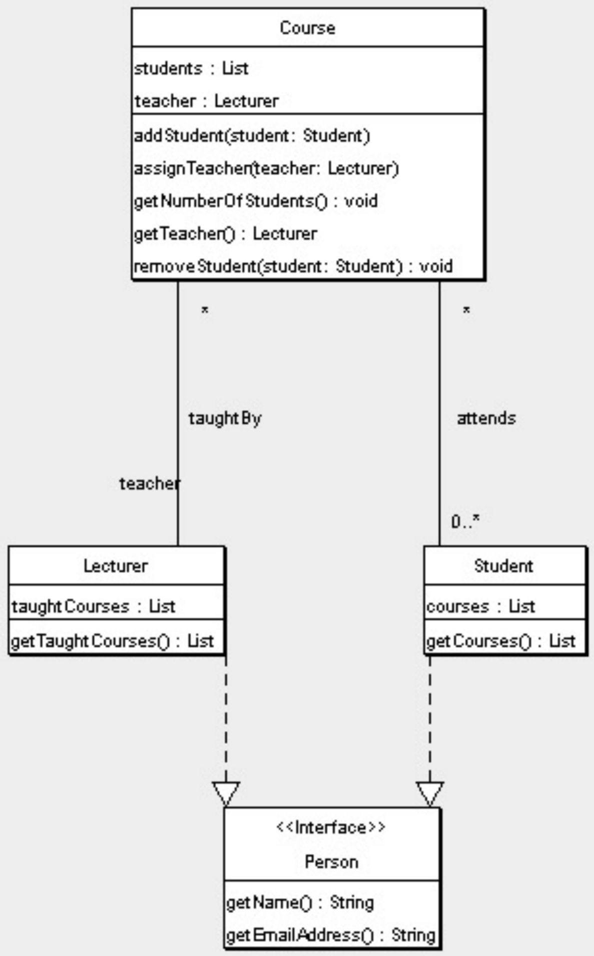
- Interfaces are a way to specify behaviour (a public contract) without data or implementation.
- Interfaces are classed with an extra label next to their name: `<<Interface>>`
- A dotted arrow from a class to an interface explains that the class fulfills the contract specified by that interface



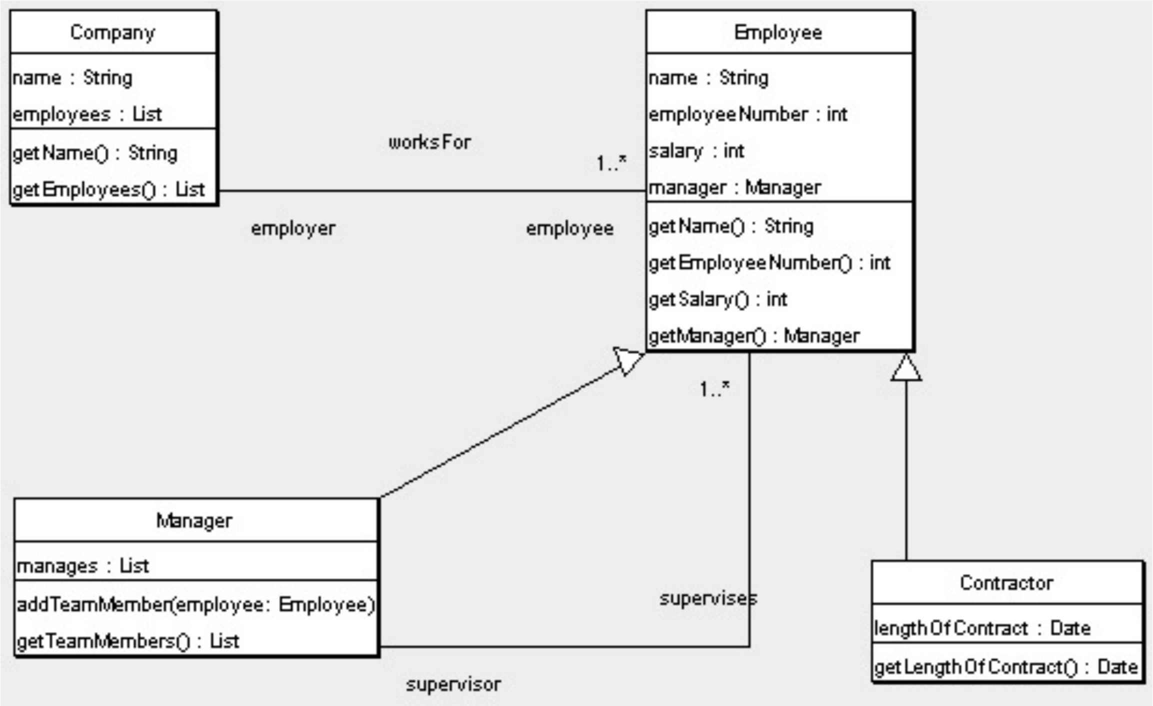
Example #1


Vehicle
speed : int
colour : int
turnLeft() : void
turnRight() : void
Bicycle
ringBell() : void
MotorVehicle
sizeOfEngine : int
licencePlate : String
getSizeOfEngine() : void
getLicensePlate() : void
MotorBike
revEngine() : void
Car
numberOfDoors : int
switchOnAirCon() : void
getNumberOfDoors() : void

Example #2


Course
students : List
teacher : Lecturer
addStudent(student: Student)
assignTeacher(teacher: Lecturer)
getNumberOfStudents() : void
getTeacher() : Lecturer
removeStudent(student: Student) : void
*
*
taughtBy
attends
teacher
0..*
Lecturer
taughtCourses : List
getTaughtCourses() : List
Student
courses : List
getCourses() : List
<<Interface>>
Person
getName() : String
getEmailAddress() : String

Example #3



Company

name : String
employees : List

getName() : String
getEmployees() : List

Employee

name : String
employeeNumber : int
salary : int
manager : Manager

getName() : String
getEmployeeNumber() : int
getSalary() : int
getManager() : Manager

Manager

manages : List

addTeamMember(employee: Employee)
getTeamMembers() : List

Contractor

lengthOfContract : Date

getLengthOfContract() : Date