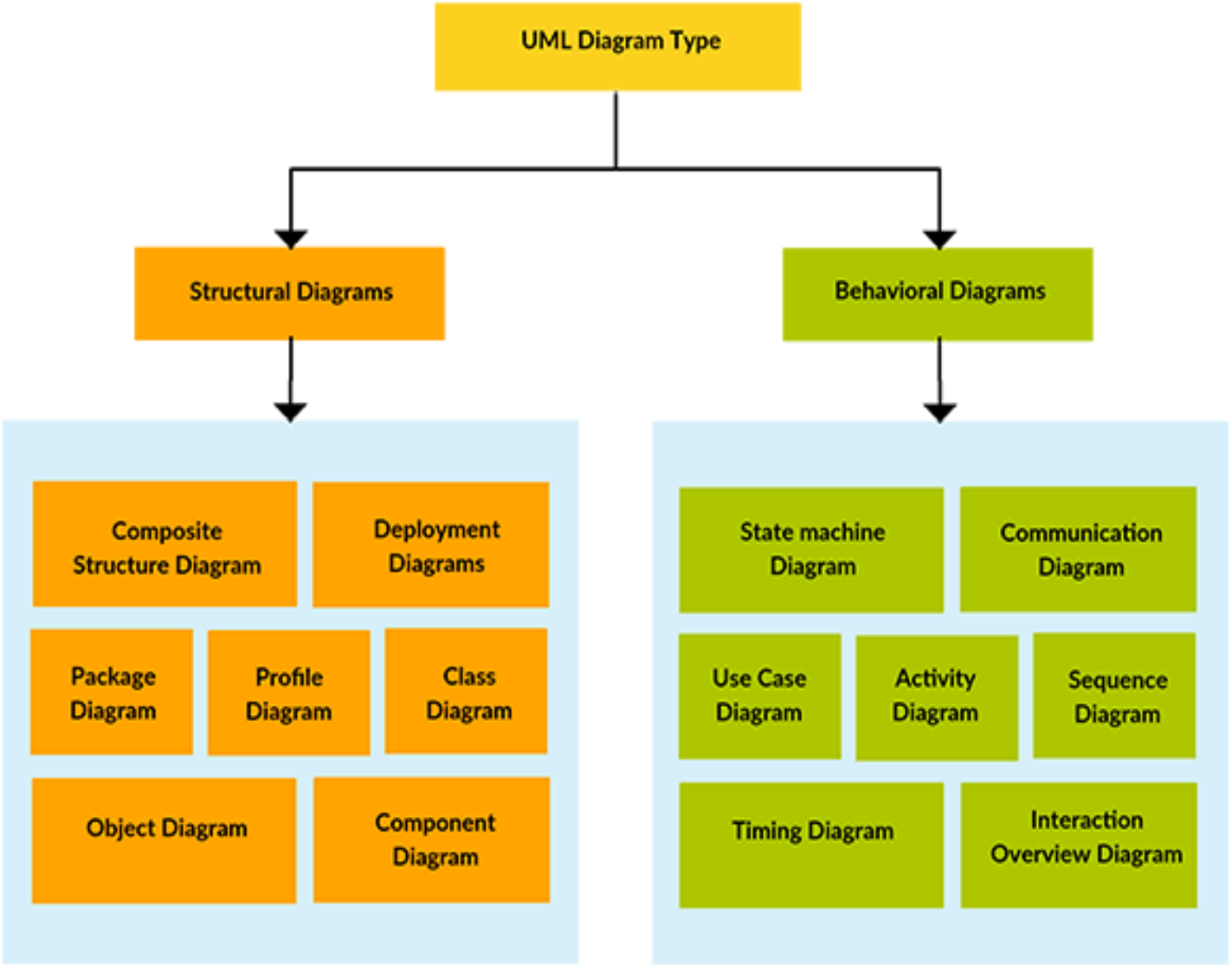
COMP 2511
Object Oriented Design & Programming

- So far,
  - we have defined classes and object instances
  - explored key OO principles (abstraction, encapsulation)
- let us now look at **relationships between objects** e.g.,
  - a dog **is-a** mammal
  - an instructor **teaches** a student
  - a university **enrols** students
- Relationships between objects can be broadly classified as:
  - Inheritance
  - Association

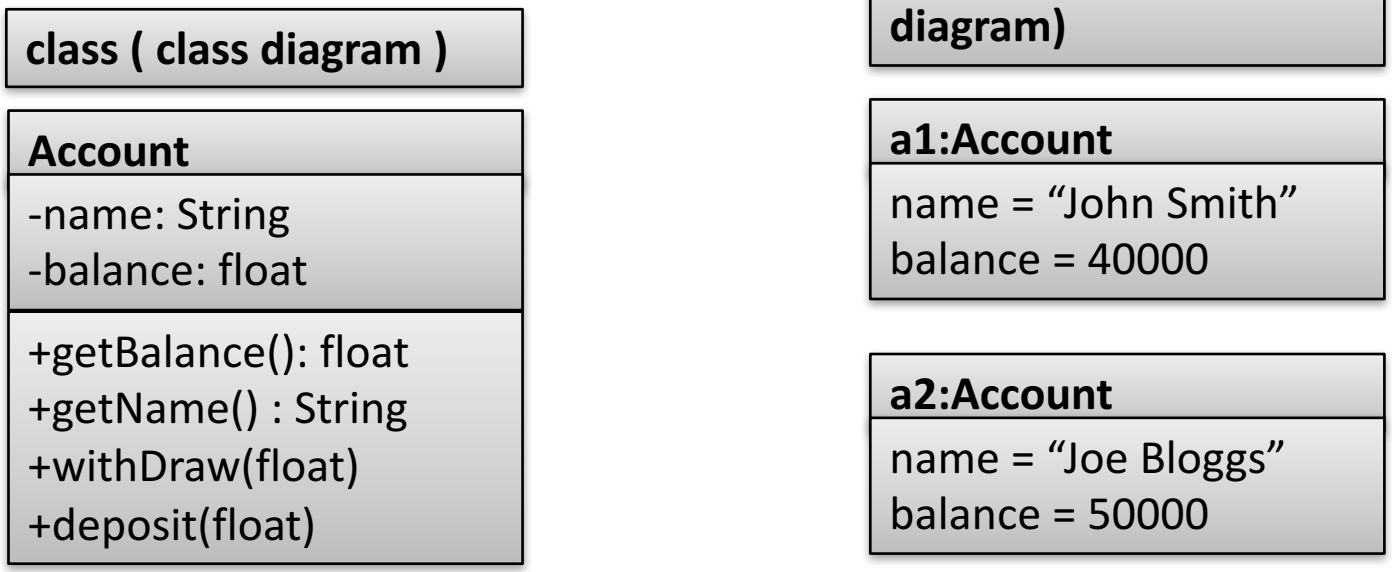
What is UML?

- Programming languages not abstract enough for OO design
- Software design must be expressed in a modelling language – UML, BPMN etc
  - UML (Unified Modelling Language) - an open source, graphical language to model software solutions, application structures, system behaviour and business processes (http://www.uml.org/)
  - A language independent modelling tool
  - Sometimes, used for auto code-generation

UML Diagram Types



Representing classes in UML



Relationships (1) – Inheritance

- So far, we have logically grouped objects with common characteristics into a class, but what if these objects had some special features?

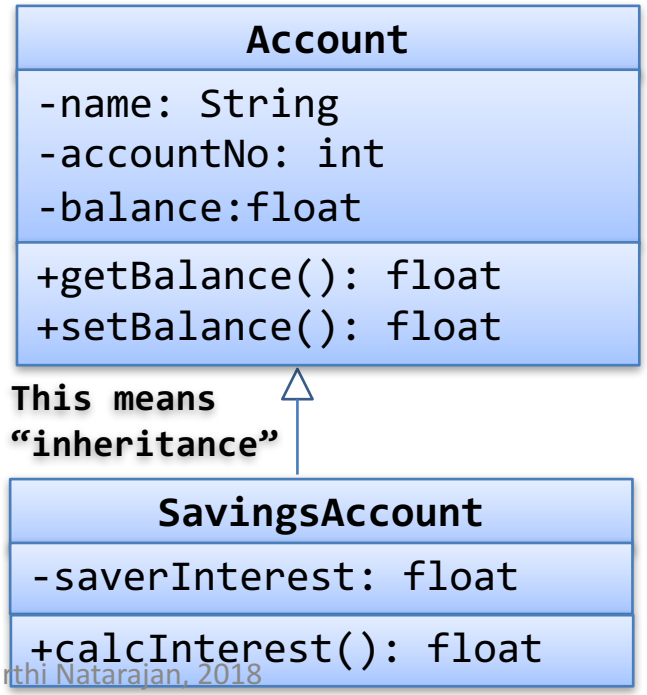
  e.g., if we wanted to store that sports car has spoilers

- Answer is inheritance - models a relationship between classes in which one class represents a more general concept (parent or base class) and another a more specialised class (sub-class)
- Inheritance models a "is-a" type of relationship e.g.,

  - a savings account is a type of bank account
  - a dog **is-a** type of pet
  - a manager **is-a** type of employee
  - a rectangle **is-a** type of 2D shape

Inheritance

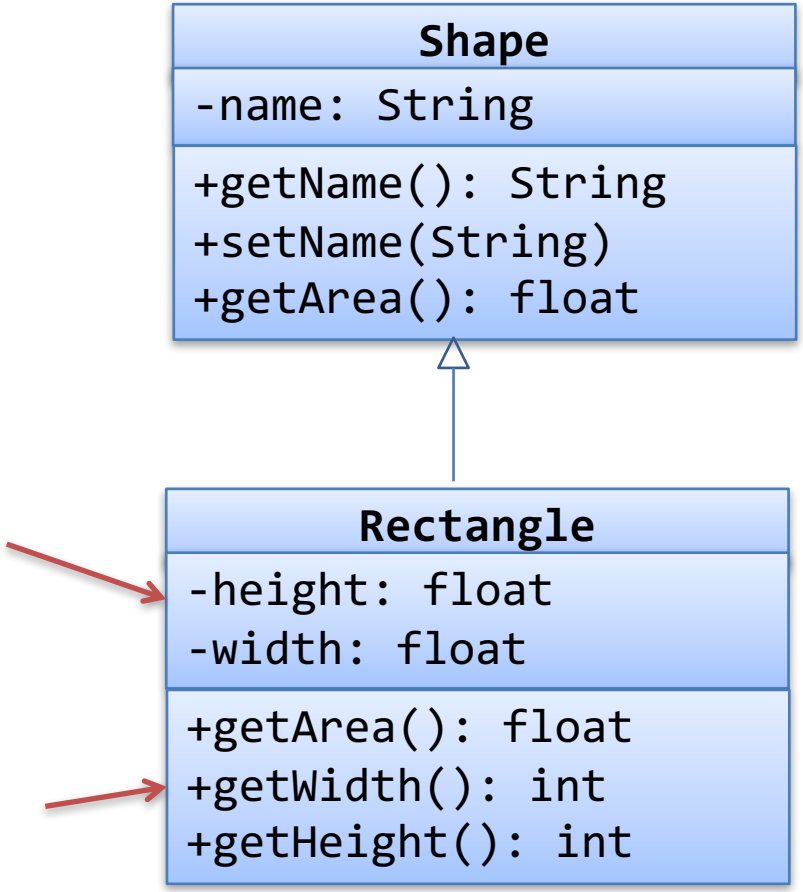
- To implement inheritance, we
  - create a new class (sub class) , that inherits common properties and behaviour from a base class (parent class or super class)
    - We say the child class *inherits/ is-derived from* the parent class
  - sub-class can extend the parent class by defining additional properties and behaviour specific to the inherited group of objects



Superclass: Account (parent class)
class Account defines *name*, *accountNo*, *balance*

Subclass class: SavingsAccount (child)
- extends Account (SavingsAccount **is an** Account)
- adds its own attributes and methods e.g., *saverInterest & calcInterest()*

Inheritance – another example



class Rectangle *extends* Shape adding attributes *height* , *width*

class Rectangle *overrides* method *getArea()* to provide its own implementation

Using super in subclass

- To invoke a parent's method or to access a field (non-private) field in the parent class, use super e.g., `class SavingsAccount` can invoke fields and methods in parent `class Account` as:

```
super.account
super.getBalance()
```

- The invoked method does not have to be defined in the immediate parent class, but could be inherited from some class further in the inheritance hierarchy
- To invoke the constructor in the parent's class use `super()`

Constructors are not inherited

- A sub-class **does not inherit constructors**
- To create an instance of a subclass, there are two options:
  1. Use the default "no-arg" constructor
     - This default constructor will make a default call to `super()`, which is the constructor in the parent class
  2. Define a constructor in the sub-class
     - then, a default constructor is no longer provided
     - use `super()` to invoke the parent's constructor

       e.g., `SavingsAccount` calls the constructor of the `Account` as `super (bsb, accountNo, salary)`
     - Call to `super()` must be the first statement of the constructor. If this call is not made, a default call to super() is inserted

Overriding Methods

- A sub-class can *override* methods in the parent class with its own specialised behaviour

  `class Rectangle` overrides method `getArea()` in parent `class Shape` to provide its own specific implementation

```java
public class Shape {

    public String color;

    public Shape(String color) {
        this.color = color;
    }
    /*
     * @return Returns the area of the shape
     */
    public float getArea() {
        return 0;
    }
}
```

```java
public class Rectangle extends Shape {

    public int height;
    public int width;

    public Rectangle(String color) {
        super(color);
    }

    @Override
    public float getArea() {
        return this.height * this.width;
    }
}
```

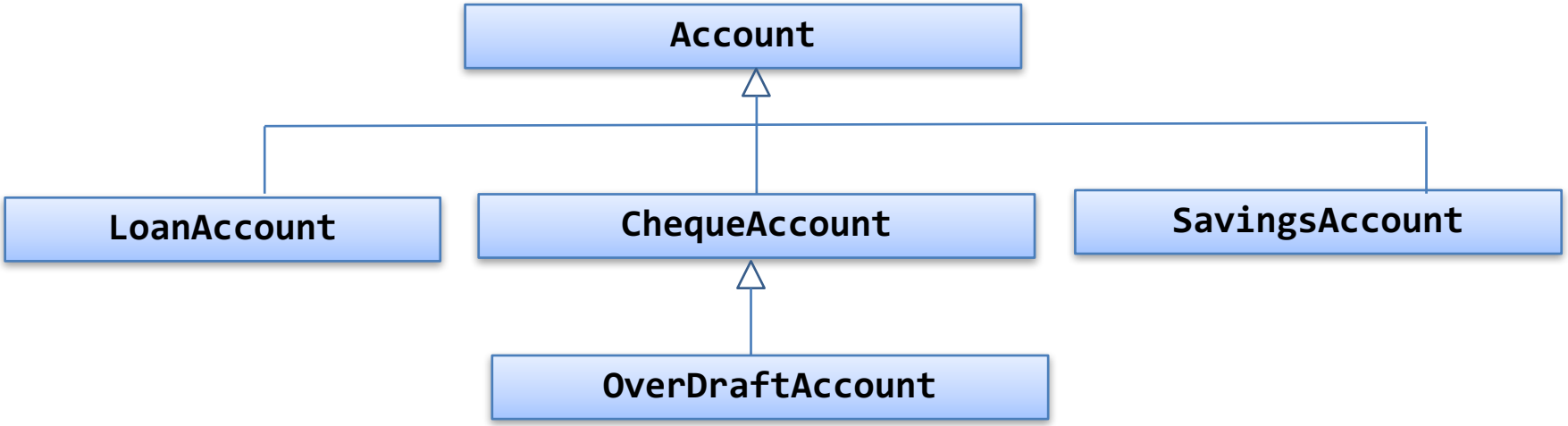
- Method name and order of arguments of a child method must match the signature of the corresponding method in parent class
- Overridden methods cannot be less accessible

You should know how to …

- Use `package` and `import` statements
- Use `access modifiers` in Java
- Make classes immutable
  - No setter methods, use constructor
- Understand Inheritance, polymorphism
  - Overriding `toString()` and `equals()` in objects
  - Overriding `static` methods (Is this possible?)
  - Overloading constructors

Single Inheritance

- The Java language allows a class to extend only one other class – single inheritance
- Multiple inheritance allows you to inherit from more than one super class
  - Supported by languages such as C++, Python



Understanding object types

- All object references have a type
- An object, which is an instance of a class has a type, but is also an instance of its parent class e.g.,
- A sub-class ***C*** has all the members of its parent class ***P***

```
Rectangle aRect =  new Rectangle();
// But aRect is also an instance of class Shape
```

- Hence, where-ever an object of class **C**, it can be referenced as an object of class **P**

```
Shape aRect =  new Rectangle();
Account myAccount = new SavingsAccount();
```

- Java, gives you the ability to refer to an object using its actual form or parent form

Polymorphism

Polymorphism means "many forms": an important OO principle that supports software reuse and maintenance

- Here the *variable* `s1` is said to be polymorphic as it can refer to objects of different forms

```
Shape s1 = new Rectangle();
s1 = new Circle();
```

  - These assignments are legal as `Rectangle` and `Circle` are both types of `Shape`

- However, the following does not compile:

```
s1.getHeight();
```

  - Using the variable `s1`, you can only access parts of the object that belong to the `class Shape`; the `Rectangle` specific components are hidden;
  - The Java compiler recognises that `s1` is a `Shape` NOT a `Rectangle`

Polymorphism

- Here, the function `getArea()` defined in class `Shape` and `Rectangle` is said to be "`polymorphic`" as the function can be applied on objects of different classes to achieve the same semantic result e.g.,

```
Shape s1 = new Shape();
Rectangle r1 = new Rectangle();
```

Calling methods `s1.getArea()` and `r1.getArea()` invokes different behaviour but achieve the semantic result

Dynamic Binding with Polymorphism

But, what happens here?

```
Shape s1 = Rectangle();
```

A *variable* is polymorphic, but an *object instance* has only one type (form), defined when it is instantiated.

Here, dynamic binding or Virtual Method Invocation ensures ensures that when a method is invoked, you get the behaviour associated with the object to which the variable refers to at runtime

The behaviour is not determined by the compile time type of the variable

The **instanceOf** operator

- As objects can be referenced using their parent classes, it is sometimes necessary to know what is the actual type of the object at run-time

- Use the **`instanceOf`** operator

```
public void getCoordinates(Shape s)
  if (s instanceOf Rectangle) {
       // do something
  }
  else if (s instanceOf Circle) {
       // do something
  }
```

Access Modifiers in Java: Summary

Modifier	Same Class	Same Package	Sub Class	Everyone
public	√	√	√	√
protected	√	√	√	
default	√	√		
private	√			

Next …

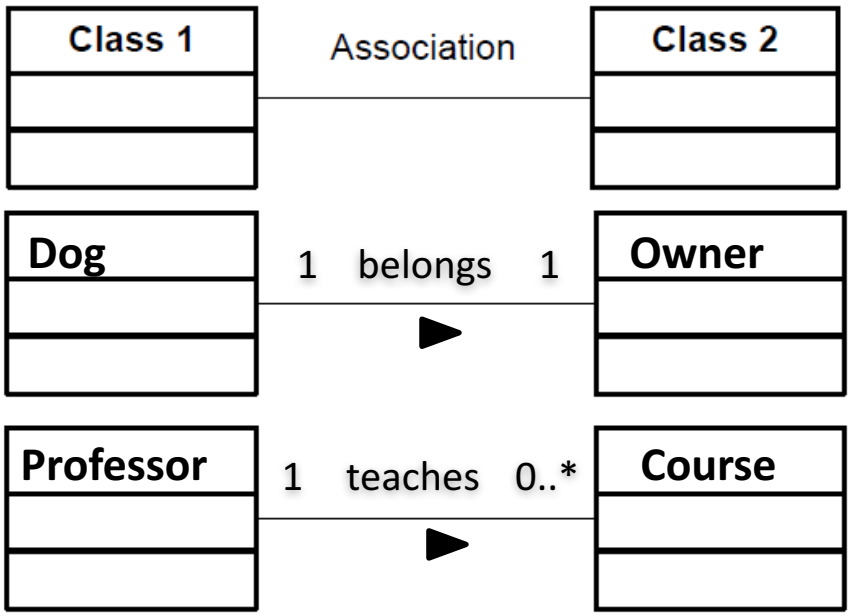
- Relationships between classes (association , composition, aggregation)
- Creating a domain model applying object-oriented design principles…

Relationships (2) – Association

- Association is a special type of relationship between two classes, that shows that the two classes are:
  - linked to each other

    e.g., a lecturer *teaches* a course-offering
  - or combined into some kind of *"has-a"* relationship, where one class "contains" another class

  e.g., a course-offering *has* students
- Modelled in UML as a line between two classes



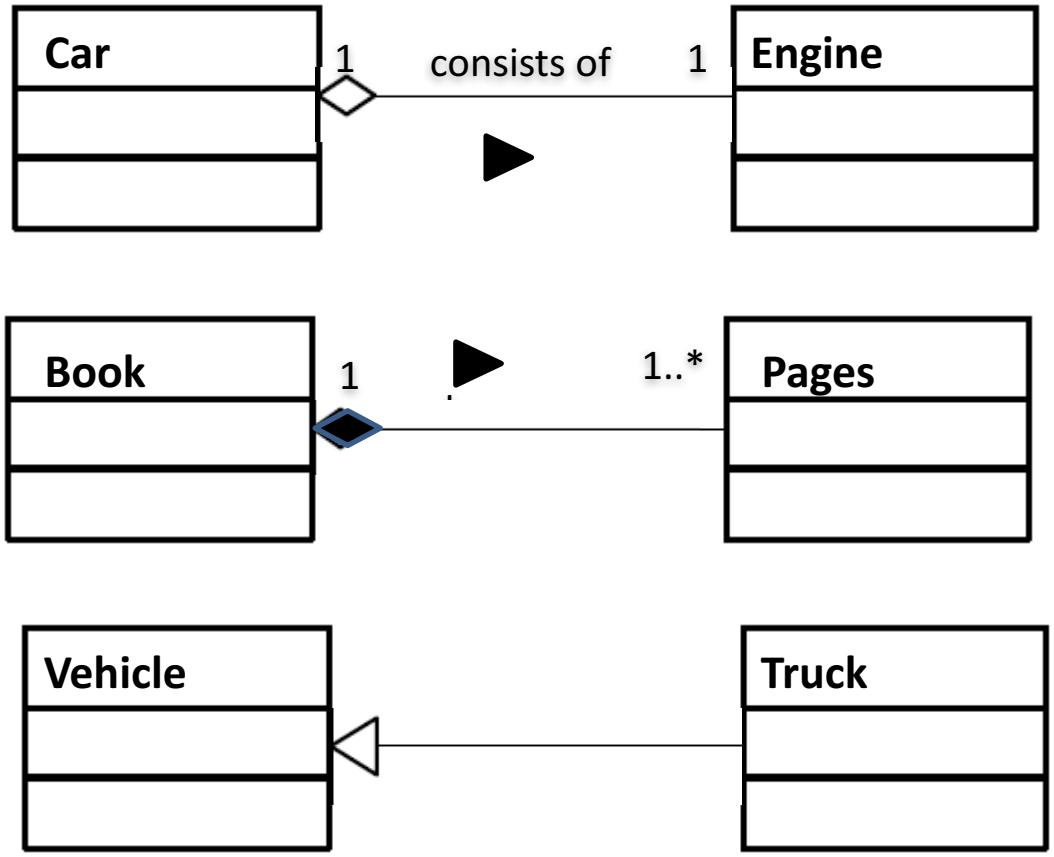
Relationships – Association

- Associations can model a ***"has-a"*** relationship where one class "contains" another class
- Associations can further be refined as:

Aggregation relationship (hollow diamond symbol ◊): The contained item is an element of a collection but it can also exist on its own, e.g., a lecturer in a university or a student at a university

Composition relationship (filled diamond symbol ♦ in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk, or engine in a car

More examples of associations



- Aggregation - “has-a” relationship where the part can exist without container
- Composition – “is-composed-of” relationship where part cannot live without container
- Inheritance – “is-a-kind-of” relationship

COMP 1531 Recap

Requirements Engineering and Domain Modelling

- In COMP 1531 you were taught how to:
  - Perform requirements engineering (requirements elicitation, analysis, specification and validation)
  - Develop a use-case diagram to describe the functionality of the system-to-be (“what features”)
  - Develop a domain model to implement the system (“how to deliver the features”)

Requirements Analysis vs Domain modelling

- Requirements analysis determines " external behaviour "
  "What are the features of the system-to-be and who requires these features (actors) "
- Domain modelling determines (internal behavior) - "how elements of system-to-be interact to produce the external behaviour"
- Requirements analysis and domain modelling are mutually dependent - domain modelling supports clarification of requirements, whereas requirements help building up the model.

Domain model

- Also referred to as a conceptual model or domain object model

- Provides a visual representation of the problem domain, through decomposing the domain into key concepts or objects in the real-world and identifying the relationships between these objects

- Techniques to build a domain model
  - Noun/Verb Analysis
  - CRC Cards

Noun/Verb Phrase Analysis

- Analyze textual description of the domain to identify **noun** phrases
- Caveats: Textual descriptions in natural languages are ambiguous (different nouns can refer to the same thing and the same noun can mean multiple things

Consider this text about an ATM machine:

A customer arrives at an ATM machine to withdraw money. The customer enters the card into the ATM machine. Customer enters the PIN. The ATM verifies whether the customer's card number and PIN are correct. Customer withdraws money from the account. The ATM machine records and updates the transaction.

Candidate conceptual classes: ATM, Customer, Account, Card

Using CRC cards

- CRC stands for:
  - ***Class*** : Represents a collection of similar objects
  - ***Responsibility*** : Something that the class *knows* or *does*
  - ***Collaborator*** : Another class that a class must interact with to fulfil its responsibilities
- Written in 4 by 6 index cards, an individual CRC card use to represent a domain object
- Featured prominently as a design technique in XP programming

Student	
Enrols in a Course-Offering *Knows* Name *Knows* Address *Knows* Phone Number	Course-Offering

Case Study

A Domain Model for an Enrolment System

Enrolment System

- Students enrol in courses that are offered in particular semesters
- Students receive grades (pass, fail, etc.) for courses in particular semesters
- Courses may have prerequisites (other courses) and must have credit point values
- Course offerings are broken down into multiple sessions (lectures, tutorials and labs)
- Sessions in a course offering for a particular semester have an allocated room and timeslot
- If a student enrols in a course, s/he must also enrol in some sessions of that course

Use Case (UC1): Enrol a student in a course

Flow of events for usage success scenario:

1. **Student** requests **EnrolSys** to *display list of courses* for the current **semester**
2. Student selects a **course**
3. Student request EnrolSys to *enrol* in the course
4. EnrolSys *check* if student meets **pre-requisite course** (satisfied)
5. EnrolSys checks available **sessions** (lectures, tut/lab)
6. EnrolSys ***create*** an **Enrolment** and provide details to the Student

Use Case: Enrol a student in a course

Flow of events for usage success scenario

1. System shows list of courses
2. User selects a course
3. User asks system to enrol in course
4. System checks pre-requisite of course (satisfied)
5. System allocates sessions to a user
6. System displays enrolment details to user

Walkthrough

1. EnrolSys displays list of Course (CourseOffering)
2. User requests EnrolSys to enrol in a particular **CourseOffering**
3. EnrolSys asks **Course** for the relevant pre-requisites
4. EnrolSys checks if Student passes prereq(satisfied)
5. EnrolSys allocates **Session** to the Student
6. EnrolSys creates an **Enrolment** and provides **Enrolment** details to Student

UML Class Diagram

- Association
- Aggregation
- Composition
- Inheritance (extends)
- Realization (implements an interface) (coming up …)

Useful Resources

UML Sequence Diagram: *https://www.ibm.com/developerworks/rational/library/3101.html*

Importing existing project into Eclipse: *https://webcms3.cse.unsw.edu.au/COMP2511/18s2/resources/20186*

Java Style Guide: https://slack-redir.net/link?url=https%3A%2F%2Fwww.oracle.com%2Ftechnetwork%2Fjava%2Fcodeconventions-150003.pdf