CSCU9T4: Managing Information

The Module

Module co-ordinator: Dr Jingpeng Li

Lectures by:

- Dr Jingpeng Li (jli@cs.stir.ac.uk), Dr Nadarajen Veerapen (nve@cs.stir.ac.uk), and Dr Amjad Ullah (aul@cs.stir.ac.uk)

Two parts:

- Java, OO design, file and string handling, Data security
- XML Technologies

Student representative

Interested? Give in your name via Succeed.

CSCU9T4: Object Modelling, principles of OO design and implementation

Aims

The purpose of this section of CSCU9T4 is to consolidate your understanding of

- objects and classes
- why objects are important/useful
- OO features and their implementation in Java :
  - interfaces,
  - inheritance, polymorphism,
  - dynamic binding, overloading, redefinition

And to introduce the Collections framework

What is an object?

What are they for?
What are their features?
What is the relationship between objects and classes?

Object-Orientation in Java

Java is an object oriented language
as is Swift, Objective C, C++, and C#

The text of an object-oriented program consists of a set of *class definitions*

A *class* defines:

- The *attributes* (variables, data) of an *object* (usually *private*),
- The public *operations* (methods) "offered" by the object – for call from other objects,
- The *behaviour* of the object when one of its operations is called,
- Any supporting *private methods*

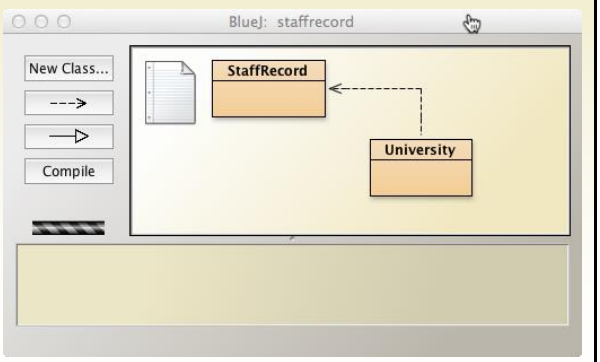
Object-Orientation in Java

An object is an instance of a class

- A class is like a *template*, and an *object* is like a copy (instantiation) of the template
- Created using Java's **new** keyword, eg:

```
StaffRecord sr = new StaffRecord("Simon
  Jones");
```

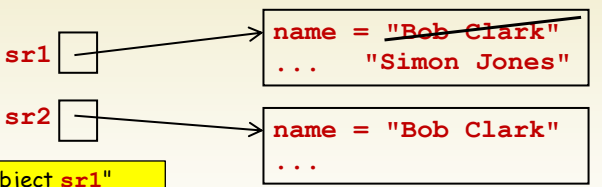
At run-time, a Java program consists of a group of communicating objects which are created from a set of class definitions



What is the physical identity of an object?

Example:

```
StaffRecord sr1 = new StaffRecord("Bob Clark");
StaffRecord sr2 = new StaffRecord("Bob Clark");
sr1.setName("Simon Jones");
sr1 = sr2;  ???
```



Note: "the object **sr1**" means "the object referred to by variable **sr1**"

Note: the *references* in **sr1**, **sr2** are typically the *RAM addresses* of the objects they *refer* to

What happened here?

About *constructors*:

In order to be able to have:

```
new StaffRecord("Bob Clark")
```

Note: reminder about `this`

The class `StaffRecord` must have a *constructor*:

```
public StaffRecord(String name) {
    this.name = name;
}
```


What do Constructors do?

Reminder: When a **new** instance of a class is constructed:

1. Fresh memory (RAM) is allocated (by the JVM itself)
2. The constructor is called
3. A reference to the new object (in RAM) is returned

A class may have:

- No explicit constructor (there is an implicit "no-args" constructor)
- One constructor (could be "no-args" or have parameters)
- *Several* constructors – *overloading*
  Examples: `JButton()`, `JButton(Icon i)`, `JButton(String text)`

Important Design Features (at the Class Level)

Cohesion

a class should represent a single concept

Coupling

the degree of dependency between classes

Consistency

in naming conventions, and use of parameters

Design in the Large

A structured approach is needed for large systems

Building large systems involves extensive group work

- A common understanding of the task and proper communication is essential
- Each member of the group needs to understand their task and how it *interfaces* with other tasks
- A method for dividing up the problem into manageable portions is important
- Groups and individuals need to communicate with a commonly agreed *design language*

This is Software Engineering

Software Engineering & Object Modelling

Software Engineering involves modelling a problem

- Problems with many components are usually complex
- Complexity can often be dealt with through simplification of a system into its main component parts - a model

> "A model is an abstract representation of a system that enables us to answer questions about the system"
>
> Bruegge & Dutoit, p6

> "All models are wrong, some are useful"
>
> G.E.P. Box

- Models allow us to visualise and prototype a system

Software Engineering and Object Modelling

Building a model involves

- The Real World, A problem domain, A solution domain

The Real World

- Environment within which the system must operate
- For example, trading rules within a stock-trading system

The Problem Domain

- Concepts from the real-world that are *relevant* to the users requirements, e.g. stock components, traders

The Solution Domain

- Objects from the problem domain
- Objects that enable problem domain objects to interact

SE & Problem Solving

Software Engineering is about Problem Solving

- Formulate the problem
- Analyse the problem
- Search for solutions
- Decide on appropriate solution
- Specify the solution

There is no standard algorithm to follow…

- You often have to try a number of paths (solutions) and then select the one that appears most appropriate
- Previous experience is very useful in selecting a solution
- You can also benefit from other peoples' experience (more later)

SE & Knowledge Acquisition

As you design a system, you acquire more information

- You will very rarely have all the facts at the start
- New knowledge does not always add to prior knowledge
- Sometimes newly acquired information may invalidate your previous ideas
- You need to carefully manage the knowledge acquisition process to ensure you do not miss out crucial information

Good design will help avoid these pitfalls

- e.g. Build flexibility into your design
  - » Keep component dependencies to a minimum
  - » If a major design revision is necessary due to new information, minimal dependencies will reduce the need to start again from scratch.

Typical organisation of SE lifecycle

Software Engineering - steps in the software development life cycle

- Requirements (elicitation)
- Analysis (of the problem)
- Design (of the solution)
- Implementation (of the design in code)
- Testing (the code & design)
- Maintenance

Design

Analysis of the problem indicates what the major components in the system are, it will not tell us how these components work.

Design involves

- Identification of major component boundaries
- Decomposition of the major components into smaller semi-independent sub-systems
- Design of the interfaces between these major components & sub-systems

Design

Design involves (continued)

- Identification of new components necessary to bridge the gap between objects in the problem domain and the solution domain.
- Identifying if any of these components are potentially re-usable or are available 'off the shelf'
- Flow of control within the system
- Flow of data within the system
- Data management (e.g. central repository or distributed information)

Software Engineering Methods – Object Modelling

We would like a method that allowed us to model our problem and solution at an abstract level

- Abstract model allows early prototyping
- Test various scenarios

The method should support the concept of components, sub-systems and interfaces between components

It is useful if the approach we adopt was supported by the programming language(s) we will use to implement the solution

- The abstract model could then be more easily converted to a lower level implementation whilst maintaining the core design features

Object Modelling

What has all this got to do with object modelling?

- Object modelling is one method of approaching the software development process
- Requirements analysis remains the same
- Emphasis is on the Analysis and Design phases of the software development life cycle
  - » Analysis & Design are concerned with modelling the problem and modelling the solution
- Supports the principle of early prototyping to improve the Requirements and Problem Analysis phases
- Objects identified in the design phase are readily implemented via an OO language

CSCU9T4: Object Modelling, and a hint of UML (Unified Modelling Language)

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.

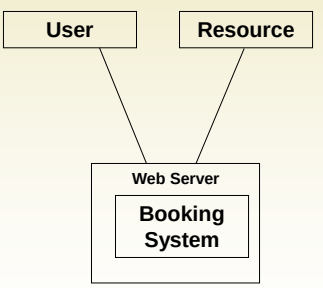
Belief: objects identified when analysing a problem can be used when creating a solution.

This is an over-statement, but is mostly true.

Object Modelling - Analysis

Analysis of the Problem

- Decomposition of elements in the requirements into objects
- We build a diagram of the objects identified in the problem and how they interact



Object Modelling - Design Solution(s)

Design of the Solution

- Further decomposition and creation of new objects to facilitate communication or purely software related features
- There is a transition from our diagram of the problem to our diagram of the solution
- We may initially develop a number of alternative solutions
- This phase is iterative, with continual refinement

The Object Oriented Approach - Implementation

Implementation - Turn each object identified in the design into a class (a template for an object)

- The class encapsulates the data and operations that define the behaviour of an object, for example:
  - » user objects, staff objects, resource objects
  - » a resource object would contain a borrow operation and data on when it was due back
- The class may contain references to other classes - allowing a hierarchy to be built
- We use programming languages that support object oriented concepts - e.g. Java, Objective C, Swift, C++, Ada or C#.
  - » Implementation process is more automated
  - » It is easier to take the high-level design structure and convert it into the equivalent low-level code due to a direct parallel with the OO programming language structure

UML 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

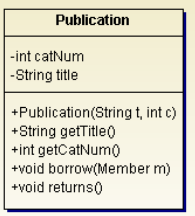
- The main UML structural diagram is the **class diagram**.
- Requirements are collected together using a **use case diagram**.
- Dynamic models are represented using **state diagrams** and **interaction diagrams** (**sequence diagrams** and **collaboration diagrams**)
- and there are more …

Representing a class in UML

Most simply: as a box **Publication**

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

- name
- attributes
- operations



Access to attributes is normally restricted and they are therefore hidden (private). This is represented in UML by the prefix '-'. The operations are visible (public) and so are prefixed with a '+'.

Corresponding outline 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 returns() { ... }
}
```

Note that this outline Java class definition shows *exactly* the same information as in the UML diagram on the previous page.

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.

Object Modelling - Example

Consider a system to keep track of the art collection of the university. This might have a number of purposes:

to allow the collection to be browsed

to allow other galleries to borrow items

…

What are the main entities in the system?

How are they related?

What are the main activities in the system?

The ArtCollection 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

In the class diagram, we use arrows to show that (e.g.):

- **`ArtCollection`** "knows about" **`Artwork`**
- **`ArtWork`** "knows about" **`Artist`**
- **`Artist`** "knows about" **`ArtWork`**

Concretely, an arrow indicates that (e.g.):

- A **`ArtCollection`** object has an attribute referencing an **`ArtWork`** object
  - and can send messages *to* that object

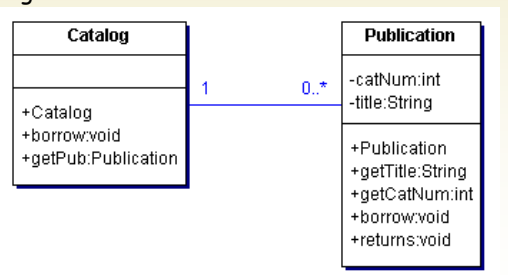
Associations

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

- Indications that they "work together" in some way.

For 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:



Aside - UML notation

Boxes for classes.
Divided into three sections: class names, attributes, operations.
Types of attributes: public (+), private (-), protected (#)
Associations between classes: lines between the boxes.
These can have additional features:

arrowheads (more on this later)

multiplicity:

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

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 implicit attribute)

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

- At some point we must decide whether `Catalog`s "know about" `Publication`s, and/or `Publication`s "know about" `Catalog`s
- 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*

Implementation of Associations

Navigability is represented in a class diagram by adding one or two arrows to the line representing an association.

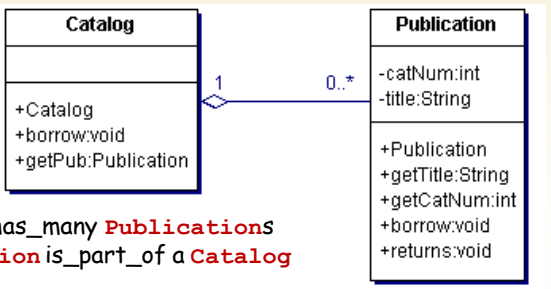
- We have the meaning that if no arrow is present then we are saying nothing (yet) about navigability
- If one arrow is present then we are saying that the association is navigable only in that direction

Navigability suggests in which class an implementing attribute should be located

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 write:



A **`Catalog`** has_many **`Publication`**s
A **`Publication`** is_part_of a **`Catalog`**

Class diagram vs executing program

The class diagram shows us a static view of the responsibilities and relationships of the individual classes

- This helps us understand the Java program

Classes exist in the program text.

*We compile the program text to produce an **executable program** that we can execute (**run**).*

The executing program comprises a collection of objects, actual instances of the classes, that interact by sending messages

- In general each class may give rise to any number of distinct objects
- At ***run-time***, ***instances*** of classes are created using the keyword **new**.

Objects exist in the running program.

Classes and objects

The JVM launches the program by calling the **`main`** method

- This typically creates an *instance* of a **`Boundary`** class

```
Boundary app = new Boundary();
```

- Typically the **`Boundary`** constructor then builds a GUI (**`JFrame`**)

The interactions between the user and the program are represented in **`Boundary`** as a set of operations: e.g. **`mkArtist, mkArtwork, mkGallery`** …

- These are called from within **`actionPerformed`** method, say, and create further objects, send messages, etc as necessary

Object-Oriented Programs

Each object has a separate identity

- its attributes might change, but it remains the same object.
- two objects may have the same set of attribute values, but be two distinct objects, for example you could assign the value "Hello" to two different String objects.

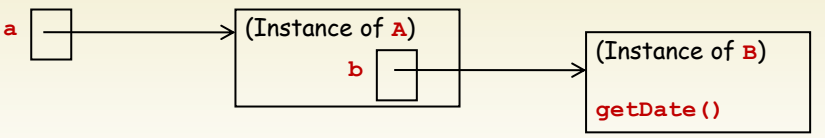
A **client** object **A** communicates with a **server** object **B** by calling a publicly available *operation* offered by object **B**.

In languages such as Java, we usually talk about methods rather than operations.

Consider an object **a** (an instance of class **A**) and an object **b** (an instance of class **B**)

A **client** object **a** communicates with a **server** or **supplier** object **b** by calling an *operation* "offered" by object **b**

- We often think of **a** as sending a message to **b**
- **a** may be sending information to **b**, requesting information from **b**, requesting that **b** carry out some action, or a combination
- Client object **a** must contain a reference to the supplier object **b**



The client **a** has a task it needs to perform e.g. calculate days expired:

```
Date d = b.getDate();
```

where **a** has previously created an instance of **b**:

```
B b = new B(...);
```

The supplier **b** provides a service that helps **a** with its task e.g. get the current date:

```
public Date getDate() {…}
```


Summary

The **class diagram** shows the **structure** of the Java program

- The *possible* run time references and communication paths

We can understand a lot about how a Java program works by examining its class diagram

- In particular the connections ("associations") between the classes that indicate how messages are passed to obtain results "collaboratively"

End of lecture