Worksheet 6: Introduction to Object Orientation

Updated: 10th April, 2021

The objectives of this practical are to:

- design the basics of model class concepts, including class fields;
- design the basics of accessors and mutators;
- understand the basics of how objects relate to real-life.

1. Talking about basic model classes

Before we start designing and implementing them, let's talk about basic model classes.

Together as a class, spend twenty minutes discussing these questions. You can work together for this part of the worksheet. The idea of this exercise is to get you actively thinking about the problems you are about to face. The answers to these questions will also help you design your classes.

Each question also has several sub points that your discussion should address.

(a) What is the difference between a class and an object?

(b) Why do we need accessors?

  (i) What relationship do accessors have with class fields?

  (ii) What class fields would you need for **Student** class?

(c) Why do we need mutators?

  (i) How are mutators related to class fields?

  (ii) Why do you **not** need a mutator that changes all class fields at once?

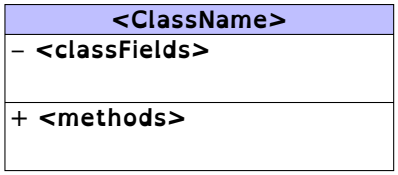
(d) Why **might** we use submodules for validation?

(e) Why should model classes not have **INPUT**s and **OUTPUT**s?

2. Defining a class using UML

Before we design a class using pseudocode, we must first determine the properties of which the class will consist of – such as the class fields it contains and the methods which will utilise them. The unified modelling language, commonly known by its abbreviation UML, provides a method to describe the components of a class without specifying the logic underlying it. This is done in a diagrammatic form, as seen below. In later studies, we will look at relationships between classes, which UML allows us to represent.

Defining a class in UML, using a diagram as seen below, is the first step to creating a software solution to a problem. It is the link between a **problem statement** outlining what the software will need to do and the design of the solution using pseudocode.



The negative sign (–) before an element – a class field or method – indicates that it is **private**. This means it can only been 'seen' and hence accessed or mutated (changed) by the current class.

The positive sign (+) before an element – a class field of method – indicates that it is **public**. This means it can be 'seen' and hence accessed or mutated (changed) by any class.

Consider how classes encapsulate methods and data before you decide which elements will be public and which will be private. In general for model classes, the class fields will be private and the methods will be public. Also note the capitalisation, following the standards of this unit so far for existing variable, class and method names.

Each method should have its signature fully defined – that is, its name, imports and exports, similar to the first line of a function definition in Java. The format for doing so is as follows:

`<methodName>(<importParameters>):<exportDataType>`

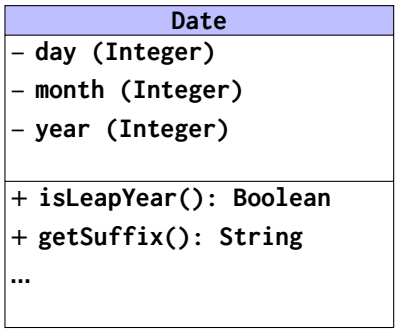
In some cases, it may make more sense to describe the name of the export rather than its datatype. Some class methods – like any methods – may not have imports and/or exports.

A mutator method ('setter') that validates (if required) and changes the value for each class field to a value supplied to the method is implied by the diagram, as is the presence of an accessor method ('getter') for each class field used to access its value.

Ensure you are familiar with the above UML diagram before continuing, as you will be required to design and implement a class based upon a UML diagram.

3. Creating the Date class

The first class you should design is a **Date** class. The complete class should be implemented as defined using UML below. You should maintain the order of the class fields in their declarations and the order in which mutators (setters) are defined.



Let's think about how each class field in this class can be validated:

- **day**: how do we calculate whether the day is valid? First, we need to know the month, as each month has a different number of days. We could use an **IF** or **CASE** statement as we did in earlier practicals. Then, we also need to check to see if it is a leap year or not.
- **month**: what constitutes a valid month? This is relatively straightforward, as it does not change from year to year.
- **year**: how do we validate the year? Are any values that this class field could store considered invalid?

Firstly, identify any class methods which are not defined in the UML diagram above. Using pen and paper or software, draw a UML diagram containing these methods alongside the class fields and methods detailed above.

Next, it is time to begin writing pseudocode by first specifying the class name and class fields in Pseudocode. Use the examples provided in the lecture as a format guide.

Then, complete the pseudocode for the accessors ('getters') and mutators ('setters'), as per the examples in the lecture slides. Do not forget to develop validation code for the class fields; this can be achieved within the functions that require it directly or by calling another function. Once you have done this, develop the pseudocode for the remaining functions (**isLeapYear** and **getSuffix**).

There are multiple solutions to this problem, so do not worry if you think there is more than one way to solve it.

4. Modelling the World

In the lecture, it was explained that real-life objects can be modelled in the object-oriented programming paradigm. Select three items in the room you are currently in and first identify what they are and their attributes.

Once you have identified three objects and their attributes, create a UML class diagram for each, ensuring you note the class name, class fields (including their data types) and methods.

Finally, for each of your UML class diagrams, develop a full set of pseudocode to describe the class, as we have learnt up to and including Lecture **P06**and Worksheet **P06**. Ensure that you validate your data where it is appropriate when you design your mutators.

End of Worksheet