Lab 2: Interfaces and Classes

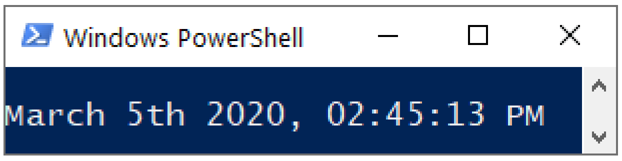
Aim

This lab class gives you an opportunity to:

- create a Java interface from a UML diagram;
- gain some experience with multiple ADTs which have been specified by an interface and implemented by a class;
- implement a class that implements an interface and also extends another class (an *is-a* relationship);
- implement a class that implements an interface and uses composition of another class (a *has-a* relationship); and
- practise your Java programming.

Context

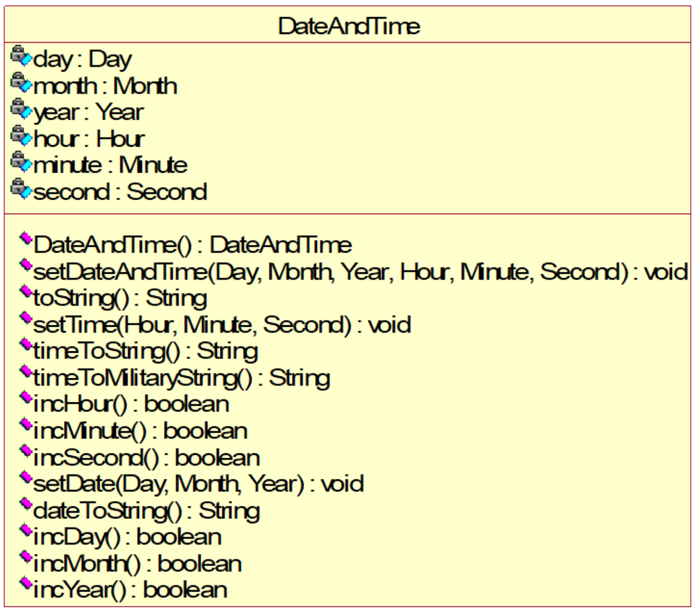
A simple desktop clock application could be written to look like this:



```
March 5th 2020, 02:45:13 PM
```

To implement this ourselves (and to use it as an opportunity to learn about ADTs) we'd need an algorithm which gets the current time from the computer, stores it within a model, and then repeatedly displays the values, waits for a second, and increments the value.

Date and time can be modelled using an ADT as follows:



```
DateAndTime
day : Day
month : Month
year : Year
hour : Hour
minute : Minute
second : Second
DateAndTime() : DateAndTime
setDateAndTime(Day, Month, Year, Hour, Minute, Second) : void
toString() : String
setTime(Hour, Minute, Second) : void
timeToString() : String
timeToMilitaryString() : String
incHour() : boolean
incMinute() : boolean
incSecond() : boolean
setDate(Day, Month, Year) : void
dateToString() : String
incDay() : boolean
incMonth() : boolean
incYear() : boolean
```

There are some restrictions on the way these operations work:

- years must be between 1 and 3000 only. Incrementing a year beyond 3000 should result in a year value of 1;
- hours (0–23), minutes (0–59), seconds (0–59), days (1–28/29/30/31 depending upon the month and year), months (1–12), and years (1–3000) should all be `int`s;
- normal time display should be of the form: `HH:MM:SS AM` (or `PM`);
- military time display should be of the form: `HHMMSS`;
- normal date display should be, e.g.: `3rd July 2006`;
- newly created dates should default to 1/1/3000, 000000 hours;
- the `setTime()` and `setDate()` routines should perform error checking (throwing a `NumberFormatException` exception if a component is illegal); and
- it should be possible to increment the second/minute/hour/day/month/year value by one (using the `incSecond()/incHour()/incDay()/incMonth()/incYear()` methods). Each of these methods should add one to the relevant instance variable. If the upper limit is reached (e.g. if the seconds becomes 60) they should be reset to the lower limit (e.g. 0) and the next component (e.g. minutes) should be incremented and its return value should be returned. If the upper limit is not reached, `false` should be returned. In this way, incrementing the second should eventually lead to incrementing the minute which should eventually lead to incrementing the hour, then the day, then the month, and finally the year.

Approach
We could develop all of the `DateAndTime` ADT from scratch. Or, we could argue that `DateAndTime` is a special kind of `Time` in which we're adding those things which it doesn't have (i.e. all of the `Date` related properties and methods) and get the rest (i.e. all of the `Time` related properties and methods) through inheritance. We'd do this by saying

```
public class DateAndTime extends Time implements DateAndTimeInterface
```

But, given that C isn't object-oriented and doesn't have inheritance, we'll do it using aggregation. I.e. `DateAndTime` *has-a* `Time` rather than `DateAndTime` *is-a* `Time`.

```
public class DateAndTime implements DateAndTimeInterface
{
     protected Time theTime;
```

As a starting point, recall that `Time` has been discussed and implemented in lectures. That implementation implements a time value but does not allow incrementing the seconds/minutes/hours. A new class (`AdjustableTime`) could be defined based on the `Time` class in which all instance variables and methods of `Time` would be inherited and in which only the `incSecond()`, `incMinute()`, and `incHour()` methods need to be written. These would simply add one to the appropriate component, resetting it to 0 if overflow occurs and returning `true` if overflow occurs and `false` otherwise. Then, `DateAndTime` could be implemented where the time component could be an `AdjustableTime` object and the time related methods are wrappers for calls to `AdjustableTime` methods. The date component and all other methods would need to be written from scratch.

Tasks

1. The compressed project folder should be obtained from MyLO and all contents extracted to your home directory (`P:`). Open all the files in *DrJava*.
2. Create a new Java interface file (with name `DateAndTimeInterface`) within the folder and convert the above UML diagram into a Java interface within this file.
3. Inspect the `TimeInterface` interface and `Time` class files (which are taken from lectures). Inspect the `AdjustableTimeInterface` interface file.
4. Create a new Java library class file (with name `AdjustableTime`) and complete the class — by extending `Time` and implementing `AdjustableTimeInterface`.
5. Examine the harness class `Lab2` if desired. It uses some classes which are beyond the scope of this unit, but notice that at its heart it: creates a `DateAndTime` object with the current date and time, and then repeatedly displays the values, waits for a second, and increments the value by a second.
6. Compile the whole project.
7. In your folder view in Window's *File Explorer*, hold down the <Shift> key and right-click to call up a pop-up menu. Choose "Open PowerShell window here". Run the program by typing **java Lab2**. If you resize the window you have a rudimentary desktop clock!

Note: you should compile each file as you write it and remove all compiler errors.