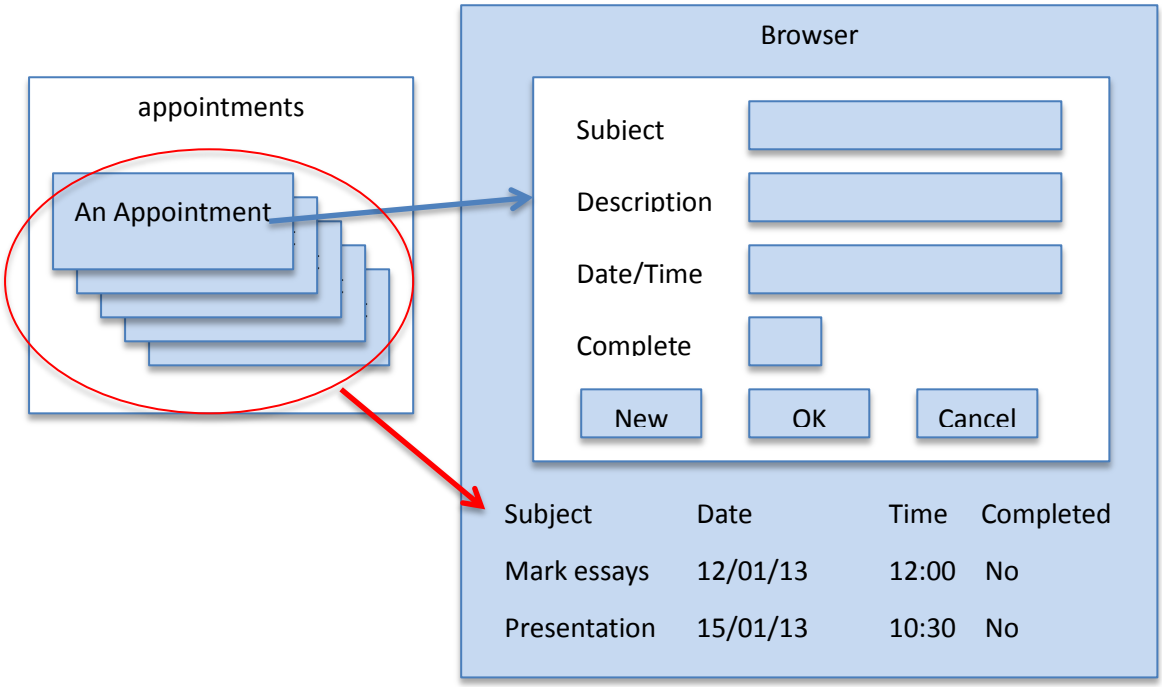
The Appointments App Deconstructed

Over the series of labs in this module (if you've been keeping up), you've created an appointments application – nothing to rival the calendar in outlook, but a basic application that lets the user set up a diary of appointments and manipulate the information in them to get some useful(ish) behaviour. Of course, this application was not developed for any purpose other than to clarify some of the stages you would go through in developing an app, so it is a very simple app. My main purpose was to highlight the useful features of object-oriented programming, and you should bear that in mind through the rest of this.

Development Process

I've been fairly unremitting, bordering on nagging, in my description of the process best used to develop this (or any other) application. Build a little at a time; test that before you move on. Objects help here, because you can create a small but functional part of a project in a way that makes it easy to integrate with other parts as the whole app progresses. Pictorially, we could view an Appointment like this:



It is often a good idea to depict a project like this – it provides a common point for discussion that anyone can understand. An end user will easily see if there is information missing (e.g. there is no location field in this diagram so I'll need to add one at some stage). It also makes it easier to think through scenarios for the software (what steps to go through to add a new appointment, edit an existing one, delete an existing one etc.). In technical terms, these scenarios are called "Use-Cases", a term that indicates that the technical details are less important at this stage than simply considering how the system will be used.

Once we've come up with a model of our system (like the diagram above), we're well equipped to be able to say what goes into it. This principle is the same whether we're developing a program in Javascript & HTML5, a database or a paper-card-index based system. From this diagram we can see that:

- The main **entity** type in the project will be an Appointment
- The system should manage a collection of appointments
- It should provide for the user to add a new appointment or edit an existing one (another missing feature – the diagram shows no way to delete one just now)
- The user should be able to see a list of existing appointments. Although there is nothing specific about this in the diagram, discussions of use cases should lead to a more detailed specification where to edit (or delete) an existing appointment, the user would need to **select** it – clicking on the existing appointment is the normal way to do this
- Although the diagram cannot show this, the system should provide warnings (preferably audible and visible) of when an appointment is due.

With this small amount of information, we can start designing the composition of an appointment. Each appointment will have:

- A subject – a short text description of the appointment – e.g. Dentist
- A fuller description – e.g. filling, clean and scale. Note that this will not show up in the on-screen list of appointments, which is simply a summary
- A data and time. Javascript deals with time-related information by storing a single value, which is the combination of date and time. This would be awkward for the user, so a better approach is to prompt the user to enter a separate data and time, and then combine these. Storing a combined data and time will simplify much of the other work – e.g. finding out whether an appointment is due
- A 'completed' indicator. This is a Yes/No value (Javascript has a true/false Boolean type that will suit this)
- Some further analysis (along with my desire to incorporate some way of explaining how to incorporate a mapping service into the app) suggests that it might be a good idea to include a location field – a UK postcode will do. This would give the user the facility of following a map to the place where an appointment is to be kept – useful if it is in an area the user is unfamiliar with

In addition to this, we need a way to manage multiple appointments. The easiest solution in Javascript is an array, which is just a list of objects. Once we have an array, we'll need to provide mechanisms to view all of it, select one of them for edit/delete, update the view (for when an appointment has been added, deleted or changed), and some way of showing the location (to show-off the mapping interface – basically I'm a nerd and think stuff like this is interesting).

Basically, we've just analysed the requirements, come up with some suggestions for the implementation, and now we can carry on with the first stage of the implementation. Our appointment type, and the collection of appointments can now be coded:

```
var Appointment = function (subject, description, date, time, location) {
        this.subject = subject;
        this.description = description;
        this.datetime = new Date(date + " " +  time);
        this.location = location;
        this.completed = false;
};

var appointments = [];
```

Explanation of the code

There are two parts to the above code. The Appointment definition describes a new **type**. It is a constructor function and by convention, the only time in Javascript when you start an identifier (name) with a capital. The capital in the name does nothing – it is purely there because Javascript programmers follow the convention. However, the function itself follows a very set format. It takes a number of parameters (the pieces of data that makes up an appointment) and assignes each of these to a **this** variable (aka a **member variable**). The sequence **this.<variable name>** indicates that the new object this function creates will contain an internal variable with that name. In use, a constructor function is called using the following syntax:

```
var a = new Appointment("Dentist", "Clean, scale and a filling", "2013-01-12", "10:30");
```

The new keyword does the job of allocating some storage space for the appointment data, and then executing the constructor to assign the values to it.

The second part is the list of appointments. **var appointments** = [] creates an empty array called appointments (again, the convention is that the array/collection name should be a plural of the type name (Appointment) but without the capitalization. These conventions are not enforced by the language, but make life easier for other Javascript programmers reading your code. The purpose of this array is to keep hold of any number of Appointment objects. One can be added to the array using the **.push()** method, as in:

```
var a = new Appointment("Dentist", "Clean, scale and a filling", "2013-01-12", "10:30");
appointments.push(a);
```

Does it work

The good thing about working in Javascript is that it takes very few lines of code to make something that does a job. The down-side of this is that it is all too easy to have mistakes in the code – errors that a Java, C++ or C# compiler would have picked up on can go unnoticed. Because of this (and also because it is best practice, even for Java programmers) is to test code as soon as you have enough of it to test. In this case, we can call the constructor and see if it gives us back an Appointment object, and we can try adding an appointment to the array to see if it will store appointments for us. You can do this testing without writing another line of Javascript code – simply open the project in a browser, open a console window and enter some testing statements: