# Object Oriented Modelling and Application Models

This chapter describes how Javascript code relates to the environment it runs in; usually this is the browser and Javascript code has TWO object structures to content with; the browser itself (via the Browser Object Model, or BOM) and the mark-up of the current page, called the Document Object Model or DOM. A developer who wishes to control the environment their app runs in needs to access the properties and methods of both of these object models, as well as devising their own object model (an Application Object Model, or AOM) to manage the specific application features.

Object Model Structure

So what is an Object Model? The term describes the objects in a program, the way that the objects are interconnected, and takes into account inheritance between object types, objects that have other objects as properties (composition), collections of objects (aggregation) and objects which call on the methods of other objects (interactions). An object model is simply the way a program is built from objects taking these four types of interconnection (inheritance, composition, aggregation and interaction) into account. For example, a graphics program will be made up of several components:

- The user-interface, which is the set of objects that the user interacts with to make the program do things. This will usually be an arrangement of menus or a toolbar containing a set of tools

- The workspace, which is where items are drawn – this will interact with the toolbar so that objects can be positioned, manipulated and deleted. It will also be required to initiate the rendering all of the elements in a drawing, and to interact with specific tools

- The drawing itself, which will be a single object made up of all of the drawn elements (e.g. circles, lines, rectangles, polygons etc.)

- Some management objects, which are responsible for saving drawings into files, restoring them from files, managing copy and paste from other programs etc.

Overall, we could draw a diagram to depict the whole model:
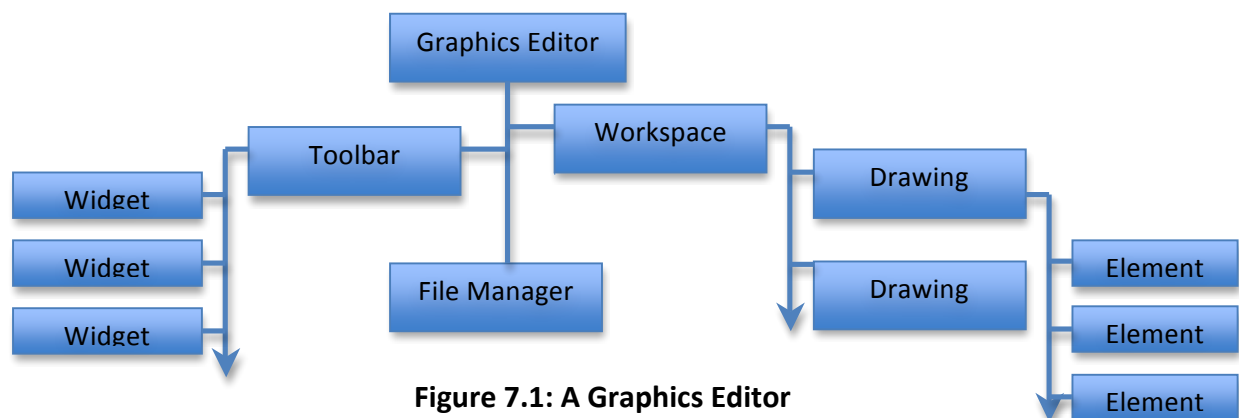
Figure 7.1: A Graphics Editor

Note that the figure 7.1 shows how objects are composed (e.g. the main components of the Graphics editor are a Workspace, a Toolbar and a File Manager). This does not take into account inheritance (where, for example, all of the different Element types would inherit from a common object type) or object-to-object interactions (e.g. the File Manager would access the Drawing element to save it to a file). It does however show composition and aggregation relationships. Generally, inheritance and object interactions are more detailed aspects of software design:

inheritance lets us re-use object designs to reduce the amount of work and also to support polymorphism; object-interactions are ways of sharing the work required to perform a task.

So, object modelling is a form of software design, where the aim is to identify the major components of a system so that the overall software design can be broken up into smaller components. This has always been a strong principle in software development, and has become easier since the emergence of object-oriented methods such as those supported by Javascript.
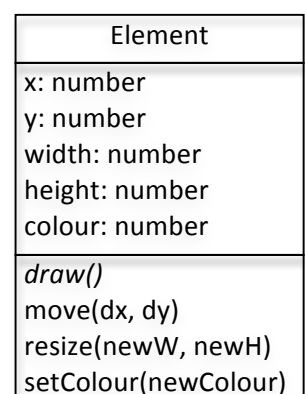
Implementing Object Model Components: Properties and Methods

An object model lets us consider what major components we will need to implement to build a software system, without much consideration of the details. For example, we already know the main components of our Graphics Editor, but have yet to describe what the code for a Drawing object will look like. This is a good thing, because homing in on the details too early in a design would be likely to push us into making decisions that could make things more awkward later in the development. However, we can be clear at this stage that when we do need to get into the detailed form of an object, we will want to consider it from the perspective of what the responsibilities of each object are. In general, that is the information that each object will encapsulate (its properties) and the operations each object will need to perform (its methods).

In lots of situations, an object's properties and methods will be obvious from what its job is. For example, in a drawing, each drawn shape (we'll call it an Element) will need to have a location (a screen position), a colour, a size (width & height) and possibly a drawing style (e.g. solid or dotted line). It will also need to implement at least a draw() method, a move() method (so we can move it to a different location), a setColour() method and a resize() method. I may have missed some crucial methods here, but the details of these would emerge as the design was refined. In some situations, we may not have recognised the need for a property or method until we started implementing the object code. For now, the Element type might look something like this (listing 7.1):

```
function Element(x, y, w, h) {      // Constructor
  this.x = x;        this.y = y;
  this.width = w;  this.height = h;
  this.colour = "#000000";           // initially black
  this.draw = null;          // we'll need this later.
}
Element.prototype.move = function(dx, dy){
  this.x += dx;
  this.y += dy;
};
Element.prototype.resize(newWidth, hewHeight){
  this.width = newWidth;
  this.height = newHeight;
};
Element.prototype.setColour = function(newColour){
  this.colour = newColour;
};
```

Element
x: number
y: number
width: number
height: number
colour: number
draw()
move(dx, dy)
resize(newW, newH)
setColour(newColour)

Figure 7.2: The Element type, shown as a UML **Class Diagram**

Listing 7.1: A drawable Element type

Figure 7.2 shows a useful shorthand for describing objects – a UML class diagram. The three, stacked boxes show *class name*, *properties* and *methods*. While there are no classes in Javascript, the concepts of object design and class design are similar enough that a class diagram works for this.

With this class design, we have defined a grammar for manipulating Element objects. Note that while we've coded up methods for moving, setting the colour of and changing the size of an element, the draw method has not been implemented – instead we've put a **stub** in the object (and shown the method name in the class diagram in *italics*), which indicates that drawing is something we should expect Element objects to be able to do, but not yet – *draw()* is an **abstract method**.

The reason we can do this is that we'll never actually use an Element in a drawing. The purpose of the object design is that it gives us a common set of properties and methods that all types of element ought to be able to do. Element is there to **inherit** from. In object-oriented terms, Element is an **abstract class.** Note that since Javascript does not have classes, this is probably a use of the term that will annoy some object-oriented purists. However, if we intend to use inheritance to distinguish between how several distinct *types of object* will behave, we are classifying them and I think the terminology is appropriate (so sue me).

If you think back to the description of how we could apply inheritance from chapter 4, you may see that we can now create an actual usable type based on the Element type. For example, to create a Rectangle object type, all we need to do is:

```
function Rectangle(x, y, w, h) {        // constructor
  Element.apply(this, arguments);       // inherit the properties
}
Rectangle.prototype = new Element();   // inherit the methods
Rectangle.prototype.draw = function(context){ // implement the draw method
  context.strokeStyle = this.colour;
  context.strokeRect(this.x, this.y, this.width, this.height);
};
```
Listing 7.2: A type that *inherits* from Element

This is pretty neat: all of the methods and properties defined for the Element class are automatically available for the Rectangle class (by virtue of the call to Element.apply() and the assignment of an Element as a prototype), but we have *re-defined* the draw member to be a function that works this way only for Rectangle objects. Because the Rectangle prototype has inherited the entire Element prototype, we can assume that the relationship "Rectangle Is-A Element" holds, so we could use a Rectangle in code wherever we used an Element with no problems. We could go on to create constructors and associated draw() methods for Circle, Line, Triangle etc., writing only the minimum amount of code needed – essentially the code that described how a specific type differed from the Element type.

Events
Another part of an object design, typically important in the design of interactive systems, is how the object interacts with other objects when something is done to it. For example, if we called a method in an Element to change its colour, we would want the element to be re-drawn in the new colour. However, individual elements should not control the drawing process since that is the responsibility of the Workspace component. When we change an Element, we need to give it some way to **trigger** the Workspace code that draws the entire Drawing. We can use an Event for this.

An event in Javascript is a placeholder for a function that we can attach to an object. Since a function is just another object in Javascript, we can give the Element object design a **_redraw_** member. If no function is assigned to redraw, changing the Element will have no outward effect. If, however, we assign the Workspace object draw() method to redraw, then we can arrange it so that any change to an element that would have a visible effect would call the method attached to the redraw member. The correct terminology for this is that an Element has a redraw **_event_**, and we can attach a method to act as an **_event handler_**. For example, we can add a redraw event to the Element definition created earlier:

```
function Element(x, y, w, h) {   // Constructor
  this.x = x;         this.y = y;
  this.width = w;    this.height = h;
  this.colour = "#000000";        // initially black
  this.draw = null;               // we'll need this in objects that inherit
  this.redraw = null;             // this is for an event handler
}
Element.prototype.draw = function(newX, newY){
  this.x = newX;
  this.y = newY;
};
Element.prototype.move = function(dx, dy){
  this.x += dx;
  this.y += dy;
  if(this.redraw) this.redraw(); //if redraw has been assigned a function
                                 //call the function
};
Element.prototype.resize(newWidth, hewHeight){
  this.width = newWidth;
  this.height = newHeight;
  if(this.redraw) this.redraw();
};
Element.prototype.setColour = function(newColour){
  this.colour = newColour;
  if(this.redraw) this.redraw();
};
```
Listing 5.3: The Element type revised to allow events

What is going on here? Well, lets assume that there is a function defined at the application level called **drawAll**. Whenever this function is called, the Workspace will be cleared and then the Drawing will draw all of its elements. When we create a new element in the program, we will create it like this:

```
function addNewRectangle(x, y, w, h, c){
  var r = new Rectangle(x, y, w, h);
  r.setColour(c);
  r.redraw = drawAll;     // We've assigned drawAll to handle redraw events
  drawing.elements.push(r);// this adds the rectangle to the drawing
}
```
**Listing 5.4: Using the redraw event in an application**

Now, whenever we change the location, size or colour of _any_ Rectangle (we would make the same arrangement for any other class that inherits from Element), the drawAll() function will be called.

The benefit of this organization is that we are not hard-coding into the prototype what happens when a Rectangle is moved etc., but are providing for the possibility that it may perform some behaviour we decide on when the *application* is being coded. Key to this is the way that the redraw event is actually called from the Element code – we first check whether a function has been assigned, and only then do we call it. If I don't add an event handler to an object, it won't matter, but if I do, it will be called automatically if either of `move()`, `resize()` or `setColour()` are called.

There is a lot more that could be said about object modelling, and we'll get into a bit more detail, particularly class design, later in the module. For now, we'll have a look at a couple of object models that will figure in many Javascript projects you'll work on: the Browser Object Model, and the Document Object Model.

The Browser Object Model (BOM)

Every Web Browser is made up of a collection of objects – screens for displaying web pages, toolbars, a history of pages that have been visited, the locations that the browser has connected to etc. To make web browsers scriptable (so that some of their features can be scripted to provide new automated commands), browser vendors provide an object model which acts as a scriptable interface for accessing different parts of the browser. Because the Browser Object Model is part of the browser design, different browser vendors have decided on different ways to design the model and so there is no *standard* BOM. Most current browsers do, however, implement a common subset of objects:
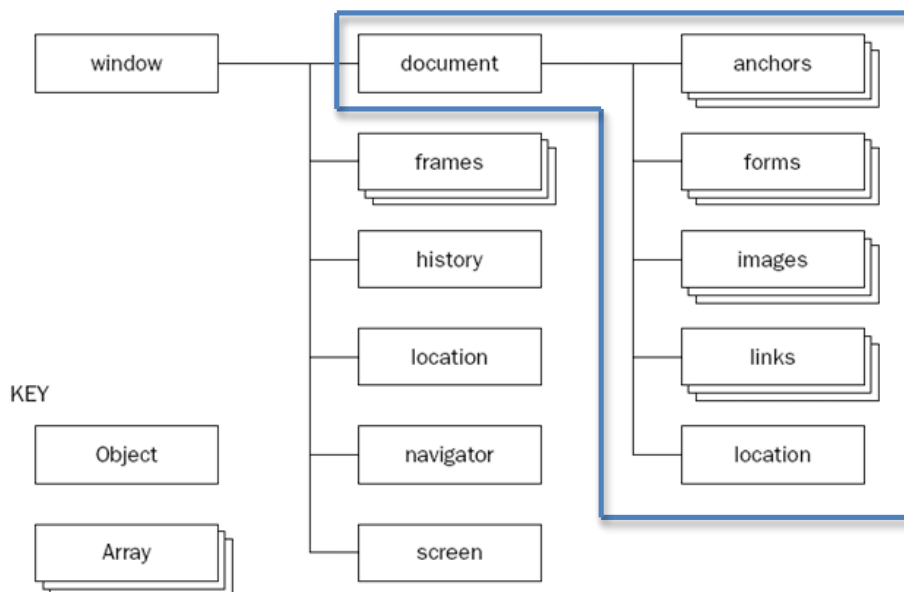
Figure 7.3: The common Browser Object Model

As you can see in figure 7.3, the top of the BOM is the Window object. Each Window has a Document (within the outline), the top of the Document Object Model, which we'll look at later. In addition, a window has a collection of Frames, a History, a current Location, a Navigator object and (access to) a Screen object. These are useful to a programmer for a variety of reasons:

- Frames: the frames collection that belongs to a window is so that a web-page can be organised as a number of separate sub-pages, each of which occupies a rectangular area of the browser's display. Organising a web page (or site) as frames is now not recommended because it makes it hard to bookmark particular parts of the site (the bookmark is always for

the top-level frame), and does not provide good accessibility (e.g. screen readers for blind users can not navigate frames). If you were to ignore this advice, you could manipulate frames in Javascript in the same way that you could manipulate a window with no frames

- History: a browser-window's History is the object that supports the Back and Forward movement to different sites. Each time a new page is navigated to, this page is added to the bottom of the History – a lost of locations. Pressing the Back button, or using the back() method in Javascript would return you to the previous site. Pressing the Forward button or calling the forward() method would return you to the page you just went back from

- Each site in a browser's history is kept in a Location object (so the History object contains a collection of Location objects). Location objects provide methods to reload() the current page, assign() a new page or replace() the current page with a new one. It also provides properties to query the various parts of the URL that leads to a location (host, port, pathname, prototcol etc)

- The navigator object provides data about the actual browser in use, so you can use it in an application to figure out which browser (Firefox, Safari, Chrome etc.), which version, which platform the browser is running in (Windows, Apple Mac etc.), whether cookies and java are enabled or not. This is particularly useful in HTML5 development, since not all browsers provide all of the HTML5 features – if you can detect which browser, you can provide alternative (less slick) methods for performing certain tasks

- The Screen object is useful for figuring out whether there is enough screen space for an application. Its properties include availHeight, availWidth, colorDepth & pixelDepth, all of which are useful if you need to find out whether a graphic or user-interface will display properly in the browser.

There are other components in a typical BOM, but these differ from browser to browser and so it is safer not to use them for a general web page. These features might be useful in an application in a corporate intranet, where it is known which browser everyone is using; otherwise, it is best to ignore them. For the objects described here, W3CSchools provides a good tutorial on their use in Javascript at http://www.w3schools.com/jsref/obj_window.asp. The starting page is the description of the Window object, since from this you can get to any other object in the BOM via a window's properties.

BOM programming is good for controlling how a browser operates from within a web page or application. You can scroll windows, add new sites, manipulate the history etc. – all of which are useful in some circumstances. However, the majority of web page programing in Javascript is done via the Document Object Model, which we'll examine in a little more detail.

The Document Object Model (DOM)

Since the entire purpose of a web browser is to host HTML documents, you might expect that the Document type is important enough to merit its own object model, rather than just rolling it into the BOM. The DOM is what most web developers will spend the majority of their time working with, since it provides ways to manipulate the content of a page and interact with its elements. Also, the composition of the DOM is not left to individual browser manufacturers to define; it is too important for that, since the DOM defines the *content* of a web page, and this must be usable across all browsers. The W3C has produced the definitive design of the DOM (see http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html), and web browser manufacturers are required to make their browsers compatible with this to maintain their compliance.   What you can

take from this is that any DOM coding you learn will be useful, whatever browser it ends up on.

In essence, the DOM has a simple structure, in which the Document element contains a collection of Element elements:
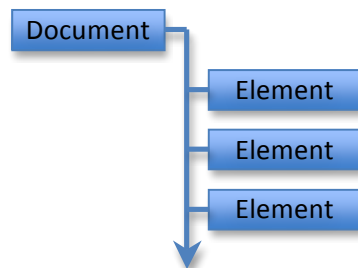
Figure 7.4: A simple view of the HTML DOM

However, this view is simplistic and misses the main feature of the DOM, which is that most elements can actually contain collections of other elements. A more realistic picture is:
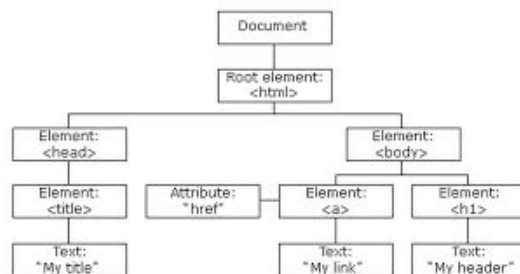
Figure 7.5: A more complex DOM

In the DOM, there are Elements, Attributes and Text nodes, each of which refers to a specific type of HTML content.

- An Element is anything that represents a specific section of a document, such as <p> (a paragraph element), <h1> (a top level header) etc.

- An Attribute is a setting that applies to an Element: for example, an <img> element is useless on its own because the browser needs to know *which* image; this is given by the *src* attribute, so a fully defined image is ******.

- A Text node simply represents the textual content between HTML opening and closing tags: <p>This is a paragraph.</p>. Everything *between* the <p> and </p> tags is a single Text node

Everything in an HTML document that is an Element is a component of the document structure (Attributes and Text nodes define the content and format of a specific Element), and all elements have a type:

- <p>: paragraph element

- <h1>: top level header

- <h2>: second level header

- <table>: a html table, which contains rows and columns of cells (each of which is an element)
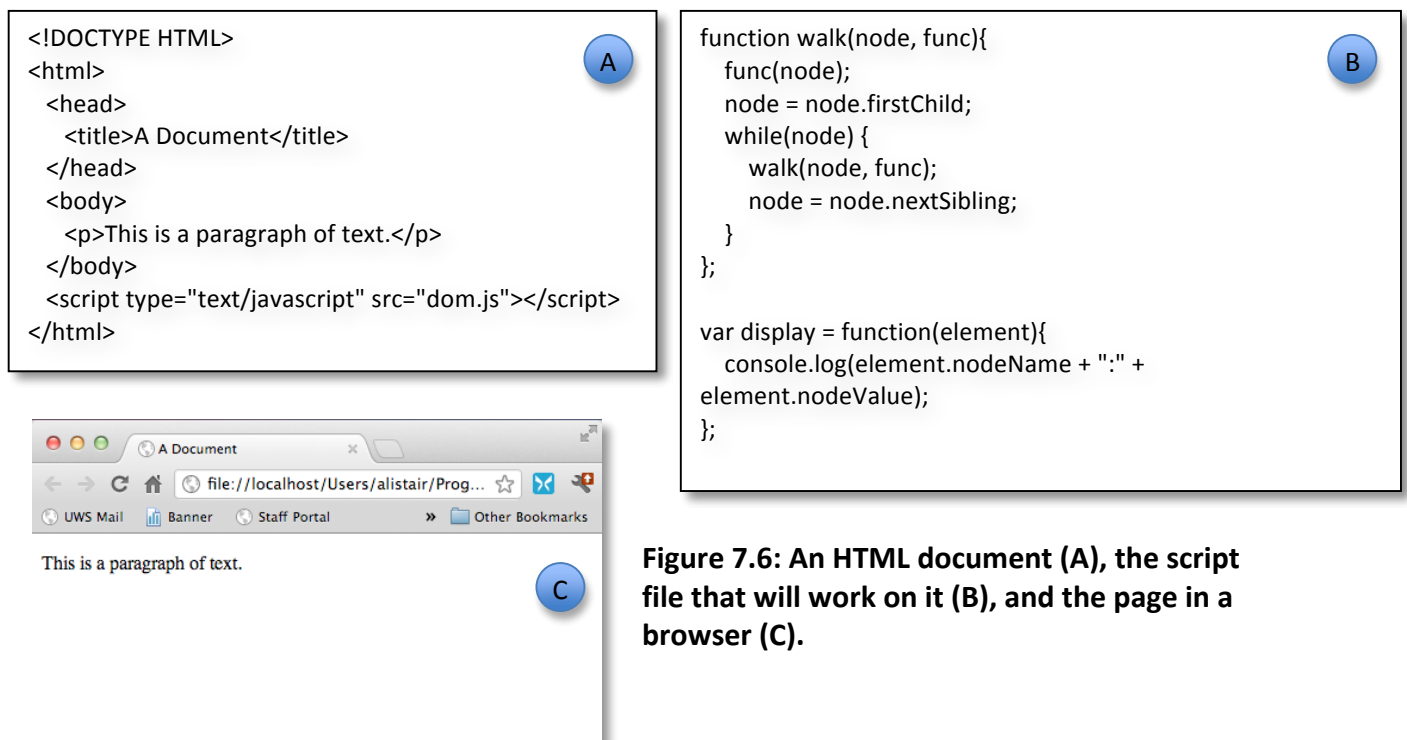
- <img>: a document image etc..

This structure (where everything is an element) is important for the management of HTML documents, since it makes it possible to write code to manipulate elements without getting too bogged down in the differences between the different types. So that this structure works, each HTML Element object must include code that allows its own structure to be interrogated – does this node have child elements?, how many?, what is its parent element? etc. Here is a very simple code fragment that "walks" through an entire HTML document (any one), accessing every element on it:

```
function walk (node, func){ // A node and function are passed in.
  func(node);               // Call the function to work on the node.
  node = node.firstChild;   // Now go to the node's first child element...
  while(node) {             // ...if it exists
    walk(node, func);       // Call THIS function, which will walk the child
    node = node.nextSibling;// Keep doing this for all the node's children
  }
};
```

Listing 7.5: A function to "walk the DOM"

This code is structured to work its way down from a given DOM node, applying the function `func()` to it, and then to all of its children in turn. It does this by going through each of node's child elements, *calling itself* for each of them. This method, where a function calls itself repeatedly, is known as *recursion* and is particularly useful for working through hierarchical structures of data and objects.

Lets look at an example based on a very simple web page:

```
<!DOCTYPE HTML>                                            A
<html>
  <head>
    <title>A Document</title>
  </head>
  <body>
    <p>This is a paragraph of text.</p>
  </body>
  <script type="text/javascript" src="dom.js"></script>
</html>
```

```
function walk(node, func){                                 B
  func(node);
  node = node.firstChild;
  while(node) {
    walk(node, func);
    node = node.nextSibling;
  }
};

var display = function(element){
  console.log(element.nodeName + ":" +
  element.nodeValue);
};
```

Figure 7.6: An HTML document (A), the script file that will work on it (B), and the page in a browser (C).

The HTML document (Figure 7.6 A) is about as simple as it can get. The <html> element contains three child elements - <head>, <body> and <script>. The <head> element contains a <title> element and the <body> contains a single paragraph (<p>). When this document is loaded, figure 7.7 shows what will be printed out on to the console.

```
HTML:null
HEAD:null
#text:
TITLE:null
#text:A Document
#text:
#text:
BODY:null.
#text:
P:null
#text:This is a paragraph of text.
#text:
SCRIPT:null
```

You should note that there are several unexpected TEXT elements in this output: between HEAD and TITLE, between the TITLE text and BODY, between BODY and P, between P and SCRIPT. The only expected text nodes were the ones with actual text ("A Document", and "This is a paragraph of text.").

Figure 7.7: The output from figure 7.6 B.

The worrying thing about this is that there is no simple way to figure out which text elements will be inserted. This is up to the browser's implementation of DOM, and although this is done to a standard, there is obviously some room for interpretation. If you were to add a few blank lines to the HTML text, this should have no visible effect on the mark-up, although it will have an effect on the number of empty TEXT elements reported by the script.

The consequence of this for a web programmer is that it is not a simple matter to step through a web page structure trying to find elements based on their position relative to the whole page. Fortunately, the DOM programmers' interface provides a number of useful functions that can filter elements out of a page to make the process of finding specific elements easier.

Querying the DOM

There are situations where it can be useful to walk through an entire HTML document from start to finish, but in most cases, a programmer is more interested in accessing specific elements. The most important DOM methods for doing this are the getElementByXXX() methods of the document object. These functions let you go straight to a specific element or set of elements; having accessed the elements, you can extract data from them, change them, apply CSS styles or check their content. The functions are:

Method	Description	Example
getElementById():	returns a single element with a given *id* attribute	<table id="t1">…</table> getElementById("t1"); returns the whole table
getElementsByClassName()	returns a list of elements with a class attribute equal to some specific value	<p class="red">Paragraph text…</p> getElementsByClassName("red"); returns a list of elements (any tag type) which have a class attribute equal to "red"
getElementsByName()	returns a list of elements which have a given *name* attribute	<input name="text" value="hello"> <input name="text" value="world"> getElementsByName("text"); will return a list containing both of these elements
getElementsByTagName()	returns all elements which have a given html tag type	getElementsByTagName("p"); returns a list of all of the <p> elements

Table 7.1: HTML query methods

Which of the methods shown in table 7.1 you use will depend on what you wish to do. For example,

if you wanted to add a CSS style to every <p> element in a document, you would use getElementsByTagName('p'); to retrieve a list of all of them. It is easy to miss that of these four functions, only one has a name that starts with **getElement…**. The others start with **getElements…** What to infer from this is that three of the functions return collections of elements (arranged as an array). This is a sensible organization, since in a document, it is likely that many elements can share a Name, ClassName and TagName), but the whole point of an ID attribute it to make an element uniquely identifiable.

jQuery Wrappers

You may have noticed that the four DOM query functions (getElementById(), getElementsByClassName(), getElementsByName() and getElementsbyTagName()) are equivalent to jQuery selectors. getElementByID() does *almost* the same job as $("#id"), getElementsByClassName() is replaced by $(".classname"), getElementsByName() can be replaced by $("[name=elementname]") and getElementsByTagName() is simply replaced by $("tagname"). In fact there is a difference in how these operate in that the jQuery versions always returns an array of jQuery objects, which are *wrappers* for the raw DOM objects.

A jQuery "wrapper" object is a specialized object that **contains** the equivalent DOM object. It also adds behaviour to the equivalent DOM object so that in jQuery we can do shorthand things like $("#myheading").html("A new Heading");. Note that DOM objects do not have an html() method, so we would have to assign a new value to the innerHTML property instead. The jQuery way is neater and, typically, the jQuery version irons out differences between the different browser implementations of DOM objects.

The jQuery wrapper objects add a number of new methods to the DOM objects that they wrap, including:

- html() : a function that can be used to access the current HTML inside an object or to apply a new set of mark-up
- text() : a function that can be used to access or update the text in a DOM objects
- parent() : a function to return the parent object (i.e. the one that encloses this object)
- hasClass("classname") : a function to find out whether a given object has a given class attached to it
- attr("name") : a function to get or set the value of a given attribute
- children() : a function that returns the list of child DOM elements inside this one
- css("name", "value") : a function to assign CSS style parameters to a DOM element (e.g. $("#pageheading").css("color", "red");)
- click(function(){…}) : adds a click event handler to the selected object(s)

There are many more jQuery wrapper functions added by jQuery selectors to DOM objects – a fuller list can be found in this document: http://james.padolsey.com/stuff/jQueryBookThing/.

There is one other significant difference in what jQuery selectors return compared to the plain DOM methods – the jQuery selectors always return an array of objects. The DOM getElementByID() function returns a single object; since each element in a web page must have a unique ID if it has one, this is a sensible thing to do. All of the other getElementByXXX() functions return an array of

matching elements. However, in jQuery, it is accepted that it is more useful to be able to deal with selected elements in the same way, no matter what the selector is, and so even `jQuery("#myelement")` will return an array, even though it has only a single object in it. Note that this behaviour can sometimes catch you out. For example, if you have an element with the id "mainheading" and you wanted to change that on the page, you might assume that the following code placed inside `$(document).ready()` would do for updating it:

```
$("#mainheading").innerText = "A new heading";        // Doesn't work
```

Unfortunately that would fail because `$("#mainheading")` returns an array - not a DOM element. It will fail silently (no error alerts or warning beeps) because that is standard for errors on web pages (otherwise, many would never shut up). To get to the actual element, you would need to use this code:

```
$("#mainheading")[0].innerText = "A new heading";    // Fixes the problem
```

since then you would be addressing the first (and only) element of the array of matching elements. Of course, the jQuery text() method would work without having to access the array element:

```
$("#mainheading").text("A new heading");       // Works as expected
```

Generating DOM elements

Lets look at a simple application of these functions – code that builds a table of squares into a document and then applies some CSS styles to it. The point of this application is to demonstrate how to access and manipulate DOM Elements.

1. We start by creating a simple HTML document. The <body> element contains a table with some headings. The application includes a JS file called buildTable.js:

```
<!DOCTYPE HTML>
<html>
<head>
   <title>Squares of Integers</title>
</head>
<body>
   <table id="table" border="1">
     <thead><th>Number</th><th>Square</th></thead>
   </table>
</body>
<script type="text/javascript" src="buildTable.js"></script>
</html>
```
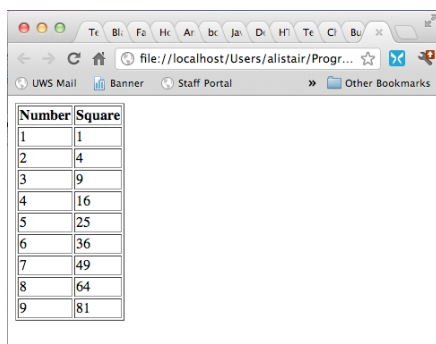
Listing 7.6: A document we will manipulate

2. We now add the `buildTable.js` script file.

```
var fillTable = function(){
    var t = document.getElementById("table");      // Select the table element
    if(t){                                          // If that worked…
       var r, cell, n=1;                            // we'll need these variables for the task
       while(n<10){                                 // we'll do squares 1 to 9
          t.insertRow(-1);                          // add a new row to the end (-1) of the table
          r = t.rows[t.rows.length-1];              // set r to be the new row element
          r.insertCell(-1);                         // add a cell to the row
          r.insertCell(-1);                         // add another
          r.cells[0].innerHTML = n;                 // set the first cell's content to the value n…
          r.cells[1].innerHTML = n*n;               // …the second cell's to n squared
          n+=1;                                     // now increment n
       }
    }
};
window.onload = fillTable;                          // When the page loads, call this…
```

Listing 7.7: The script file that does the manipulating

3. We can now load the document into a browser to see the result:

Figure 7.8: The result in a browser

We'll now add some style to this document. You can see tables laid out in many web pages, and one feature that appears commonly is that alternate rows of the table have different colours. This lets the user read across a long row of the table without losing their place.

To do this, we will need to define two CSS classes. These will be added to the HTML document inside a <style> tag. We'll then add the code needed to apply the styles to the table rows.

4. Add two CSS classes to the HTML document <head>:

```
<head>
   <title>Squares of Integers</title>
   <style> <br>
     .odd{background-color: white;}
     .even{background-color: gray;}
   </style>
</head>
```

Listing 5.8: Adding CSS classes to the document

Note the format of the CSS style settings – a dot, followed by a class name, followed by a class definition in curly brackets.

5. Next, add the additional code to apply the styles to the table rows:

```
var styleTable = function(){
   var rows = document.getElementsByTagName("tr");      // Get all the rows.
   if(rows){                                            // Don't assume
     for(var n= 0, num=rows.length; n<num; n++){        // Step over each row

       if(n%2){                                         // if n is even
         rows[n].className = "even";                    // apply 'even' style
       } else {
         rows[n].className = "odd";
       }
     }
   }
};
```
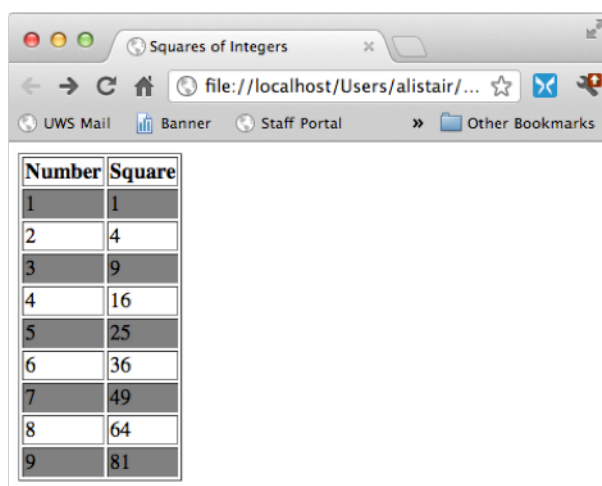
Listing 7.9: The code needed to apply the styles to the table rows

6. To incorporate it into the code, call the styleTable() function from the end of the fillTable() function:

```
       // ^^^ The fillTable() function is here.
       n+=1;
     }
     styleTable();                // Call the function to style it.
   }
};
```

Listing 7.10: The amendment to the fillTable() function to call styleTable()

7. Finally, refresh the page in the browser to see the result:

Figure 7.9: A table with style.

In this chapter we have merely scratched the surface of the BOM and DOM browser components. I've rather freely mixed plain Javascript code with jQuery code; no apologies for this – if you find yourself looking for help by Googling for examples of code (e.g. "HTML tables alternate rows" was how I found out how to style a table as in Figure 7.9), you'll also find that you need to be able to follow both types of code.

There are many good online sources that provide good, detailed knowledge of each of these aspects of web programming. The best approach to getting familiar with them is to learn how to use Google (other search engines exist..) to get solutions for what you wan to do. The list below is of site that have general-purpose usefulness:

Online Sources

W3CSchools HTML DOM Tutorial - http://www.w3schools.com/htmldom/default.asp

Wikipedia: Document Object Model - http://en.wikipedia.org/wiki/Document_Object_Model

Introduction to the DOM of IE/Firefox - http://www.javascriptkit.com/javatutors/dom.shtml

W3CSchools Javascript and DOM reference - http://www.w3schools.com/jsref/default.asp

BOM - http://hwang.cisdept.csupomona.edu/cis311/client_side_behavior.aspx?m=bom

Microsoft BOM - http://msdn.microsoft.com/en-us/library/ms952643.aspx

Javascript & The BOM - http://www.daaq.net/old/javascript/index.php?page=the+js+bom

Questions

1. Explain the benefits of designing an object model that works for any given application

2. What is an *abstract method*, and why would one be added to an object type?

3. Why would you define an event within an object type definition, and how would a user of the object type be expected to employ it?

Exercises

Open a small HTML page in a browser (Chrome, Safari, or Firefox with Firebug installed) and use the browser's console window to interrogate the BOM and DOM to find out the following:

1. How many anchors (URL links) does the page contain?

2. What is the host name of the current window's location?

3. What is the width and height of the screen the browser is displayed on?

4. How many pages were visited before you loaded the current page (the window's history property has a length)?

5. What is the browser's navigator's application name (be prepared to be surprised by this)?

6. What is the browser's userAgent setting?

7. What is the URL of the current document?

8. Using the Javascript console only, create a new <h1> element, set its innerText property to "New H1" and insert it as the last element in the document's body section. Note – you'll need to use document.createElement(), apply the value to the element's innerText property and then use document.body.appendChild() to add it into the document.

9. How many paragraph (<p/>) elements are there in the document?

10. Using document.getElementsByTagName(), select the first <p/> element in the document and change its innerText property so that it shows "Changed By Console".