Scripts

This is the browser technology which brings web pages to life

JavaScript  browser objects  animation
This chapter is about JavaScript as a client-side technology, for writing scripts that run in browsers. It is one of our three major languages:

- **HTML**: for the content
- **CSS**: for what it looks like
- **JavaScript**: for what it does
What it's used for

In browsers, JavaScript is used for:

- inessential things (e.g. Google sets focus)
- click and mouseover effects (beyond CSS)
- pop-ups and similar effects (mostly bad)
- validating forms before submitting them
- drag-and-drop effects (e.g. re-ordering lists)
- animation (of content, canvas)
- fetching data and generating HTML

For the more complex of these things, it is common to use JavaScript libraries or frameworks
A simple server-side framework design is:

A simple client-side framework design is:
Frameworks

Client-side frameworks for RIA\textsuperscript{s} include Bootstrap, Angular, jQuery/jQlite, ...

Server-side frameworks: Express, Meteor,...

Use frameworks if you want to get further, or not if you want to learn more, but in any case:

- be aware of the \textit{tensions}
- be aware of whether a framework design suits you
- check what is going on underneath the surface
The JQuery (or jqLite) library:

- it has a long learning curve - another language
- tutorials are full of terrible syntax
- it tends to cover up the 'truth'
- its support for old browsers is obsolete
- a lot of its best features, e.g. selectors, drag-and-drop, can now be done directly in HTML5

But it still has some useful features, so use it if you like
Polyfills

We saw a tiny polyfill before, for the `section` tag:

```javascript
document.createElement('section');
```

```css
section { display: block; }
```

JavaScript can be used to add a style element to the DOM, so this can be done just with JavaScript.

To take advantage of other people's work, use ready made polyfills for old browsers (if you care) or old versions of HTML5 browsers.
The first example you see might be:

```javascript
$(document).ready(function() {
  $("button").click(function () {
    $("p").toggle();
  });
});
```

Removing the in-line function style to avoid callback hell, this is:

```javascript
$(document).ready(start);
function start() {
  $("button").click(change);
}
function change() {
  $("p").toggle();
}
```
Downloadable programs produce a potential security problem, because any **hacker** can write arbitrary JavaScript code, put it on a web page, and have it executed by any unsuspecting visitor.

So JavaScript can **only** access the current web page and its own site, and interact with the user, **not** access a user's files or devices or programs, or other network sites (though HTML5 & WebSockets are easing this).

There were some early security problems, but JavaScript is now **officially safe**.
Some people disable JavaScript, even though it is safe now. They are wrong: JavaScript is one of the integrated technologies that are supposed to be guaranteed to be available on every browser.

It is the official way to animate pages, and it is often essential (e.g. see Google maps)

However, JS may not work in some fairly rare circumstances, so ideally, you should try for graceful degradation
ECMAScript and DOM

The **ECMA5** standard describes a reasonably good object oriented language, which is well supported by all browsers, with **ECMA6** coming along.

The **DOM = Document Object Model** is a substandard of HTML5 defining the structure of a web page in memory, as a tree of objects seen by JavaScript, plus the **window** object representing the browser itself.

In the past, adherence to the DOM standard was poor, so polyfills are needed for old browsers.
Reference Materials

We've already seen resources for JavaScript itself, but we also need information about the DOM and browser-related techniques.

The caniuse site has browser compatibility tables, to check for cross-browser support for particular features.

The Unobtrusive JavaScript and HTMLSource tutorials are good, but beware: all tutorials contain obsolete info.
Inclusion of Scripts

The right way to include a script file is one of:

```html
<script src="file.js"></script>
<script src="file.js" defer="defer"></script>
```

Do this in the head element

There are lots of wrong ways:

- don't put scripts at the end
- don't use event attributes
- don't abbreviate the script tags
- don't put content in the script element
- don't escape problem characters
Don't put scripts at the end

Some people put script elements at the end of the page, just before </body>:

```html
<script src="..."></script>
</body>
```

This is legal, but not in this unit because I need to know where to look to find your script inclusions.

The idea is to make the page display faster, by delaying the script.

This is better done now with the `defer="defer"` flag (plus `async="async"` if you want).
Don't use on... attributes such as:

```html
<body onload="...">
<a onclick="...">
<form onsubmit="...">
```

These are legal (but not in this unit), and work but, like style attributes for CSS, they have two problems:

One is that you cause maintenance effort: scripting is scattered, and not shared in just one place.

The other is that you may develop bad habits, so it is better to avoid using them altogether.
Don't do this:

```html
<script src="file.js" />
```

This is a valid XML abbreviation, because the element has no content.

But the polyglot rules require an explicit close tag for an element which could have some content, and require the abbreviated form for void elements where there can't be any content.
No Content in Script Tags

Don't do this:

```html
<script>...code...</script>
```

This is bad because:

- it doesn't separate out JS from HTML
- characters like `<`, `&` cause XML problems
Don't escape characters

Don't do this:

```html
<script>...&lt;...&amp;...</script>
```

This works in XHTML because script text is part of the XML, but not in plain HTML because script text is treated as verbatim.

If you must include a script in the HTML (in later life, not this unit) there is a way of escaping the characters using `CDATA`, but check the polyglot HTML5 document carefully to find out how (it is not pretty).
Browser Differences

There are very few differences in the language itself.

For DOM differences, there is no neat way of providing a separate script for difficult browsers.

People used to use browser detection ("if IE .. else if FF ...") but this is terrible and should not be used any more because it is unstable – the right approach is polyfills or object detection:

- "if object x exists, use it" events
- "if function f exists, call it" listeners
Events

The standard form of an event handler is:

```javascript
function keypress(event) { ... }
```

An event object is passed, but not in IE8, so a polyfill is:

```javascript
... if (!event) event = window.event; ...
```

where `!event` is true if `event` is `undefined`, or:

```javascript
... event = event || window.event; ...
```

(that's if you like obscure code)
A standard call for adding an event listener is

```
addEventListener('load', start);
```

But not in IE8, so a polyfill is:

```
if (addEventListener) addEventListener('load', start);
else attachEvent('onload', start);
```

or more generally you can create a new `addEventListener` method on `window` in IE8 which mimics the standard one
You can debug with `console.log` or maybe `alert`:

```javascript
console.log("reached here with x =", x);
alert("reached here with x=
 + x);
```

The quickest way into the console is right click, inspect element, Console.

Advantages of `console.log` are that you can use it on a live page, you don't need to click to continue, and it has breakpoints, single stepping, watching.
A script can have "use strict", immediate code, global variables, and/or function definitions.

```
"use strict";
console.log("starting");
var count = 0;
function go(...) {
  var count = 0;
  ...
}
```

The immediate code runs when the page is loaded, and the globals and functions remain, to be used later.
Lack of modules

On web pages, there is no module system (before ES6)

It is common to have multiple scripts on one page

All their global variables and functions are mixed, in fact they are all attached to the `window` object

That causes name clashes and other bad interactions, unless care is taken to play nicely

There are some libraries which support ES6-like modules, but there is a common 'simple' DIY approach (pseudo-modules)
Pseudo modules

You can create pseudo modules by putting a whole module inside a power constructor or class

```javascript
function newCounter() {
    "use strict";
    var n = 0;
    counter.get = get;
    counter.add = add;
    function get() { return n; }
    function add(m) { n = n+m; }
}
```

- local strict
- private/public
function ... {
    "use strict";
    ...
}

This "use strict" declaration only affects the function body (which is essentially the whole module)

If "use strict" was done globally, it could cause another sloppy script on the same page to fail, so doing it at the top of the function body is safer
The variable `var n = 0` is private because it is not attached, and is only accessible via the closure.

Public fields like `get` are explicitly attached, then accessed via `this`.
A suggestion for unit testing is to have a page test.html containing:

```html
<script src="X.js"></script>
<script>newX().test();</script>
```

and define a public test method inside the script

```js
function newX() { ...
 X.test = test; ...
 function test() {
  is(X.get() == 0); ...
  function is(b) { if (!b) throw 'test failure'; } 
 }

}```
A script is executed *during* page-loading, so the page (document object) is *not* reliably available! Do this:

```
"use strict";
addEventListener('load', start);
function start() {
  ...
}  called after loading
```

**Never** use `document.write`

Don't use `window.onload = start` because it removes other scripts' listeners

A common mistake is to *call* `start()` when passing it
A lot of obsolete tutorials show you how to use `document.write("...")` to insert computed HTML text into a page (e.g. today's date)

However, this is strongly discouraged in HTML5 and forbidden in polyglot HTML5 because it doesn't work in XHTML

Instead, you should wait until the page is loaded, and then find and change the appropriate page content
Page Objects

Scripts have access to page objects, and those objects have fields and methods (e.g. `alert("url = " + location.href);`)

```javascript
var url = location.href;
```

The two most important top level objects are:

```
window  (the current browser window or tab)
(all globals are attached to this)
document (the current XHTML page)
```

See caniuse or quirksmode for available methods with compatibility tables
The DOM tree

The `document` object represents the whole page.

`document.documentElement` is the html element, and `document.body` is the body element.

`node.childNodes` is a list of all subnodes (including text and comments), and `node.children` is a list of subelements (excluding text and comments).

Also, there are `node.firstChild`, `node.lastChild`, `node.parentNode`, `node.id`, `node.className`, `node.classList`. 
There are two ways to get attribute values, e.g. for a link node:

```javascript
link.getAttribute("href")  // gets the original string value of the href attribute, e.g. "page.html"
```

```javascript
link.href  // gets an interpreted version of the attribute, maybe an object rather than a string, in this case the expanded full URL "http://www.../page.html"
```

Style is included in the DOM as objects, e.g.

```javascript
node.style.color
```
Selecting objects

You can select objects using the same ideas as CSS, using the `querySelector` or `querySelectorAll` function on `document` or a node:

```javascript
node.querySelector("h2"); // by tag name
node.querySelector("#save"); // by id
node.querySelector(".angry"); // by class
```

One method returns the first node found, the other returns a list of all nodes found.

Note: `getElementById("x")` is the same as `querySelector("#x")` and similarly for `getElementsByTagName`, `getElementsByClassName`
Here's how to create a button which pops up an alert:

```html
<button id="b">Press me</button>
```

```js
"use strict"
addEventListener('load', start);

function start() {
    var b = document.querySelector("#b");
    OR var b = document.getElementById("b");
    b.addEventListener('click', popup);
    OR b.onclick = popup;
}

function popup(event) {
    alert("You pressed the button!");
}
Default behaviour

Suppose you add an event handler for the `click` event in a link `<a href=...` which is called just before following the link:

```javascript
function click(event) {
    alert("read this first");
}
```

To cancel the default behaviour, include:

```javascript
    event.preventDefault();
    event.stopPropagation();
```

Note: `return false` at the end is equivalent to calling `event.preventDefault()`
To change the page, there are two techniques, one being to use methods like these:

```javascript
document.createElement('p');
document.createTextNode('text');
node1.appendChild(node2);
node1.removeChild(node2);
```

An approach which is often simpler is to use:

```javascript
node.innerHTML = '<p>text</p>';
```

- `innerHTML` changing style
For a long time, `innerHTML` was non-standard, and well supported except for a couple of browsers which failed to support it in XHTML mode.

But it is now in HTML5.

**Beware:** when you write to it, the text you use **must** be a valid XHTML fragment, or there'll be an exception.

**Beware:** if you read it, you will probably not get what you want (e.g. extra xmlns attributes) so this is not recommended.
Changing Style

You can do e.g. `node.className = "angry"` so that the style of a node changes via the stylesheet.

You can change style directly, e.g. to toggle visibility do
```
node.style.display = 'none'
```
```
node.style.display = 'block'
```

In dynamic situations, you can change style with e.g.
```
node.innerHTML = "<p style='..'>..</p>"
```
If you want to remember something from one visit to another, e.g. the current slide number on this slideshow page, the easiest way is to use local storage:

```javascript
localStorage.setItem("slide", current);
current = localStorage.getItem("slide");
```

It is much easier than using 🔄 cookies.

The same 🔄 restrictions apply.

Local storage variables are site-wide, so you may want to include the site-url in the name.
A cookie is also a variable=value string pair stored in the browser.

The differences are: (a) cookies are sent with every request to the server and (b) code to handle cookies is much more complex.
Restrictions

Both cookies and local storage are limited in how much space they can take up - use only for snippets of data.

According to the European Union, sites must declare their use of cookies or local storage to store data, and give visitors a chance to opt out.

This applies to visitor tracking, ad targeting, and other things which visitors might reasonably object to.

A declaration isn't needed for intranets or where the data is used only for 'the normal operation of the site'.
There is a facility for making server requests:

```javascript
var q = new XMLHttpRequest();
q.onreadystatechange = receive;
q.open("GET", url, true);
q.send();

function receive() {
  if (this.readyState != XMLHttpRequest.DONE) return;
  var s = this.responseText;
  ...
}
```

The name is bad - the request needn't use XML

This example gets a text file, but JSON, binary or custom formats are OK, as is putting a file
The general strategy of using server requests is called Ajax, which makes it sound too grand.

The example is asynchronous with a callback, but a synchronous call is also possible (but not recommended).

The example doesn't set any request headers, and doesn't check for errors.

POST with a suitable content type header is OK for form-like data, but to send a file, it is much simpler to use PUT, with a filename as a url parameter, and the request body consisting of exactly the file content.
Example: forms

A form sends parameters to an action page:

![Diagram of form, URL with parameters, action page]

An action page normally uses server-side programming to deliver a parameterized answer, but we will use a plain HTML action page with JavaScript instead for now, to build a page which displays the parameters.
The form page contains a simple form like this:

```html
<form action="params.html" method="get">
  <input type="text" name="pet" value="cat" />
  <input type="text" name="car" value="bmw" />
  <input type="submit" value="Go" />
</form>
```

Pressing the button is like clicking a link; it just jumps to a different URL.

The page `params.html` is the action page, and the method is "get" for debugging, so we can see the data in the URL.
Example: parameters

The form goes to this URL (with newly typed values)

```html
...params.html?pet=cat&car=bmw
```

The URL is followed by a ? and then pairs of parameter names and values, separated by &

When the parameters are in the URL like this, there are many potential problems, but the browser translates difficult characters

With method "post", the parameters go in the body of the HTTP request, not visible, and no restrictions
Example: the script

The page for our parameter viewing exercise contains a default message and an empty list:

- visit form.html
- visit params.html
- visit params.html?pet=cat&car=bmw
- view source of params.html
- view params.js

The script finds the URL and picks it apart to get the parameter names and values, then creates a list entry for each pair and adds it to the list:
Stale Data

Suppose Alice goes to a form page and thinks about updating, and meanwhile, Bob goes to the same page, edits the data, and presses commit.

Now Alice commits, and Bob's changes are lost!

One solution is to make sure that each piece of data is only updated by one person.
Example: marks entry

Suppose Alice and Bob are entering marks for an assignment using a form like this:

<table>
<thead>
<tr>
<th>Student</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>45</td>
</tr>
<tr>
<td>B</td>
<td>73</td>
</tr>
<tr>
<td>C</td>
<td>61</td>
</tr>
</tbody>
</table>

Alice is seeing marks inserted from the database, before Bob's update

When she submits, Bob's updates are lost
Example: marks entry

But suppose the form looks like this (as in SAFE):

Student A 45
Student B 73
Student C 61

The existing mark is shown, plus a box for an updated mark

When a marker submits, only filled boxes have any effect, so all is well provided Alice and Bob never mark the same student.
In more complex circumstances, a more sophisticated approach is needed; to load-enter-submit a form needs to be treated like a transaction

A 'locking' technique would hold the lock for too long, possibly forever if the page is abandoned

A 'versioning' technique is better: the form holds the original values as hidden data, checks to see if they have changed on submit, and gives an error-please-reload-and-retry message if so
Animation

Animation means different things to different people: in the film industry, it means *offline, frame*-based generation of successive *non-interactive* images.

On the Web, it means anything behavioural, e.g. *online, programmable* generation of successive images, *not* necessarily frame-based, which may be *interactive*.

A long non-interactive animation on the Web suspends surfing and spoils the active feel of the Web, so it is best regarded as a different environment with plenty of warning on entry.
Old animation technologies

Animation no longer involves obsolete plugin-style technologies such as Flash, Java applets, QuickTime, AVI, Real, FLI/FLC, animated GIFs

Some of these will take time to die because of the re-engineering needed to transfer to HTML5
New animation technologies

For non-interactive frame-based animation, you can create a video, and many simple effects can be achieved using CSS features.

General animations consist of JavaScript manipulating the DOM, including canvas, SVG, MathML, WebGL (3D).

JavaScript is now efficient enough to do impressive things, e.g. emulators of old computers.

For efficiency in sophisticated animations, you need a single timer controlling everything, not a separate one for each object, see: Schillmania.
The canvas element in HTML5 gives you a raw area to draw on, and JavaScript functions to draw with.

It appears to use pixel coordinates, but is actually scalable, see: wanderer

There are plenty of HTML5 Canvas animation tutorials and example sites.

The style is frame-based, i.e. draw then clear then redraw, which is not very OO, but there are some interesting libraries which use it.