Advanced JavaScript for Web Sites and Web Applications

JavaScript Problems

Problems

- JavaScript programs won't always run as expected!
  - Some devices don't support JavaScript or support a limited version of it
  - The user (or System Administrator) might switch off JavaScript support in their device
  - Different browsers/browser versions = different language features/behaviour

So What?

- Consider our *FormRecall* app that used *localStorage* to store and retrieve form data for the user
- Imagine a user with Javacript disabled or who is using a browser that does not support *localStorage*
- Result: buttons that promise something, but that do nothing!

Dealing with problems

- There are 3 common approaches to these problems:
  - Ignore them!
  - Employ *Graceful degradation*
  - Employ *Progressive enhancement*

Option 1: Ignore them

- This is not an option!
  - results in confused and disillusioned users
  - undermines user’s impression of website

Option 2: Graceful degradation

- Graceful degradation (*GD*) is a way of writing code so that *modern* browsers will enjoy a full featured app
- But users of older browsers will be presented with a *crippled* app, where not all features will work
  - But it should still be usable/error free

Graceful degradation and `noscript`

- The HTML `<noscript>` tag is a tag who's content is only displayed when the user's browser does not support JavaScript
- It is frequently used with *GD* to inform the user that certain features will not work.

```
<script src="myFunkyScript.js"></script>
<noscript>
    This website won't work without JavaScript!
</noscript>
```

`noscript` and manners

- Some developers use the `<noscript>` tag to encourage their users to upgrade/change their browser

```
<noscript>
    Please upgrade to a modern browser that
    supports JavaScript properly!
</noscript>
```

- However, this is considered rude/bad practice as the user does not always get to choose their browser!

Graceful degradation and FormRecall

- To apply *GD* to *FormRecall*, we might add a `noscript` tag to our HTML with a message informing the user that some of the buttons will not work

Graceful degradation and FormRecall

```
<noscript>
    <p>
    Sorry, we won't be able to remember your data
    as JavaScript is not enabled in your browser.
    </p>
    <p>But you can still submit the form :)</p>
</noscript>
<form role="form" id="adduser">
    <!-- The app form -->
</form>
```

Option 3: Progressive enhancement

- However, the preferred approach in modern web development is to employ *progressive enhancement* (*PE*)
- *PE* is a way of writing code with the same result as *GD*, but approaches the problem from a different angle

Progressive enhancement - overview

- We start by building an application that provides the basic functionality
  - usually, without JavaScript!
- We then use JavaScript to enhance the app
  - assuming the browser supports JavaScript…
  - and the browser supports the features we are using

PE with the FormRecall app

- To apply this concept to our form recall app, requires a restructuring of both the HTML and JavaScript code

PE with FormRecall: The HTML...

- First, we remove the *restore* and *clear* buttons from the HTML
  - these buttons only work with JavaScript, so should only be displayed if JavaScript is available!
- The *Add user* button will remain in the HTML
  - we can assume it will be linked to a server-side script (i.e. it will work without JavaScript)

PE with FormRecall: The JavaScript…

- The buttons are:
  - created dynamically within the module
  - inserted into the DOM (using `insertAdjacentHTML` or similar)
- No Javascript… code never runs… No misleading buttons displayed
- We can do this in a function…

PE with the FormRecall: Build the interface

```
prepareInterface = function () {
    var cBtn, rBtn, btn;
    btn = document.getElementById('add-button');
    // The buttons:
    cBtn = '<button id="clr">Clear</button>';
    rBtn = '<button id="res">Restore</button>';
    // Add them to form:
    btn.insertAdjacentHTML('afterend', rBtn);
    btn.insertAdjacentHTML('afterend', cBtn);
    // After adding them, get references:
    clearButton = document.getElementById('clr');
    restoreButton = document.getElementById('res')
};
```

PE FormRecall - Build the interface

- The `prepareInterface` function:
  - Creates the 2 buttons (as HTML strings)
  - Inserts them into the DOM (after the “Add” button)
  - Gets node references to them (to store in module variables)
- We need to call this function from our `init` function, before we do anything else

PE FormRecall - The `init` function

```
init = function () {
    // Make buttons first... Can't add event
    // handlers until they are present in DOM!
    prepareInterface();
    // Now do other stuff...
    form.addEventListener("keyup", storeValue);
    restoreButton.addEventListener("click",
        populateForm
    );
    clearButton.addEventListener("click",
        clearStorage
    );
};
```

PE Benefits

- By employing this strategy:
  - Users with JS enabled get the benefits our module brings to the app
  - Users without JS can still use the core functionality of the form
  - Nobody is confused or disillusioned by our app!

Dealing with missing features

- However, there is still a problem…
- Consider a user with JS enabled, but using a browser that does not support `localStorage`
- They will see the buttons, but the buttons will do nothing!
  - More broken promises!

Missing features Solution

- To deal with this scenario, we can use a technique called *feature detection*
- *Feature detection* is the process of checking whether the user's browser supports a certain feature *before* using it
- Depending on the feature, there are different ways to check for support…

Simple feature detection

- We can check for properties/methods of the document object with a simple comparison expression.

```
if (document.querySelectorAll) {
    // browser supports the method, use it
} else {
    // browser does not support the method,
    // do something else...
}
```

Advanced feature detection

- Sometimes, things are not so simple.
- When testing for `localStorage` support, some browsers will throw *exceptions* as soon as we try to do *anything* with `localStorage`
  - Even using it in a comparison test
- So, we need a different approach…

Try/catch

```
// The "try/catch" construct
try {
    // Code placed in the "try" will be executed.
    // If any of it throws an exception, the JS
    // parser will jump from it to the catch
    // block below. Nothing in the try block
    // that follows the code that threw an
    // exception will be executed...
} catch(e) {
    // Code in here runs if exception is thrown
    // in try block above. If no exception, this
    // never runs...
}
```

Try/catch

- If an *exception* is thrown by code in the `try` block, the code in the `catch` block will run.
- Otherwise, only the code in the `try` block will run
- In short: If something goes wrong in the `try` block, run the code in the `catch` block
  - Code in the `try` block that follows the *exception* does not get executed

localStorage feature detection

```
// Based on modernizr code
function isStorageAvailable() {
    try {
        // This might throw exception:
        localStorage.setItem('test', 1);
        localStorage.removeItem('test');
        // If no exception above, this runs:
        return true;
    } catch(e) {
        // If exception above, this runs:
        return false;
    }
}
```

PE and feature detection

- We can test for localStorage support before we initialise the module:

```
// "canStore" will be true if localStorage
// is supported by browser
var canStore = isStorageAvailable();
if (canStore) {
    FormRecall.init();
}
```

- Or, we could check from inside the module before setting things up (in the ‘init’ function)

Browser differences

- Another thing that effects our scripts is inconsistencies between the different browsers
  - And between different browser versions
- While this is not as much of a problem as it once was, it still needs to be considered

Browser differences: example

- Attaching events to elements in most browsers is done with `addEventListener()`:

```
element.addEventListener(event_type, func);
```

- But in older versions of IE, we use `attachEvent()`:

```
element.attachEvent(event_type, func);
```

Browser differences: example

- Additionally, IE prefixes the event names with *on*:

```
// Most browsers
element.addEventListener('click', func);

// IE
element.attachEvent('onclick', func);
```

Dealing with browser differences

- To deal with these differences, we can use a variation of the *feature detection* we saw earlier:

```
if(element.addEventListener) {
    element.addEventListener('click', myFunc);
} else if (element.attachEvent) {
    element.attachEvent("onclick", myFunc);
} else {
    // For prehistoric browsers!
    element["onclick"] = myFunc;
}
```

Dealing with browser differences

- Writing code like we saw on the previous slide for every event handler we define will not be fun!
- Instead, we can implement the *Facade* pattern to abstract the complexity away from our code

The *Facade* pattern

- The *Facade* pattern is all about hiding complex code behind a simplified interface
  - LIbraries such as jQuery use it a lot!
- There are many ways to implement the pattern: simple functions, revealing modules, etc.

A *Facade* pattern function

```
function addEvent(element, eventType, func) {
    var onType = "on" + eventType;
    if(element.addEventListener) {
        // Modern browsers
        element.addEventListener(eventType, func);
    } else if (element.attachEvent) {
        // Old IE
        element.attachEvent(onType, func);
    } else {
        // Prehistoric browsers!
        element[onType] = func;
    }
}
```

Using the *Facade* pattern function

- Any time we want to add an event listener to an element, we call our *facade* function:

```
var myEl = document.getElementById('blah'),
    myEvt = 'click',
    myFunc = function (event) {
        // Do stuff
    };

addEvent(myEl, myEvt, myFunc);
```

The *Facade* pattern with modules

- If we have other browser differences to cater for, we can:
  - create a *revealing module* containing all of the *complex* functionality to deal with differences
  - expose methods in the returned object, allowing other code to easily access that functionality

The *Facade* pattern with modules

```
var gQuery = (function () {
    var addEvent = function(el, event, fn) {},
        doAjax = function(url, fn1, fn2) {},
        selectAll = function (selector) {};
    return {
        addEvt: addEvent,
        select: selectAll,
        ajax: doAjax
    };
})();
gQuery.addEvt(myElement, myEvent, myFunction);
```

Getting help...

- Dealing with browser differences can be a complex thing to achieve
- When writing large apps, it can make sense to utilise a library such as jQuery which takes care of a lot of the inconsistencies for you
- But, including an entire library can be overkill if the app only uses a small number of its features

Shims and polyfills

- As an alternative to including an entire library, which attempts to fix *all* differences between browsers…
- … We can use *shims* or *polyfills*, which are functions/snippets of code designed to solve *one particular issue*

Example app

- Consider a simple app that toggles the `class` attribute of an element when a button is clicked.
- To achieve this, we might use:
  - `getElementById()`, `addEventListener()` and `classList()`
- However, `addEventListener` and `classList` are not consistently implemented across all browsers!

To shim or not to shim

- **Option 1:** Use jQuery, which has cross-browser methods that do the same thing as `addEventHandler` and `classList`
- *Side-effects:*
  - approx 100kb of extra code for browser to load
  - Temptation to use jQuery for things which might be better done with plain JavaScript (e.g. `getElementById`)

To shim or not to shim

- **Option 2:** Use plain JavaScript with `classList` and `addEventListener` *shims* (from the MDN website):
  - classList
  - addEventListener
- *Side-effects:*
  - approx 8kb of extra code for browser to load

Summing up

- When writing JavaScript, be aware of the differences that exist between browsers/versions of browsers/versions of JavaScript
  - Check the MDN website, http://caniuse.com, etc.
- Test your code in as many browsers and devices as possible
- Test your code in different environments (locally and on a public web server)
- If possible, decide which browsers you want to support *before* you wite any code