Client-Side Programming

Client-Side Programming

- So far, we have talked mostly about programming on the server-side.
  - There is no way to ignore how important the server is: it's the only thing you can rely on for security, data storage, etc.

- Logic can also be embedded in HTML and executed by the user's browser.
  - The `<script>` tag is used to insert logic into a page.
  - In theory, `<script>` can contain any language, but since you rely on browser support, it's going to be JavaScript.
  - Like with HTML and CSS features, you are relying on the user's browser to run your code. That means…
    - The code might not run. They might have JS turned off, Googlebot won't execute your JS, they might be using a weird browser without JS, ….
    - The code might not work the same way it does in your browser. Different browsers implement different language features. Your have to worry about code compatibility a lot.
  - It can be used to reference an external URL with code:

    ```
    <script type="text/javascript" src="codefile.js"></sc
    ```

  - You can also embed code directly in your HTML:

    ```
    <script type="text/javascript">
      /* code goes here */
    </script>
    ```

  - The `type="text/javascript"` is optional in HTML5.

  - *Client-side scripting [enwiki]*
  - *JavaScript [enwiki]*
  - *JavaScript [zhwiki]*

- Putting code in your HTML (instead of a separate file) is probably a bad idea.
  - Bad style: mixing more than one language in a single file is hard to read.
  - Hard to maintain: if some logic is broken, can be hard to find (especially since there can be multiple `<script>` in a page).
  - Validation problems: if you want to make a comparison in your JavaScript: `i<10 && j>0`.
    - Does that `<` start a tag?
    - If embedded in HTML, that should be `i&lt;10 &amp;&amp; j&gt;0`.
    - Do you want to code like that?

- No caching.
  - If the user needs the same JS several times (because they return to the same page, or visit multiple pages from the same template) it must be loaded every time.
  - In a separate file, it can be loaded once and cached.
- Exception: sometimes you need to call a function to get things started. That should be small, but needs to be done for the specific page.

```
<script type="text/javascript" src="/code/main.js"></
<script type="text/javascript">
  setup_this_page();
</script>
```

JavaScript Basics

- First: the Java and JavaScript programming languages are not related.
  - JavaScript was originally called LiveScript.
  - At the time, Sun was heavily promoting Java for applets on the web.
  - Netscape renamed their language JavaScript to try to borrow some of that marketing.
  - JS is also known as ECMAScript and JScript.

- Not the nicest programming language in the world, but far from the worst.
  - Dynamically typed which means (among other things) that you don't have to declare variables.
  - Object-oriented.
  - Some very functional-programming-like features.

  - *JavaScript: The World's Most Misunderstood Programming Language*
  - *Eloquent JavaScript*
  - *Mozilla's JavaScript Reference*
  - *W3Schools, An Intervention*
  - *The rise and rise of JavaScript*

- But…
  - Browser compatibility is a mess: different browsers implement the language slightly differently. Requires a lot of testing to be sure things are working. Programmer must always be aware of these differences.
  - Lots of bad documentation out there: find something good. Please stay away from w3schools: many things it says are wrong, almost-wrong, misleading, or just boringly missing the point.
  - A programmer can cause unexpected/annoying/stupid behaviour on pages. Don't.
  - Anything generated by JS won't be seen by Google (`curl`, URL fetching APIs, etc.) or by browsers with JS disabled. Be cautious using JS for “core” functionality.

    - It's easy to think of the code as “always” running. Don't forget that you're relying on the client to run it: no security, input validation, etc can be done with JS (without being re-done on the server side).

- Basic data types:
    - Numbers: no distinction between int and float. Everything is a double. e.g. `34, 5.67, 1e20`
    - Strings: all strings are Unicode characters. e.g. `"Hello world", 'Hello world', "你好", "\u4f60\u597d"`
    - Boolean: `true` and `false`.
    - Date
    - `null` and `undefined`

- Everything in JS is an object.

```
var words = "one two three";
(words[4] == "t")
(words.length == 13)
var v = 14;
(v.toExponential() == "1.4e+1")
```

- Arrays: an ordered collection.
    - No fixed size: can grow/shrink as needed.
    - Can contain any types.
    - e.g. `[1,2,3], ["one", "two", "three"], ["one", 2, false]`
    - Arrays are objects with some generally-useful methods:

```
var numbers = [0,10,20,30,40];
(numbers.length == 5)
(numbers[3] == 30)
var x = numbers.pop();
(x == 40)
(numbers.length == 4)
```

- Objects:
    - Objects have two roles in JS: they work as associative arrays of key/value pairs:

```
var values = {
  "one": 1,
  "two": 2,
  "three": 3
};
(values['one'] == 1)
values['four'] = 4;
```

    - But they can also be accessed with object notation:

```
(values.one == 1)
values.five = 5;
```

- Again, there are some useful methods:

```
("one" in values)
(!("eight" in values))
```

- There are a couple of useful functions to do output.
  - `alert(foo);` Displays the value in a popup window.
  - `console.log(foo);` Displays the value in the console window in Firebug, etc.

- You should declare variables when they are used with `var`.
  - It's better style.
  - It makes sure you have a variable in the right scope: not accidentally using a global variable or something else.
  - e.g.

```
var counter=0;
```

- Statements don't have to end with semicolons, but should.
  - Several coding tools (most notably JSMin) assume they do.

Control Structures

- Annoyingly, JavaScript uses C-like control structures for most things.

- Conditionals:

```
if ( something == 4 ) {
  alert("I'm here");
} else if ( something == 5 ) {
  alert("I'm there");
} else {
  alert("I'm nowhere");
}
```

- For loop:

```
var i;
for ( i=0; i<10; i++ ) {
  console.log(i);
}
```

- While loop:

```
var i;
while ( i<10 ) {
  console.log(i);
  i -= 1;
}
```

Functions

- Functions are defined in a way that shouldn't bother you much:

```
function add(a, b) {
  var c = a+b;
  return c;
}
```

- Functions have their own local variables. (So does any other `{…}` block.)
  - i.e. `c` isn't defined outside of that function.
- Since all of the values are dynamically typed, the arguments take on whatever types they are given, and everything else follows:

```
(add(3,4) == 7)
(add("one","two") == "onetwo")
```

- Unlike C and Java, functions are first-class data types in JavaScript.
  - That is, a function is just a value. No better or worse than a number or array.
  - For example,

```
var another_add = add;
(another_add(4,5) == 9)
```

- This is perfectly legal too:

```
function create_adder(n) {
  function add_n(x) {
    return n+x;
  }
  return add_n;
}
```

  - Then we can do this:

```
var add10 = create_adder(10);
var add100 = create_adder(100);
/* add_n and n are both undefined here. */
(add10(5) == 15)
(add100(5) == 105)
```

  - What happened was that a new function was defined while `create_adder` was executing.
  - It was returned as `create_adder`'s result.
  - That result became `add10`, which is a variable holding a function. (`create_adder` is also a variable holding a function, but we don't usually think of it that way.)
- Since `n` is defined while the new function is being created, and it is used in the function definition, the new function holds onto its value.

  - The newly-created `add_n` function is closed over `n`.
  - Or `add_n` is a closure.

- Why would you want a closure? More later. [Short answer: callbacks.]

- Since functions are just values, you can define one without a name and save it in a variable.
  - Just like doing "`var s = "hello";`" creates a string (`"hello"`) that doesn't yet have a name and stores it in a varable.
  - Example: these are equivalent definitions.

```
function add(a,b) { return a+b }
var add = function(a,b) { return a+b };
```

  - In the second line, the function that's defined is anonymous since it isn't defined with a name.
  - Why would you want anonymous functions? More later. [Short answer: callbacks.]

Classes and Objects

- There isn't exactly a way to define a new class in JavaScript.
  - But because of the first-class-functions thing, it turns out to be possible.
  - You can always create an empty object (`{}`) and put whatever properties/methods you want into it.

- For example,

```
function Point2D(x,y) {
  var pt = {};
  pt.x = x;
  pt.y = y;
  pt.magnitude = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
  }
  return pt
}
p = Point2D(3,4);
(p.magnitude() == 5)
```

- Or equivalently,

```
function Point2D(x,y) {
  return {
    x: x,
    y: y,
    magnitude: function() {
```

```
        return Math.sqrt(this.x*this.x + this.y*this.y);
      },
    };
}
```

- That's a somewhat odd way to do classes, but it works.
  - Maybe not too surprising for a language that was designed very quickly, probably without serious software engineering in mind.
  - My experience says you don't really have to create new classes very often in JS programming.
  - You create a lot of objects (with the `{…}` syntax) but less often a formal class.

Callbacks

- Another result of first-class functions: function can be arguments to other functions.
  - Might not sound useful… until it is.

- For example, you can write functions like this:

```
function map(action, array) {
  var r, result = [];
  for ( var i=0; i<array.length; i++ ) {
    r = action(array[i]);
    result.push(r);
  }
  return result
}

function double(x) {return x+x}
(map(double, [2,3,4]) == [4,6,8]) /* or it would if "==" w
```

- In JS, there is a lot of event handling.
  - e.g. what should happen when a user clicks on a certain element, or rolls the mouse over an element.
  - What you really need to do in those cases is say which function the browser should call when the event occurs.
  - In other words, register a callback function.

- Callbacks are also used by many JS libraries.
  - For events as described above.
  - For for-each type functionality.
  - To deal with the result of a slow process, without forcing your code to wait while it happens. (e.g. network access)

DOM

- We discussed the DOM (Document Object Model) last week.
  - A language-independent class hierarchy for XML documents.
- As I said, it's central to JavaScript
  - … since JS lives inside a web page, the DOM is an obvious way to access the page from your code.
- There is a global object `document` that is the DOM document object for the current page.
- For example, this gets an array of DOM element objects for each paragraph on the page:

```
var paragraphs = document.getElementsByTagName("p");
```

- This will insert a new paragraph at the end of the page:

```
var body = document.getElementsByTagName('body').item(0);
var new_p = document.createElement('p');
new_p.appendChild(document.createTextNode("I'm some text"));
body.appendChild(new_p);
```

- e.g. the April Fools thing is basically DOM manipulation: find each text block, break it up so each character has its own `<span>`, every few seconds select one at random and animate it. Uses jQuery to make some of that easier.

Events

- JavaScript programming is fundamentally event driven (事件驅動程式設計).
  - That is, your code doesn't do teh basic flow of control.
  - The browser is in charge, and calls your code whenever an event occurs (that you have registered a callback for).

  - *Event-driven programming [enwiki]*
  - 事件驅動程式設計 *[zhwiki]*
  - *DOM Events*
  - *DOM event reference*

- Some events in JS+DOM you can react to:
  - An input gets/loses focus. (`focus`/`blur`)
  - An input's value changes. (`change`)
  - Something is clicked (`click`)
  - Something is clicked-and-dragged (`dragstart`/`dragend`/…)
  - Keys are pressed (`keypress`/`keyup`/`keydown`)
  - Page has finished loading (`load`)

    - Window is resized (`resize`)

- You can register code to an event in several ways.
    - In an HTML attribute: `<button onclick="callback();">`
    - As a DOM property: `button.onclick = callback;`
    - As a DOM property: `button.addEventListener('click', callback}, false);`
    - With something your JS library provides to make those easier.

- Any of those can be anonymous function definitions too.

```
some_form_input.change = function() {
  console.log(this.value);
  …
};
```

    - A good use for anonymous functions: that function didn't need a name. We just needed to register it as the callback for that event.
    - Or is it? Is this more readable?

```
function form_input_change_handler() {
  console.log(this.value);
  …
}
some_form_input.change = form_input_change_handler;
```

    - The right answer is probably both: sometimes the anonymous function defined right there is more readable; sometimes the function's name helps clarify what is going on.
    - The anonymous callback is definitely common JS style.

- Event callbacks are definitely a useful place for closures.
    - You may have information available when defining the callback function that you need when handling the event.
    - In non-event-driven programming, you'd just demand that info be passed as an argument.
    - But events have a fixed argument list: you can't add to it.
    - So, it's common to use a closure to get the info you need into the callback.
    - Something like:

```
function register_change_callback(element_id, to_upda
  var e = document.getElementById(element_id);
  var to_update = document.getElementById(to_update_i
  e.onchange = function() {
    /* modify the thing with id="to_update_id" */
    to_update.appendChild(…);
  };
}
register_change_callback("some_input", "some_status_e
register_change_callback("other_input", "other_status
```

Server Interaction and AJAX

- One of the tricks JavaScript can do: make an HTTP request.
  - This opens up a huge variety of techniques: your code can contact the server to get/send information whenever it wants, and update the page as a result.

- Techniques that use server interaction are often called AJAX (Asynchronous JavaScript and XML).
  - That doesn't describe a particular technology, but a collection that are often used together.
  - Actually, the term doesn't really mean much of anything. Mostly “web things we thought were cool from 2005 to 2008”.
  - Compare “Web 2.0“ which is a synonym for the period 2008–2011, and “HTML5“ which describes all cool things from 2011 to present.

  - *AJAX [enwiki]*
  - *AJAX [zhwiki]*
  - *XMLHttpRequest [enwiki]*
  - *Same origin policy [enwiki]*

- But the pieces of the acronym AJAX are somewhat meaningful:
  - Asynchronous: the server can be contacted at any time, not just when a page refresh happens.
  - JavaScript: the language we're using.
  - XML: a way to serialize the data being sent in either direction. JSON is much more common in modern uses.

- The `XMLHttpRequest` object is the key to all of it: it lets your JavaScript code make and HTTP request and process the results.
  - Realistically, you won't `XMLHttpRequest` directly, but will use a JS library to make things easier.
  - I'll switch to jQuery for the examples.

- `XMLHttpRequest` requests are restricted by a same origin policy.
  - Requests can only go to the same domain (and protocol and port) as the page itself.
  - Prevents DDOS attacks: everybody that goes to a malicious site makes one request a second to someone they don't like.
  - But can occasionally trap a legitimate use, and you have to work around.

- Since network calls can take some time, your code doesn't wait for the response.
  - You give a callback function that will be called when the request is complete (or there is an error, or whatever).
  - For example,

```
jQuery.ajax("some_json_data_url", {
  dataType: 'json',
  success: function(data, status, xhr) {
    console.log(data);
    console.log(data['one']);
  },
});
```

    - Another good use of an anonymous function: the code reads in the order you think of it running, so it's fairly readable. Naming the function wouldn't add much.

- An AJAX request can be used to fetch data from, or send data to the server (or any combination). Example uses:
    - Update the page with new info.
    - Build the page from fragments generated by the server.
    - An autocomplete input that gets possible values from the server.
    - An autosave of a partially-complete form.
    - Cooperation between server and client to calculate data for display.

- It is common to want to get real-time updates from the server.
    - That is, have the server send new data whenever it is available.
    - Traditional use of `XMLHttpRequest` requires polling the server every few seconds for updates: okay but not exactly what you'd want.
    - There are several technologies that try to deal with this: Long-Polling, WebSockets, Server Sent Events, and Comet.
    - All do basically the same thing: let the request from your JS code to the server stay open, so the server can send data whenever it has it.
    - Problem 1: Everyone visiting your site is going to have a connection to your server open all the time. Most web servers aren't good at maintaining thousands of open connections: they want to respond and close the connection as soon as possible.
    - Problem 2: The server-side programming model is usually “how do I respond to a request” not “some new data has arrived, who should I send it to?” It's going to require a big change, at least for the logic that does the push.
    - Problem 3: The libraries that support doing this don't seem particularly mature.

    - *What are Long-Polling, Websockets, Server-Sent Events (SSE) and Comet?*

JavaScript Performance

- JavaScript shouldn't run quickly.
    - The language wasn't designed with performance in mind: can't pre-allocate variables, must do garbage collection, etc.
    - Being a dynamic language should make it slow.
    - Since `a+b` might be string concatenation or addition, the language has to decide at runtime what to do.
    - That adds overhead. (C can decide when it compiles, so execution should be faster.)

- That is the world I learned about, and most programmers (and professors) still think they live in.
    - Dynamically-typed, interpreted languages are slow, but more fun to write.
    - Statically-typed, compiled languages are fast, but less productive.

    - They were right 10 years ago.
    - They are very wrong now.

- But in a modern browser, JS is fast.
    - Amazingly fast.
    - There has been a browser war over who can run JS code the fastest.
    - The winner? Us.

    - *The BIG browser benchmark [Nov 2012]*
    - *By the Numbers: the Fastest Browser [Jan 2013]*

- The benchmarks don't really tell the interesting story.
    - Do you know what a score of 4133 on the Kraken benchmark means? Me neither.
    - How does JS speed compare to equivalent Python code? Or Java? Or C?

- The answer depends a lot on the computation you're doing: there is no right answer.
    - On one sample numerical calculation I use as a cheap benchmark: Chrome performs only 15% slower than C compiled with `gcc -O2`, and faster than C without compiler optimization.
    - On the same benchmark, Python run with PyPy is about the same: 15% slower than optimized C.
    - Minimally-modified Python run with Cython is the exact same speed as C.

    - *Greg's language comparison*
    - *Computer Language Benchmarks Game*

- JavaScript (and Python and Haskell) aren't crazy languages to consider for numerical computations in 2013.
    - Any time somebody says “that language is slow”, they're speaking of the past.
    - Saying “that compiler is slow” might be valid, and a language might not have a good compiler.
    - But Just-In-Time compilers have made it so every language can be fast. It's just a question of how good the compiler is.

- Back on topic…
    - The JIT compilers in Chrome and Firefox are really, really good.
    - IE isn't horrible. IE10 is supposed to be better.
    - I haven't seen much about JS on phones, but they share code with the desktop browsers, so should be fairly fast.

- So you can actually write JS code that does some work, and it will happen fairly quickly on a modern browser.
    - Within reason, of course. Don't write anything that grinds your users' browsers to a halt.

Recent JS Additions

- The HTML5 specification (and some related documents) includes some new JavaScript APIs (应用程序接口) that browsers should implement.
  - These are intended to let developers create richer applications in the browser, not just web pages with some simple behaviour.
  - Nice on the desktop, but seems even more useful on mobile devices: can create a cross-platform mobile app without targeting each platform separately.

  - *HTML5 differences: APIs*
  - *HTML5: new APIs [enwiki]*
  - *caniuse.com: JavaScript APIs*
  - *Dive Into HTML5: offline web apps*
  - *HTML5 Cross Browser Polyfills*

- Some examples:
  - The `<canvas>` element: 2D bitmaps that can be created/changed by JS
  - Offline web apps: ability to create a collection of files that can be cached in the browser, manipulated with JS, and can contact the server when online.
  - Drag-and-drop events.
  - Web storage: a key-value store in the browser that JS code can use to store information so it doesn't have to be calculated/downloaded again.
  - Geolocation: where on the planet is this browser?
  - File API: for accessing `<input type="file">` stuff on the client-side.

- Compatibility varies, of course. Not all are practically-usable now.
  - General rule: many features available on every browser people use except IE ≤7.
  - Some incompatibilities can be worked around by JS libraries that do something compatible if it's not already in the browser.
  - Notably, Modernizr and its polyfills.

- But note: compatibility on smartphone/tablet browsers is very good.
  - Only exception seems to be Opera Mini (not Opera Mobile), which is stretching “smartphone” browser anyway.
  - If you're targeting mobile, then many of these are definitely practical features.

Progressive Enhancement

- (渐进增强) or graceful degradation (优雅降级) if you're a pessimist.

- As always on the web, the problem is: you don't know the exact situation you're sending your page to.
  - Modern desktop browser, smartphone browser, older-phone browser, Google indexer, other automated tool, …

- We need to structure the page so that any of these can make sense of the content.
  - … as best possible with the capabilities they have.

  - *Progressive enhancement [enwiki]*
  - *Understanding Progressive Enhancement*

  - e.g. Google only cares about content, so it only downloads your HTML: it should be able to get useful info out of just that.
  - e.g. Chrome should be able to do all of the JS stuff you want.

- *Progressive Enhancement with CSS*
- *Progressive Enhancement with JavaScript*
- *Graceful Degradation & Progressive Enhancement*

- The underlying structure of the web helps here:
  - Content in (X)HTML, with as much semantic meaning as possible: everybody should be able to handle this.
  - Style with CSS, using media selectors to turn on/off as appropriate.
  - Behaviour with JS.

- Various web features have been designed to degrade gracefully
  - For example, the default for an `<input>` is `type="text"`.
  - If you use the new `<input type="date">` in a browser that doesn't support it, it will just look like a text input.
  - On browsers that support it, you get a nice date selector, including something media-appropriate on a mobile or other non-desktop browser
  - You can use JS to enhance it: in browsers where your JS library can build a date selector, turn it on. (I bet there's a jQuery plugin to do this automagically.)
  - In other browsers, the user gets a text input, which is okay.
  - Of course, you must validate the date server-side anyway, and do something useful if it's in a bad format.

- But you have to code your sites with progressive enhancement in mind.
  - You should have an idea of what HTML/CSS/JS features are going to work in various browsers.
  - Think about what happens to the site if individual features don't work.
  - Test in as many different situations as possible.

- For JavaScript development: should generally start with a functional page/site that can be used without JS.
  - The JS code can then modify that page to make it work the way you want.
  - May involve removing/replacing some of the content in the HTML + CSS.
  - Don't use JS to do what could be done with HTML + CSS.

JavaScript Frameworks

- JavaScript itself is basically just a programming language
  - It's not perfect.
  - There are plenty of opportunities for libraries to make things better or extend it in useful ways.
  - There are many common problems that aren't solved by built-in language features, but you don't want to solve yourself.

- *The pros and cons of developing a complete JavaScript UI*
- *Journey Through The JavaScript MVC Jungle*
- *The Top 10 Javascript MVC Frameworks Reviewed*

- There are dozens of often-used and actively-maintained libraries. They all do different stuff, but fall into some categories:
  - Basics: DOM simplification, AJAX, events, form widgets, JSON, types/collections/programming basics.
    - These are really just collections of tools that fix things people don't like about JS and have to do often.
    - They usually hide some of the browser differences, so your code doesn't have to deal with them.
    - Many are extensible, so plugins can add much of the functionality found in bigger toolkits.
    - e.g. jQuery, prototype.js, Mochikit
  - MVC Frameworks.
    - The idea is to have models that reflect data on the server, and views to create/view/update/delete that data.
    - Probably includes some kind of templates for the views.
    - With one of these, you're probably building most of your app in client-side code. The server might only be a REST API to manage the security checks and data.
    - e.g. Backbone.js, AngularJS, Ember.js, Meteor
  - Full GUI
    - These help you create full graphical interfaces in the browser: windows, menus, drag-and-drop, etc.
    - e.g. ExtJS, Cappuccino
  - Others
    - Others that do one job, but do it well.
    - e.g. Google WebFont Loader, MathJax

- Of course, none of these categories is perfectly well defined. Many of those do things from the other categories, especially with plugins.

- Should you use them?
  - Doing basic DOM, AJAX and simple things is painful in JS without some help: if you're doing any JS you probably at least want something like jQuery.
  - The more the toolkit does, the bigger it's going to feel.
  - I'm not convinced that writing everything client-side is a good idea. It's not going to degrade gracefully if your code ends up in a situation you or the toolkit makers haven't tested.

Server-Side JavaScript

- Many developers notice that JavaScript has a lot of things in its favour:
  - Generally nicely designed language.
  - Very fast execution by modern just-in-time compilers.
  - Anybody doing web development probably already knows it.

- So, maybe it makes sense to use on the server as well.

    - Only one language for the logic of your project: easier management.
    - Can re-use code on both sides. (e.g. form validation code: write function to check for valid input. Call on client in UI, and server for data integrity/security checks.)

    - *Comparison of server-side JavaScript solutions [enwiki]*
    - *Node.js [enwiki]*

- This is possible if the JavaScript engine can be extracted from the browser and run standalone.
    - This is possible: V8 from Chrome, SpiderMonkey from Firefox, Rhino also developed by Mozilla.

- Most notable project for server-side JS is Node.js.
    - Essentially a web server (probably meant to be run behind Nginx or another load balancer) that calls your JS code to handle requests.
    - Node.js is particularly good at managing many simultaneous connections. This allows the long polling where the client keeps the connection open and can be pushed updated info any time.

- My opinion:
    - JavaScript is nice, but we use it on the browser because we have no other choice.
    - I have the choice of any language in the world on my server. Is JS really my favourite? Probably not.
    - But the ability to share logic in both client and server code is **really** tempting.