Mobile Application Development (COMP2008)

# Lecture 8: Web Development (Part 2)

Updated: 25th October, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

# Outline

Client-Server Communication

Fetch (and Promises)

TypeScript

# JavaScript: Client-Server Communication

- Just as for mobile apps, web apps need client-server communication.
  - Indeed, they generally need it *more*.
  - For web apps, client & server are two parts of the same application.
- Page reloading was originally the only way of doing this.
  - Send the user to a different page on the same server.
- Web apps now (also) use "asynchronous" loading.
  - Stay on the same page.
  - Communicaton happens in the background, without the user (necessarily) knowing about it.

# Same Origin Policy

- Web browers impose restrictions on if/how a page can contact servers.
- A web app is *always* allowed to contact the same server it was downloaded from (its "origin").
  - Say the user went to `https://example.com/mywebapp/login`.
  - The browser downloaded HTML and JavaScript from that address.
  - The HTML/JavaScript may then access any URL starting with `https://example.com/`.
  - We often use a "*relative URL*" for this, omitting the server name; e.g. `/mywebapp/thedata.json`.
- Accessing *other* URLs is called a "cross-origin" request.
- Generally *not* allowed, except in certain specific cases. e.g.:

```
<script src="https://code.jquery.com/jquery-3.3.1.js">
```

# Page Reloading With Pure HTML

- The HTML <form> element was a simple way of reloading pages; e.g.

```
<form action="/path/to/submitpage.aspx">
    <p>Your name: <input name="name" type="text" /></p>
    <p>Your age:  <input name="age"  type="text" /></p>
    <p><input type="submit" value="Upload Details" /></p>
</form>
```

- Here, the user fills out their name and age.
- `<input type="submit" ... />` creates a special button.
- When pressed, the browser will load a new URL, created by adding together:
  - The initial part of the current URL (e.g. `https://example.com`).
  - The pathname given in `action="..."`.
  - The values entered into the form (as query parameters).

# Page Reloading With JavaScript

- In JavaScript, you can trigger a form submission programmatically:

```
let form = document.getElementById("theform");
...
form.submit();
```

- You can also arrange for an event handler to *validate* the user's input before submission.

# Page Reloading Issues

- Page reloading is simple...
- ...But means the entire app (on the client-side) must be reloaded.
- Can create frustrating delays for the user.
- We also need to save and restore the state (i.e. all the data).
  - Limited options for local storage ("cookies").
  - Often apps will send their state to the server.
  - The server passes it straight back as part of the page reload, embedded in the new set of JavaScript code.
  - This works, but is rather inefficient.

# Asynchronous Communication

- Communication *without* page reloads is known as "AJAX".
  - "Asynchronous JavaScript And XML"
  - Good concept, terrible name.
  - The XML is optional. We often use JSON instead.
- The basic idea:
  1. Your JavaScript sends a request to the server.
  2. This is handled in the background. The client-side app keeps running.
  3. The server eventually responds with some data (XML/JSON/etc).
  4. The browser calls your callback function (which you set up beforehand).
  5. Your callback function works out how to display the data (or otherwise deal with it).

# XMLHttpRequest (XHR)

- AJAX was first done with a JavaScript/browser construct called `XMLHttpRequest` (XHR for short).

```
let url = "...";
let req = new XMLHttpRequest();

req.onload = function() // Callback (event handler)
{
    if(req.status == 200) // Request succeeded?
    {
        // We now have the raw response data.
        console.log(req.responseText);
    }
    else { ... } // Failed -- display error.
};

req.open("GET", url); // Build and send the request.
req.send();
```

# XMLHttpRequest: Discussion

- There have been two versions of `XMLHttpRequest`. In the original version:
  - We used a different event handler: "onreadystatechange".
  - This was called four times; only the fourth call meant the server had actually responded.
- In any case, you must also check that the request succeeded (`req.status == 200`).
- jQuery also provides a wrapper around `XMLHttpRequest` called "`$.ajax()`".

# Fetch and Promises

- XMLHttpRequest is now deprecated in favour of the "`fetch()`" function.
- This does the same thing, but tries to be more readable.
  - Makes use of another JavaScript object called a "promise".
- Why? Asynchronous operations make for messy code.
  - Callbacks, by definition, are executed "out of order" compared to their surrounding code.
- Chains of asynchronous operations make for *very* messy code.
  - Happens regularly in JavaScript – when one operation ends, you need to start another based on the result.
  - Lots of nested *and/or* out-of-order functions.
  - Throw in some decent error handling, and it's very difficult to see what's happening.

# Horrible Nested Callbacks (“Pyramid of Doom”)

Just to illustrate, abstractly:

```
obj.callback = function(obj2)                    // Step 1.
{
    obj2.callback = function(obj3)               // Step 3.
    {
        obj3.callback = function(obj4)           // Step 5.
        {
            console.log("Finished: " + obj4);    // Step 7.
        };
        obj3.asyncOp();                          // Step 6.
    };
    obj2.asyncOp();                              // Step 4.
};
obj.asyncOp();                                   // Step 2.
```

- Three asynchronous operations, one after another.
- But not in chronological order! Rather difficult to read.

# It Gets Worse: Error Handling in Nested Callbacks

- Asynchronous operations can potentially fail at any step.
  - The network could be down.
  - The server could fail to find the data you need.
  - The data could be invalid.
- Most of the time, it doesn't really matter *which* step fails.
  - The end result is the same: your app doesn't get the data.
  - So, you *should* ideally only need one error handler.
- In synchronous (normal) code, this is what `try-catch` does:

```
try { ... } // Complicated algorithm
catch(err) { ... } // Deal with any error
```

- Asynchronous code breaks exception handling.
  - You can handle errors, *but*...
  - ...each callback must handle its own errors individually.
  - No way to handle all errors in one place. *Until promises!*

# Promises

- The old convention (used by `XMLHttpRequest`):

```
obj.callback = function(obj2) {...};
obj.asyncOp(); // Calls the callback when done.
```

- Instead, we now have `asyncOp()` return a "promise" object.
  - The actual operation carries on in the background.
- The promise lets us specify the callback via a `then()` method:

```
obj.asyncOp().then(function(obj2) {...});
```

- `then()` returns *another* promise, so we can chain together asynchronous operations:

```
obj.asyncOp()
    .then(function(obj2) { return obj2.asyncOp(); })
    .then(function(obj3) { return obj3.asyncOp(); })
    .then(function(obj4) {
        console.log("Finished" + obj4); });
```

# Promises: Actual Order of Execution (1)

Let's be clear about *what* is actually happening *when*.

```
obj.asyncOp()
    .then(function(obj2) { ... })
    .then(function(obj3) { ... })
    .then(function(obj4) { ... });
```

1. asyncOp()
   - Starts an asynchronous task and returns straight away.

# Promises: Actual Order of Execution (2)

Let's be clear about *what* is actually happening *when*.

```
obj.asyncOp()
    .then(function(obj2) { ... })
    .then(function(obj3) { ... })
    .then(function(obj4) { ... });
```

## 2. then()

- Each then() function runs.
- They save their callbacks, effectively building up a chain.
- Important to realise that this happens separately from (and before) the callbacks themselves.

# Promises: Actual Order of Execution (3)

Let's be clear about *what* is actually happening *when*.

```
obj.asyncOp()
    .then(function(obj2) { ... })
    .then(function(obj3) { ... })
    .then(function(obj4) { ... });

... // Other tasks
```

## 3. Everything else in the current event handler

- Virtually everything in JavaScript will be inside one event handler or another.
- This goes for this example code as a whole.
  - e.g. the whole thing might be inside a button callback, or a timer callback.
- One handler *must* finish before any other can be run.
  - (This is all single-threaded.)

# Promises: Actual Order of Execution (4)

Let's be clear about *what* is actually happening *when*.

```
obj.asyncOp()
    .then(function(obj2) { ... })
    .then(function(obj3) { ... })
    .then(function(obj4) { ... });
```

## 4. Callbacks

- The first callback runs after the first asyncOp() task eventually completes.
- Each other callback runs after the previous task.
- Likely to be a delay from one to the next.
  - They're asynchronous after all.

# Promises and Error Handling

- Promises track errors, and allow you to handle them all in one place:

```
obj.asyncOp()
    .then(...)
    .then(...)
    .then(...)
    .catch(function(err)
    {
        alert("Something terrible happened: " + err);
    });
```

- This *isn't technically* `try-catch` exception handling.
  - `catch()` is a method call, which sets up another callback.
  - Though it's basically an asynchronous equivalent to `try-catch`.
  - If you do throw an exception from a previous callback, it will trigger the `catch()` callback.

# "Arrow Functions"

- Also now worth noting: JavaScript has an alternate (lambda) function syntax.
- Instead of this:

```
let callbk = function(obj) { return doSomething(obj); };
```

- We can write this:

```
let callbk = (obj) => { return doSomething(obj); };
```

- Or even this:

```
let callbk = (obj) => doSomething(obj);
```

- Or even this (if you have exactly one parameter):

```
let callbk = obj => doSomething(obj);
```

# Promise Chains and Arrow Functions

- So, we can simplify our promise chains like this:

```
obj.asyncOp()
    .then(obj2 => obj2.asyncOp())
    .then(obj3 => obj3.asyncOp())
    .then(obj4 => console.log("Finished: " + obj4))
    .catch(err => alert("Error" + err));
```

# Fetch

- So let's get back to client-server communication.
- `fetch()` takes a URL and (optionally) an options object.
- It returns a promise, and can be used like this:

```
let url = "...";
fetch(url, { method: "GET" })
    .then(function(response)
    {
        if(response.ok)
        {
            return response.json();
        }
        else { throw new Error(response.statusText); }
    })
    .then(function(data) { ... }) // Whoo -- data!
    .catch(err => alert("Fiddlesticks"));
```

# Fetch: Discussion

- On the previous slide, there were two asynchronous operations:
  - `fetch()` itself, and `json()`.
- When the first callback runs:
  - The server is respond*ing*.
  - We do know the return status – 200 ("ok"), 404, etc.
  - We don't yet have the complete data.
- `json()` returns another promise, which gets matched up to the 2nd callback.
- The 2nd callback runs once all the data is received.
  - Courtesy of `json()`, it receives an object representing all the parsed JSON data.
  - There's also `text()`, which works the same way but gives you a raw string instead.
- We only need one error handler.
  - Throwing an exception from any callback will trigger the error handler.

# TypeScript

- JavaScript has shortcomings.
  - *Apart* from the optional semicolon, and the crazy operators.
- Dynamic typing means you don't get type safety.
  - No way to enforce the datatype of a variable, parameter or return value.
  - No way to ensure, up-front, that an object has the right fields/methods.
- You have to target the "lowest common denominator".
  - Not everyone has the latest web browser.
    - (Though with automatic updates this is better than it has been historically.)
  - You can't use language/API features that are missing from your users' machines.
- "TypeScript" fixes both of these.

# TypeScript: What Is it?

- Roughly speaking, it's a "superset" of JavaScript.
- Provides optional static typing; e.g.:

```
function format(name: string, age: number): string
{
    return name + ": " + age;
}

let details: string = format("Slartibartfast", 5000000);
```

- Compiles into "old" JavaScript.
  - Shiny new language features are translated into old, ugly but universally-supported ones.
  - You get to *use* new language features.
  - Your users' browsers don't have to support them.
  - Best of both worlds!

# TypeScript Types

- All valid JavaScript code *ought* to be valid TypeScript too.
  - This means that typing is optional.
  - If you specify types, the compiler will enforce them.
  - But you may decide, in some cases (e.g. local variables), that the code is clear enough without them.
- TypeScript has a type system that puts a lot of other languages to shame.
  - It has the same basic types as JavaScript.
  - But more advanced ones too (which compile down into basic Java types).
  - Supports interfaces, enums, generics, maps, type aliases, etc.

# Interfaces

- TypeScript has interfaces.
- You can declare that a class implements an interface.
- *But*, they also apply *automatically* to all matching objects.
  - If an object has the same properties/methods as an interface, then by definition it implements the interface.
- Interfaces can represent *functions* too, as well as objects.
  - Functions are passed around a lot.
  - It's good to require *particular kinds* of functions in particular situations.
  - A function interface can specify what parameter/return types a function should have.

# Other TypeScript Types: Some Examples

- **Union types:** for having one of several specific types:

```
function f(value: string | number) {...}
```

  You can also say whether values are allowed to be `null` or not:

```
function f(x: string, y: string | null) {...}
```

  - The compiler can catch all your `NullPointerException`s!
- **Literal types:** for having one of several specific number or string values:

```
function f(colour: "red" | "green" | "blue") {...}
```

- **Tuple types:** for a sequence of values of specific (different) types:

```
let x: [number, string, boolean] = [5, "red", true];
```

- And others still!

# TypeScript: Playing

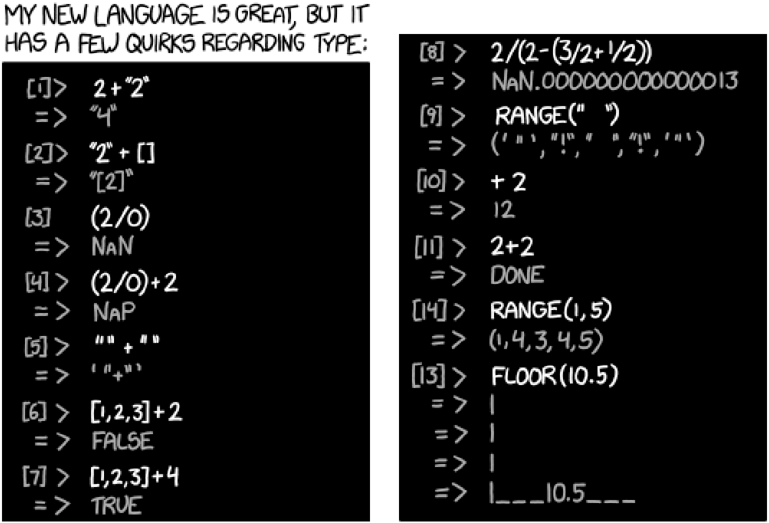
- You can play with TypeScript online:
  - http://www.typescriptlang.org/play/
  - https://jsfiddle.net/boilerplate/typescript
- You can also install it yourself:
  - First install the npm tool (normally part of NodeJS).
  - Use it to install the TypeScript compiler:

```
[user@pc]$ npm install -g typescript
```

  - Then you can compile/transpile your code as follows:

```
[user@pc]$ tsc mycode.ts
```

    - This will generate `mycode.js`, which is valid JavaScript.




(https://xkcd.com/1537/)