SENG 365 Week 1

Intro to HTTP and JavaScript

What is a web application?

What is a web application?

What makes an application a web application rather than desktop application or a mobile application, or an embedded application, or …?

What is a web application? Consider…

- TradeMe
- Gmail running in Chrome or equivalent
- Gmail app running on Android or iOS
- Facebook running in a browser
- Office 365
- DropBox, Google Drive, …

It’s 2005…

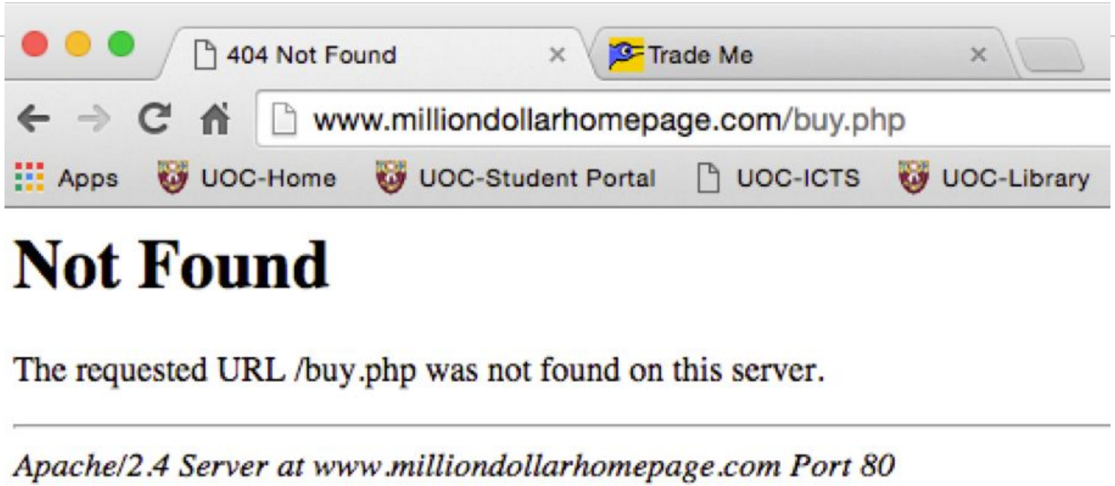
The iPhone wasn’t launched until 2007.

V for Vendetta was showing at cinemas …

… Star Wars: Episode III – Revenge of the Sith

The Million Dollar Homepage™
1,000,000 pixels • $1 per pixel • Own a piece of internet history!
SOLD OUT!
Follow @tewy
13K followers
Homepage | Buy Pixels | FAQ | Blog | Pixel List | Press | Testimonials | Contact me

php!



Not Found

The requested URL /buy.php was not found on this server.

Apache/2.4 Server at www.milliondollarhomepage.com Port 80

but what's this?
/buy.php
An API endpoint...?

... and now

The internet fad of 2005 now stands as a stark demonstration of '**link rot'** & **'system decay'**.
22% of the links were dead by 2014

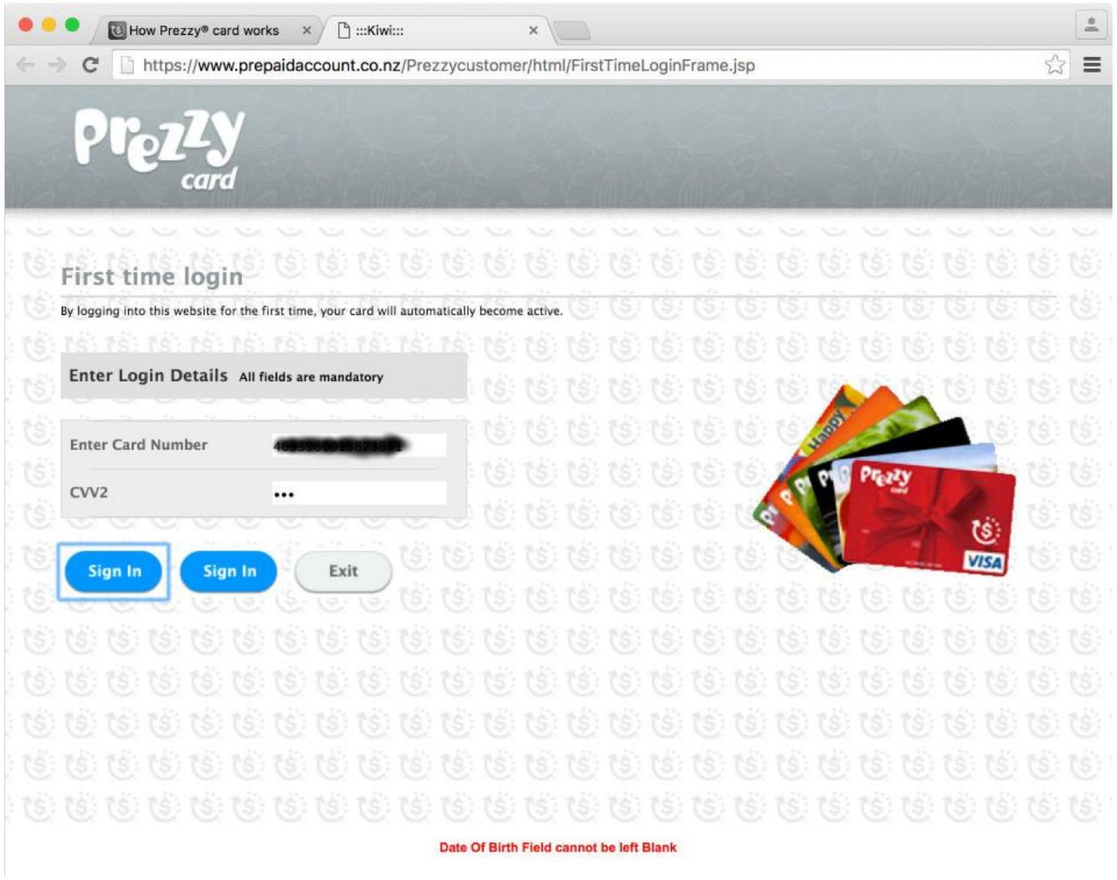
http://www.theguardian.com/technology/2014/mar/27/after-nine-years-the-million-dollar-homepage-dead (March 2014)

For a history of the site, see Wikipedia:

https://en.wikipedia.org/wiki/The_Million_Dollar_Homepage

1

Why take a course on web application architecture?



How Prezzy® card works

:::Kiwi:::

https://www.prepaidaccount.co.nz/Prezzycustomer/html/FirstTimeLoginFrame.jsp

First time login

By logging into this website for the first time, your card will automatically become active.

Enter Login Details All fields are mandatory

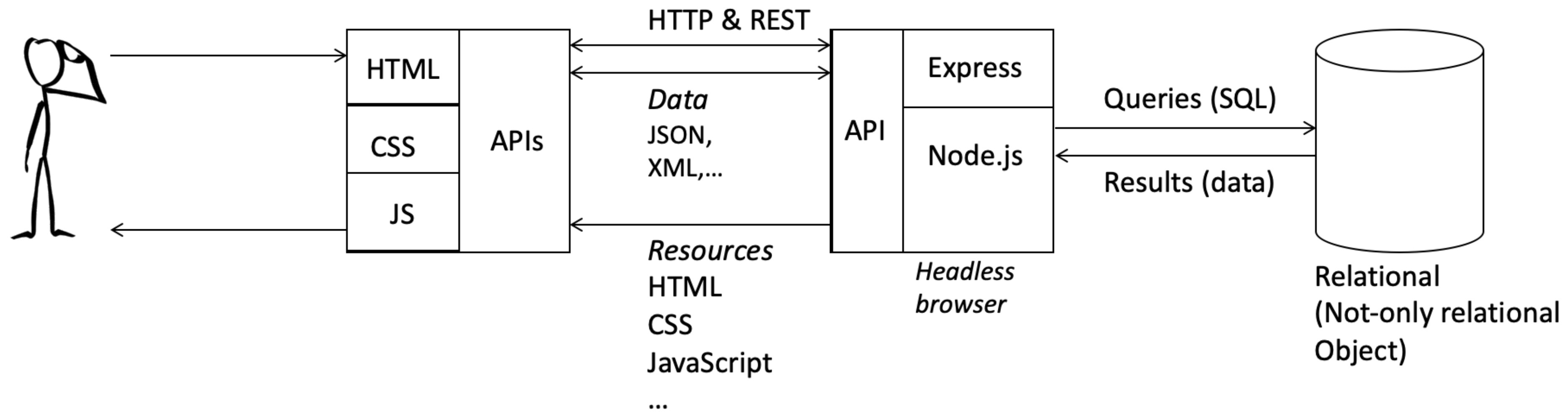
Enter Card Number	[redacted]
CVV2	•••

Sign In Sign In Exit

Date Of Birth Field cannot be left Blank

Challenges for modern web applications

- **Consume services** from another system
- **Provide services** to another system
- **Modularize** my application (to manage complexity)
- Respond to multiple **overlapping (asynchronous) requests**
- Make changes **persistent** (for large, distributed systems)
- Allow and restrict **user access** (security and privacy)
- **Display information** from a source
- **Synchronize information** shown on different views
- Maximize **responsiveness**
- **Adapt** to different devices and screen sizes
- **Protect user data** from being harvested
- Protect my business from harm (**prevent exploits**)



User	HTTP client	HTTP Server	Database
Human	Machine	Machine	Machine

Reference model

2

Course administration

Teaching team, Course requirements, Assignments

Teaching team

Ben Adams

Lecturer and Course Coordinator

Erskine 310

benjamin.adams@canterbury.ac.nz

Moses Wescombe

Tutor

Frederik Markwell

Tutor

Morgan English

Senior Tutor

Erskine 324

morgan.english@canterbury.ac.nz

Overview of lecture topics

Term 1

- **Week 1 & 2:** HTTP, JS & asynchronous flow
- **Week 3:** TypeScript and Data persistence
- **Week 4:** HTTP Servers and APIs
- **Week 5:** GraphQL, API Testing
- **Week 6:** Security, Client-side basics

Term 2

- **Week 7:** Single Page Applications
- **Week 8 & 9:** React
- **Week 10:** Communication with server, performance
- **Week 11:** Web storage, Progressive web apps
- **Week 12:** Testing, Review

Assessment

The Assessments

- **Assignment 1** (30%)
  - No extension
- **Assignment 2** (30%)
  - No extension
- **Final Exam** (40%)
  - 2 hours

Additional information and requirements

- Assignment resources on Learn
- API specification with skeleton project
- Infrastructure
  - eng-git project
  - MySQL database
  - Postman tests

Assignment 1 - API Server

HTTP server + application

- HTTP request & response cycle
- URL e.g. protocol, path, endpoints, query parameters
- HTTP headers and body
- Headers: e.g. Cookies
- Headers: e.g. CORS
- Body e.g. JSON data
- HTTP methods e.g. GET, PUT, DELETE
- HTTP status codes e.g. 201, 404

And also

- Authentication and authorization
- Asynchronous requests
- Database connectivity
- Conform to API specification
  - You will be given an API specification to implement

Assignment 2 - Client front-end

HTTP client

- HTML + CSS + JS app
- Modern browser
- Implementing user story backlog

And also

- Authentication and authorization
- Asynchronous requests
- RESTful API calls

Labs

Term 1

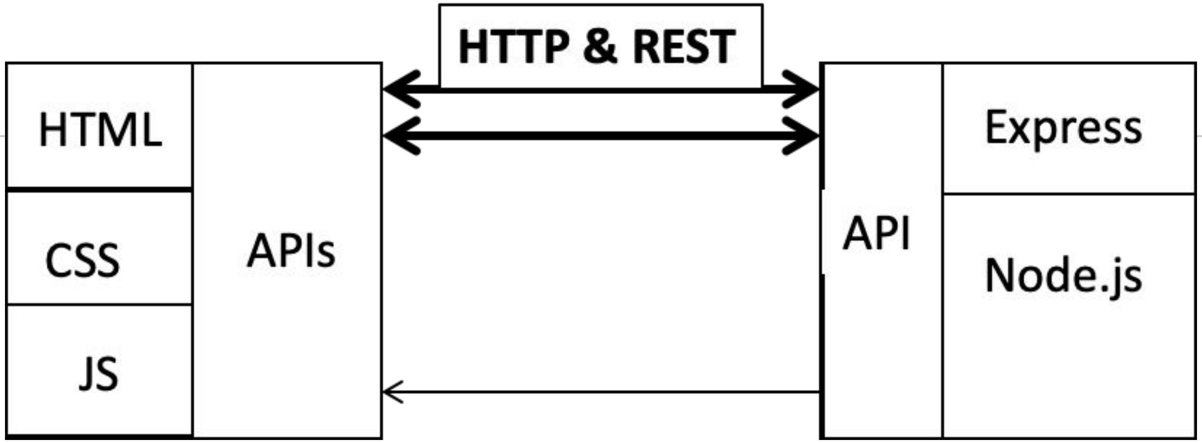
- **Week 1:** 3 x pre-labs (self-study)
- **Week 2:** Lab 1
- **Week 3 & 4:** Lab 2
- **Week 5 & 6:** assignment support

Term 2

- **Week 7:** Lab 3
- **Week 8:** Lab 4
- **Week 9:** Lab 5
- **Week 10 & 11:** assignment support
- **Week 12:** assignment 2 testing **(attendance mandatory!)**

The HTTP protocol

3



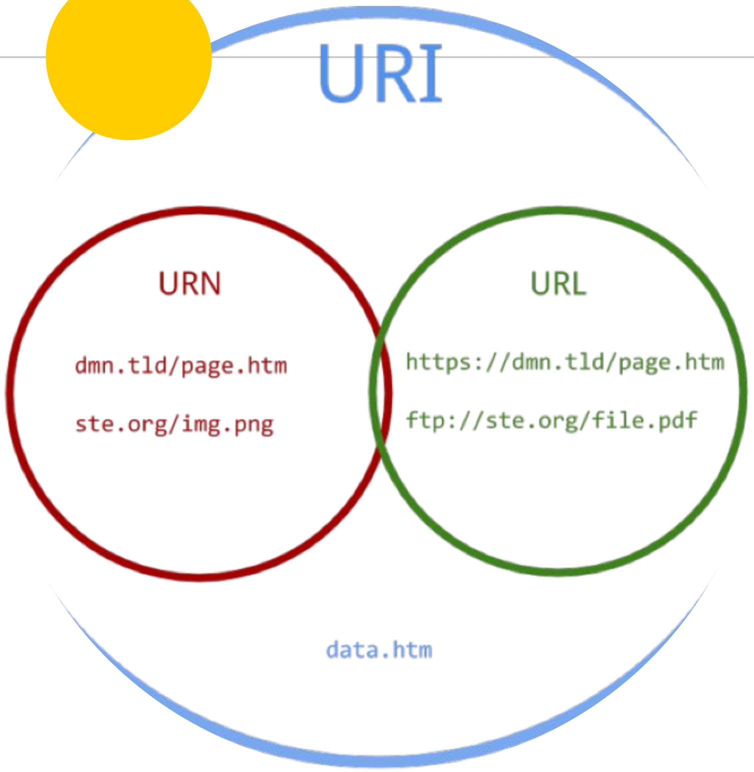
Overview to HTTP

HTTP messages are how data is exchanged between a server and a client. There are two types of messages: requests sent by the client to trigger an action on the server, and responses, the answer from the server.

HTTP messages are composed of textual information encoded in ASCII*, and span over multiple lines. In HTTP/1.1, and earlier versions of the protocol, these messages were openly sent across the connection.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

(* There's a bit more to it than just ASCII…)



Uniform Resource Identifiers

- **URI (Uniform Resource Identifier)**

  String of characters to identify (name, or name and location) resource
- **URL (Uniform Resource Locator)**

  A URI that also specifies the means of acting upon, or obtaining representation. That is, a URI with access mechanism and location
- **URN (Uniform Resource Name)**

  Deprecated: historical name for URI.

Your server needs to handle URLs like:

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

The protocol

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

The domain name

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

The port

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

- The default **HTTP** port is **80**
- The default **HTTPS** port is **443**
- The port on which your HTTP server listens for requests is a different port to the port to which the server issues queries to the MySQL database (default: 3306)

The path

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

- The path is increasingly an abstraction, not a ‘physical path’ to a file location.
- A path to an HTML file is not (quite) the same thing as the path to an API endpoint
- An API endpoint uses the standard URI path structure to achieve something different
    - In particular, parameter information
- The path may need to include information on the version of the API

An example API endpoint path

...80/api/v1/students/:id?key1=val1...

- The path contains versioning: api/v1/
- There’s an API endpoint: **students**
- The path contains a variable in the path itself: :id
    - How are these path variables handled by the server...?
- You may still pass query parameters: ?key1=val1
    - Given the path variable, query parameters may be redundant for the endpoint
    - There is also the body of the HTTP request for passing information

Query parameters and other parameters

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

The API could be designed to accept parameter information via

- The URI’s ? query parameters
- The URI’s path (see previous slide/s) e.g. :id
- The body of the HTTP request e.g. JSON
- Or via some combination of the above
- What goes in query parameters, in the path, and in the body?

# anchors

http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere

- Anchors used as ‘bookmarks’ within a classic HTML webpage
  - i.e. point to a ‘subsection’ of the page
- We don’t need to use anchors for our API requests
  - (Being creative, you might…?!?)

HTTP Headers

Distinguish between

- **General headers** required
- **Entity headers** (apply to the body of the request)

And between

- Request headers
- Response headers

Cookies are implemented in/with the header

- Set-Cookie: <...> (in the header of the server's HTTP response)
- Cookie: <...> (in the header of subsequent client HTTP requests)

Use headers to

- Maintain session
- Personalise
- Track (e.g. advertising)

Structure and example of HTTP requests

HTTP requests are of form:

```
HTTP-method SP Request-URL SP HTTP-Version CRLF
*(Header CRLF)
CRLF
Request Body
```

Example GET (no body):

```
GET /pub/blah.html HTTP/1.1
Host: www.w3.org
```

Example POST (with indication of body):

```
POST /pub/blah2.php HTTP/1.1
Host: www.w3.org

Body of post (e.g. form fields)
```

KEY:

```
SP   = space
CRLF = carriage return,
   line feed (\r\n)
```

HTTP verbs

- GET
- PUT
- POST
- DELETE
- HEAD
- Others

Structure and example of HTTP responses

HTTP responses are of form

```
HTTP-Version SP Status-Code SP Reason-Phrase CRLF
*(Header CRLF)
CRLF
Response Body
```

Typical successful response (GET or POST):

```
HTTP/1.1 200 OK
Date: Mon, 04 Jul 2011 06:00:01 GMT
Server: Apache
Accept-Ranges: bytes
Content-Length: 1240
Connection: close
Content-Type: text/html; charset=UTF-8

<Actual HTML>
```

Headers

Space

Body…

Response codes

1xx, informational (rare)

- e.g. 100 continue

2xx, success

- e.g. 200 OK, 201, Created, 204 No Content

3xx, redirections

- e.g. 303 See Other, 304 Not Modified

4xx, client error (lots of these)

- e.g. 400 Bad Request, 404 Not Found

5xx, server error

- e.g. 500 Internal Server Error, 501 Not Implemented

Brief examples in Node.JS

With **http** package and with **express** package

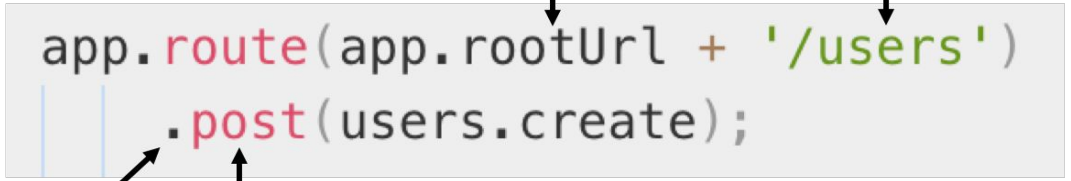
express ‘Listening’ for a request to an endpoint

Note:
Nothing stated about

- ports
- domain names
- query parameters

The root of the *path* in the URL

A simple API endpoint:
An ’extension’ to the *path* in the URL



```
app.route(app.rootUrl + '/users')
    .post(users.create);
```

dot (.)
Method chaining

The HTTP
method: POST

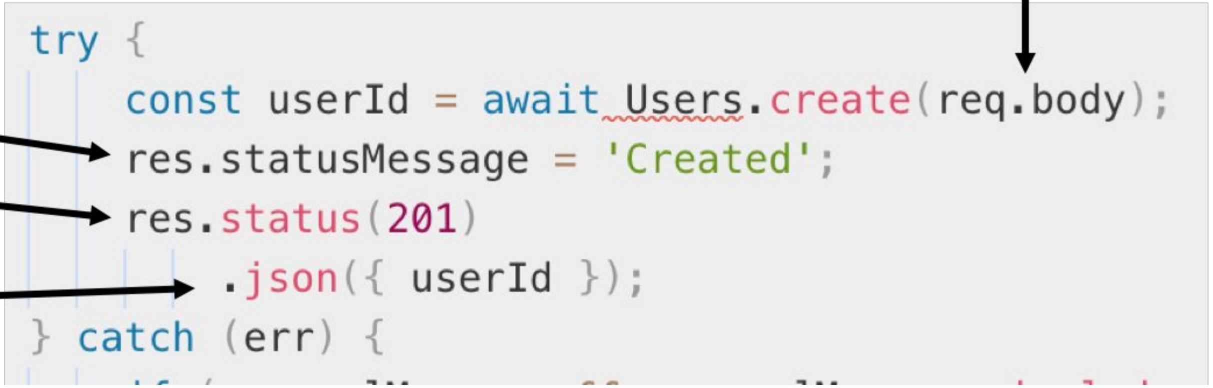
express Create a user and respond

Object containing content of request body

HTTP status message

HTTP status code (201)

Set HTTP headers and return JSON in body



```
try {
    const userId = await Users.create(req.body);
    res.statusMessage = 'Created';
    res.status(201)
        .json({ userId });
} catch (err) {
```

The body of the HTTP request

- Some HTTP requests (typically) **do not need a body**:
  - e.g. a GET request, a DELETE request
- Broadly, there are three categories of body:
  - **Single-resource bodies, consisting of a single file of known length**, defined by the two headers: Content-Type and Content-Length.
  - **Single-resource bodies, consisting of a single file of unknown length**, encoded by chunks with Transfer-Encoding set to chunked.
  - **Multiple-resource bodies**, consisting of a multipart body, each containing a different section of information. These are relatively rare.
- HTTP bodies can contain **different kinds of content**
  - We're going to be using **JSON** (because JSON is better than the others 😉)

4

JavaScript

The JavaScript way of programming

Highlights

- Objects, methods & functions
- Expressions, statements and declarations
- Functions
- Immediately invoked function expressions (IIFE)
- Scoping
- Variables
- Variable hoisting
- Closures
- this

Next week

- Method chaining (cascading)
- 'use strict'; mode
- Modularisation: export & require()
- Node.js
- Asynchronous (event) handling
- Callbacks, Promises, Async/Await

Objects and methods

- JavaScript is an **object-oriented** programming language
  - Not as strict as Java, in its definition of objects e.g. not compulsory to have classes
- An **object** is a collection of properties
- A **property** is an association between a name (or **key**) and a **value**.
  - A property can itself be an object.
- A **method** is a function associated with an object; or, alternatively, a method is a property (of an object) that is a function.

Functions

- Functions are **first-class objects**
- They can have **properties** and **methods**, just like any other object.
- Unlike other objects, functions **can be called**.
- Functions are, technically, **function objects**.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

Expressions, statements, declarations

- An **expression** produces a value.
- A **statement** does something.
- **Declarations** are something a little different again: creating things.
- BUT, JavaScript has:
  - **Expression statements**: wherever JavaScript expects a statement, you can also write an expression!
  - The **reverse does not hold**: you cannot write a statement where JavaScript expects an expression.

http://2ality.com/2012/09/expressions-vs-statements.html

Functions and their execution

Example 1

```
var result = function aFunction (){
        return -1;
};
```

What is the value of result?

Functions and their execution

Example 2

```
var result1 = function aFunction1 (){
        return -1;
};
var foo = result1();
```

What is the value of foo? Why?

Functions and their execution

Example 3

```
var result1 = function aFunction1 (){
        return -1;
}();
```

What is the value of result1?

Functions and their execution

Example 4

```
(function aFunction3 () {
    return 2;
})();
```

What is happening here? Why?

Digression: a pair of brackets ()

The pair of brackets, (), is:

- Used to execute a function e.g. function();
- The grouping operator, e.g., to force precedence (a + b) * c;

Immediately invoked function expressions (IIFE)

```
(function () {
        statements
})();
```

- The **outer brackets**, (function…)();, enclose an **anonymous function**.
- The subsequent **empty brackets**, (); **execute** the function.
- The anonymous function establishes a **lexical scope**. Variables defined in statements cannot be accessed outside the anonymous function

Uh oh: IIFE not executing

At the console I type this:

```
> function () {
        statements
}();
```

But this doesn’t work. Why?

But: These IIFEs are working, Why?

```
+function afunction () {
	console.log('Here I am!');
}();

!function afunction () {
	console.log('Here I am!');
}();
```

Scoping

Block scope (Java, C#, C/C++)

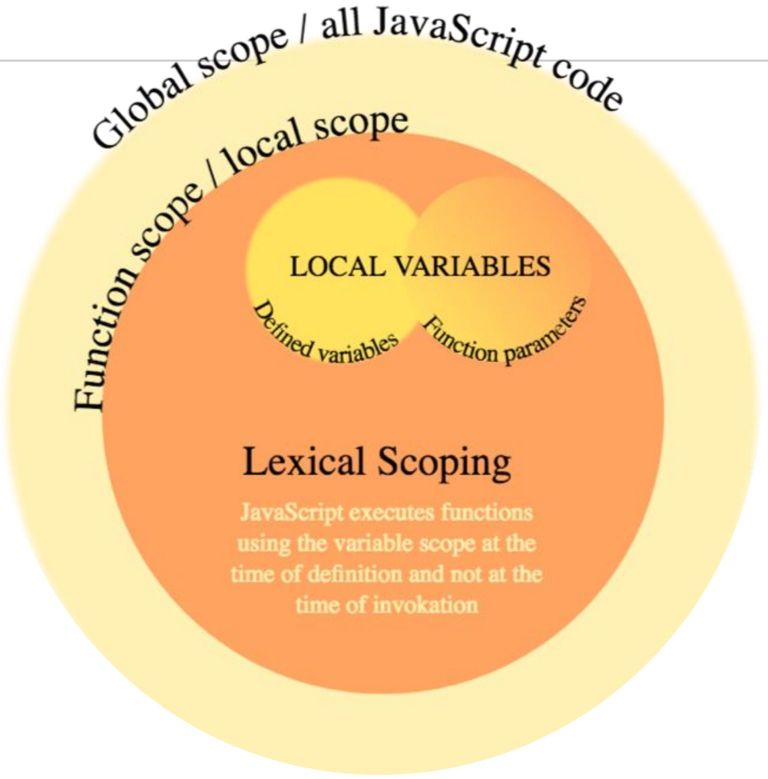
```
public void foo() {
    if (something){
        int x = 1;
    }
}
```

x is available only in the
if () {} block

Lexical scope (JavaScript, R)

```
function foo () {
    if (something) {
        var x = 1;
    }
}
```

x is available to the foo function (and any of foo's inner functions)


Global scope / all JavaScript code
Function scope / local scope
LOCAL VARIABLES
Defined variables
Function parameters
Lexical Scoping
JavaScript executes functions using the variable scope at the time of definition and not at the time of invokation

JavaScript functions

JavaScript executes any function:

- using the variable scope at the time of **definition of the function**
- **not** the variable scope at the time of **invocation of the function**

In other words:

- Did the variable exist at the time of definition, e.g., in an outer function?
- This approach to function execution supports **closures**.

Things have changed with ES6

Examples of JS variables

a = 1; //undeclared

var b = 1;

New in ES6

let c = 1;

const d = 1;

Undeclared variables shouldn't be used in code

But they can be, unless you use 'use strict';

Always declare a variable

var is lexically-scoped

let is block-scoped

const is block-scoped, and can't be changed

Variable hoisting

```
function foo () {
        // x hoisted here
        if (something) {
                let x = 1;
        }
}
```

Variable declarations in a function are hoisted (pulled) to the top of the function.

- *Not variable assignment*

Invoking functions before they're declared works using hoisting

- *Note: doesn't work when assigning functions*

Closures

When JavaScript executes a function (any function), it:

- uses the variables in-scope at the **time of definition** of the function
- **not** the variable scope at the **time of invocation** of the function
- a closure is a **record** storing a **function *together* with an environment**
    - Variables **used locally** but **defined in enclosing scope**

this needs careful attention

The context of any given piece of JavaScript code is made up of:

- The current function's (lexical) scope, and
- Whatever is referenced by this

By default in a browser, this references the global object (window)

By default in node, this references the global object (global)

this can be manipulated, for example:

- Invoke methods directly on an object, e.g. with foo.bar(); the object foo will be used as this

But this is fragile:

- let fee = foo.bar; // this=foo
- fee(); // this=global/window

More JS next week

Any questions ?