SENG 365 Week 2

More JavaScript and Asynchronous Flow

The story so far

In the lectures

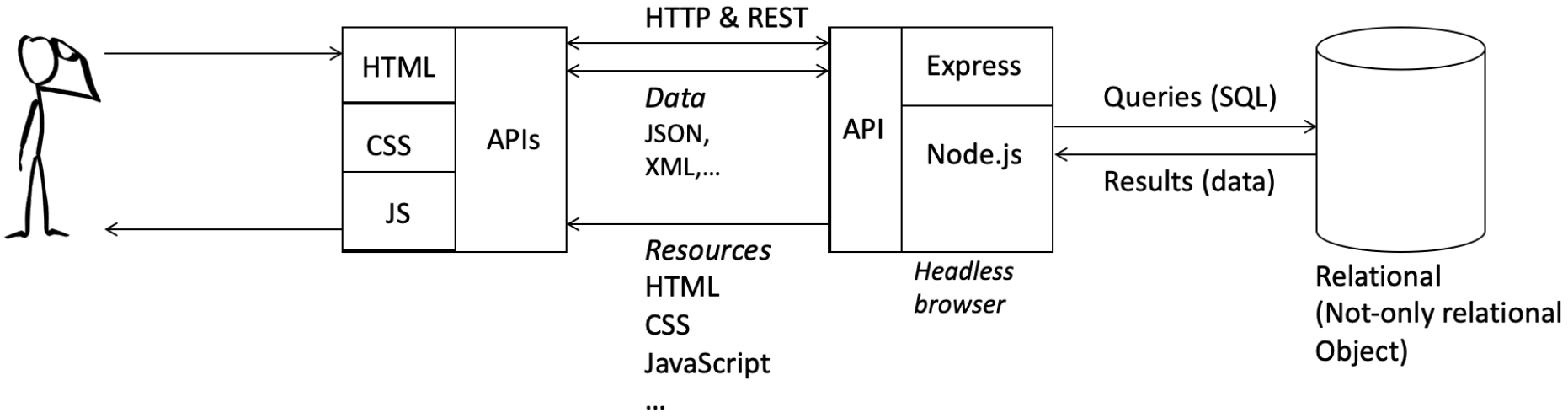
- Introduction to Web Computing
- HTTP
- JavaScript basic concepts
- Introduction to the assignments

In the lab

- Pre-labs x 3
- Coming up:
  - Introduction to Node.js
  - Introduction to persistence
  - Structuring your server application

In lecture this week

- Variable scoping and closures
- Asynchronous behaviours
- Event loop
- Callback hell
- Promises
- async/await syntax
- Module dependency
- API versioning



User	HTTP client	HTTP Server	Database
Human	Machine	Machine	Machine

Reference model

Assignment 1

- Your eng-git repo has been created
  - skeleton project
    - Clone from eng-git into your own development environment
    - Install node modules: npm install
    - Create a .env file in the root directory of your project
    - Add .env to .gitignore
    - Add your specific environment variables to .env
  - API specification (see next slide)
  - README.md

The assignment in essence

- (Assignment Briefing on Learn)
- Implement the API specification provided in the repo
- We will assess the implementation using a suite of automated tests.
  - Assessing API coverage: how much of the API was implemented?
  - Assessing API correctness: was an endpoint correctly implemented?
- The automated tests are available for you
  - See the information in the README.md
  - You can see how well you are progressing
- For the actual assessment we will use different data, but intend to use the same (or similar) automated test suite.

JavaScript cont.

Scoping

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

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

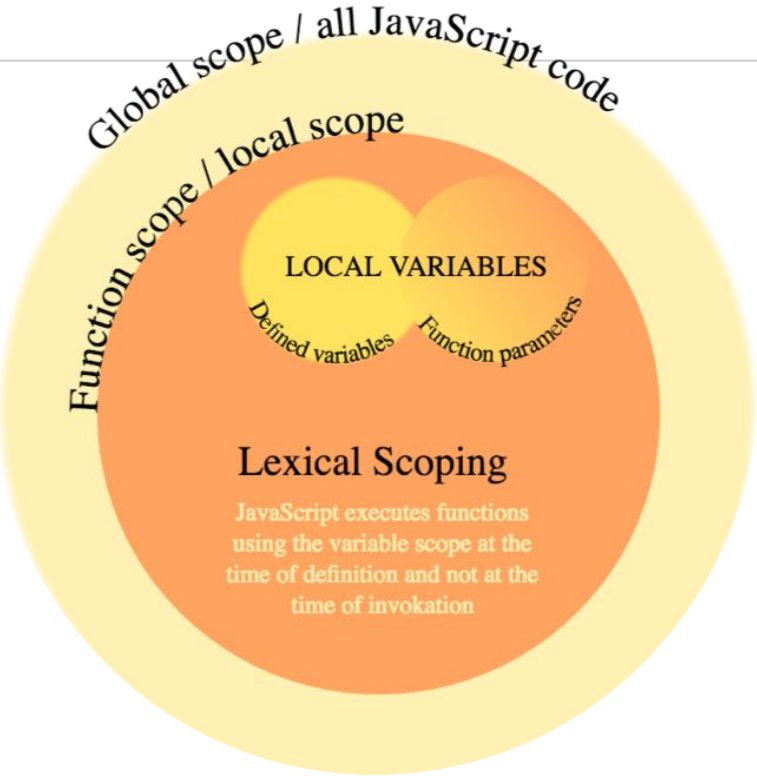
`x` is available only in the `if () {}` block

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)

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`

Arrow functions (ES6)

New anonymous function notation

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

Becomes

```
(a, b) => a + b;
```

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

Arrow functions and `this`

Unlike anonymous functions, arrow functions **do not bind their own `this`**

```
...
this.color = 'red';
setTimeout(() => {this.color = 'green'}, 1000);
```

Method chaining (cascading)

```
let result = method1().method2(args).method3();
```

- Each method in the chain returns an object…
  - each method must execute a `return…;` statement
- The returned object must 'contain' within it the next method being invoked in the chain.
  - `method1()` returns an object that has `method2()` within it, so that you can call `method2()`
- The first method in the chain may need to create the object.
- Usually, each method contains a `return this;` as a pointer to a common object being worked with
  - That common object contains all of the methods e.g. `method1`, `method2` and `method3`
- You can't just arbitrarily chain together any ol' set of methods

Example

```
let anotherperson = { firstname: 'Ben',
                      surname: 'Adams',
                      printfullname: function () {
                          console.log(this.firstname + ' ' + this.surname);
                          return this;
                      },
                      printfirstname: function () {
                          console.log(this.firstname);
                          return this;
                      },
                      printsurname: function () {
                          console.log(this.surname);
                          return this;
                      }
}
```

Why chain?

- Reduces temporary variables
  - No need to create temporary variable(s) to save each step of the process.
- The code is expressive
  - Each line of code expresses clearly and concisely what it is doing
  - (Using verbs as names for methods helps).
- The code is more maintainable
  - Because it's easier to read e.g. it can read like a sentence.
  - Because it requires a coherent design to the chained methods
- Method chaining used in, for example, Promises and other 'then-able' functions

`'use strict'`

Strict mode:

- (a way of managing backward compatibility)
- **modifies semantics** of your code
  (modifies the interpretation of your code), e.g.:
  - `this` is defaulted to undefined
  - less lenient about variable declarations e.g. `var`
  - throws errors rather than tolerating some code
  - rejects `with` statements, octel notation
  - Prevents keywords such as `eval` being assigned
- like a linter (e.g. linters give warnings and strict mode throws errors)
- Linters need to be configured to 'play nicely' with strict mode
  - can be applied to entire script or at function level

Asynchronous JavaScript

The Event Loop (JavaScript Concurrency model)

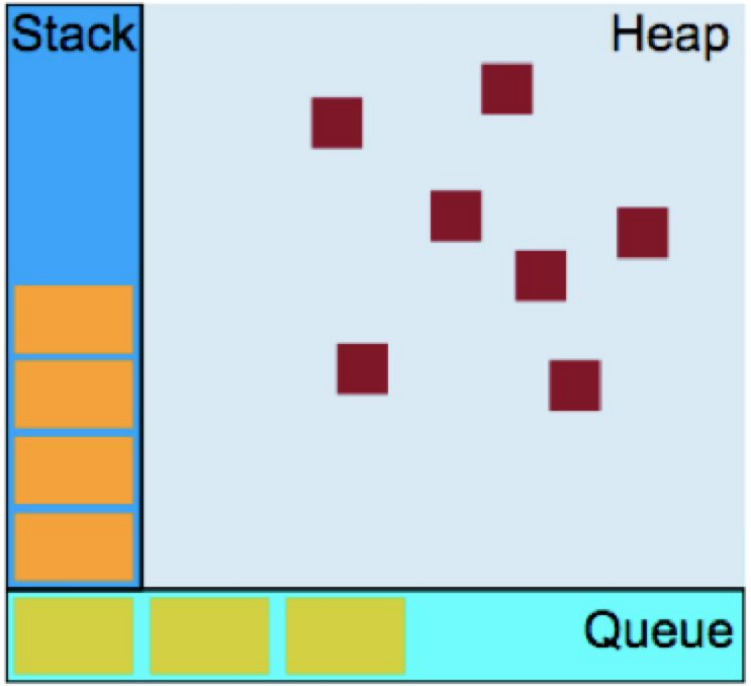
Call Stack: a data structure to maintain record of function calls.

- Call a function to execute: push something on to the stack
- Return from a function: pop off the top of the stack.

(The single thread.)

Heap: Memory allocation to variables and objects.

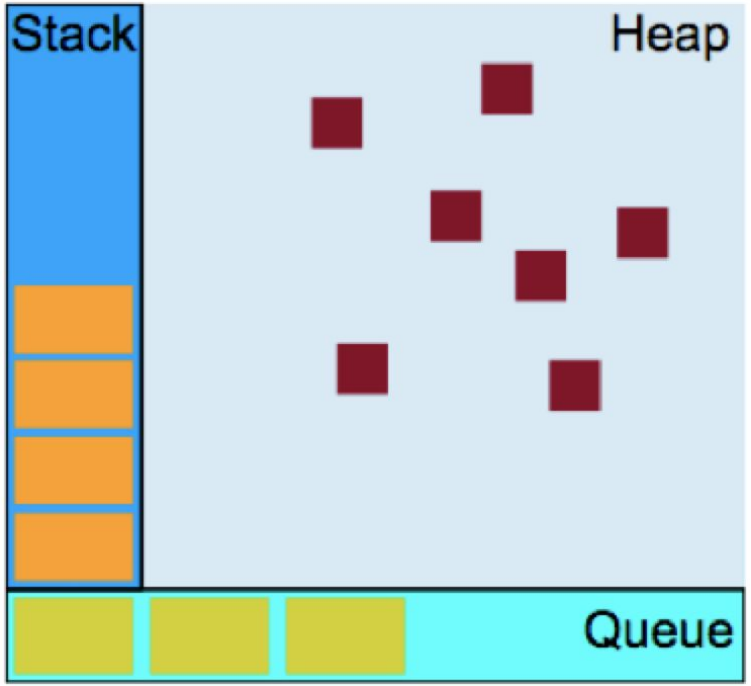
Queue:a list of messages to be processed and the associated callback functions to execute.

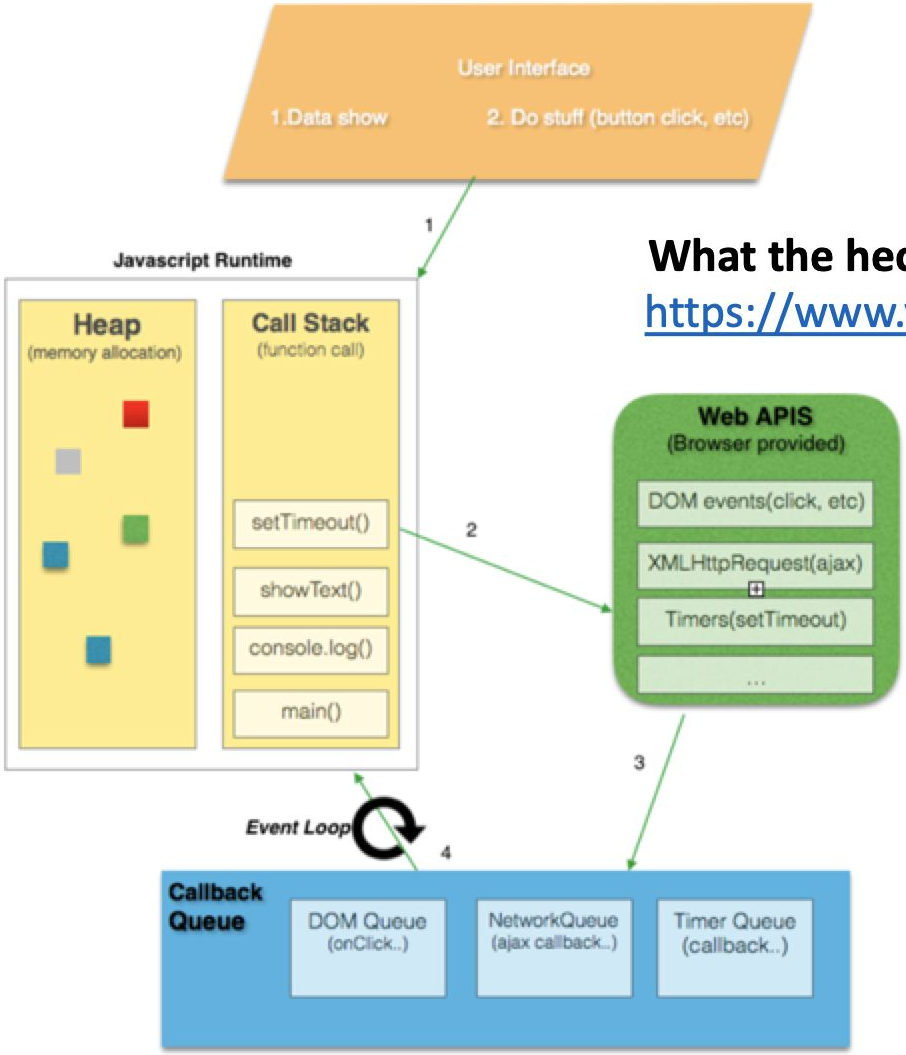


An initial example

What will complete first? … and why?

```
Line    Code
1       setTimeout(() =>
console.log('first'), 0);
2       console.log('second')
```





What the heck is the event loop anyway?
https://www.youtube.com/watch?v=8aGhZQkoFbQ

`setTime()` and `setInterval()` code examples

```
/* jshint esversion: 6 */

let currentDateTime;
let currentTime;

console.log('Script start: ' + getTheTime());

function ping() {
  console.log('Ping: ' + getTheTime());
}

console.log('ping function declared: ' + getTheTime());

function sayHi(phrase, who) {
  console.log( phrase + ', ' + who + ', it\'s ' +
  getTheTime());
}

console.log('sayHi function declared: ' + getTheTime());

setInterval(ping, 500); // Initiate ping events
setTimeout(sayHi, 1000, "Hello", "Austen"); // Initiate
message

console.log('setInterval and setTimeout executed: ' +
getTheTime());

function getTheTime () {
  currentDateTime = new Date();
  currentTime = currentDateTime.toLocaleTimeString();
  return currentTime;
}

console.log('Script end: ' + getTheTime());

```

Callback Hell and the Pyramid of Doom

by Asychronous JavaScript

```
//TODO: refactor, to avoid the pyramid of doom, by using promises

db.getPool().query('DROP TABLE IF EXISTS bid', function (err, rows){
    if (err) return done({"ERROR":"Cannot drop table bid"});
    console.log("Dropped bid table.");

    db.getPool().query('DROP TABLE IF EXISTS photo', function (err, rows){
        if (err) return done({"ERROR":"Cannot drop table photo"});
        console.log("Dropped photo table.");

        db.getPool().query('DROP TABLE IF EXISTS auction', function (err, rows){
            if (err) return done({"ERROR":"Cannot drop table auction"});
            console.log("Dropped auction table.");

            db.getPool().query('DROP TABLE IF EXISTS category', function (err, rows){
                if (err) return done({"ERROR":"Cannot drop table category"});
                console.log("Dropped category table.");

                db.getPool().query('DROP TABLE IF EXISTS auction_user', function (err, rows){
                    if (err) return done({"ERROR":"Cannot drop table auction_user"});
                    console.log("Dropped auction_user table.");
                    done(rows);
                });

            });
        });

    });
});
```

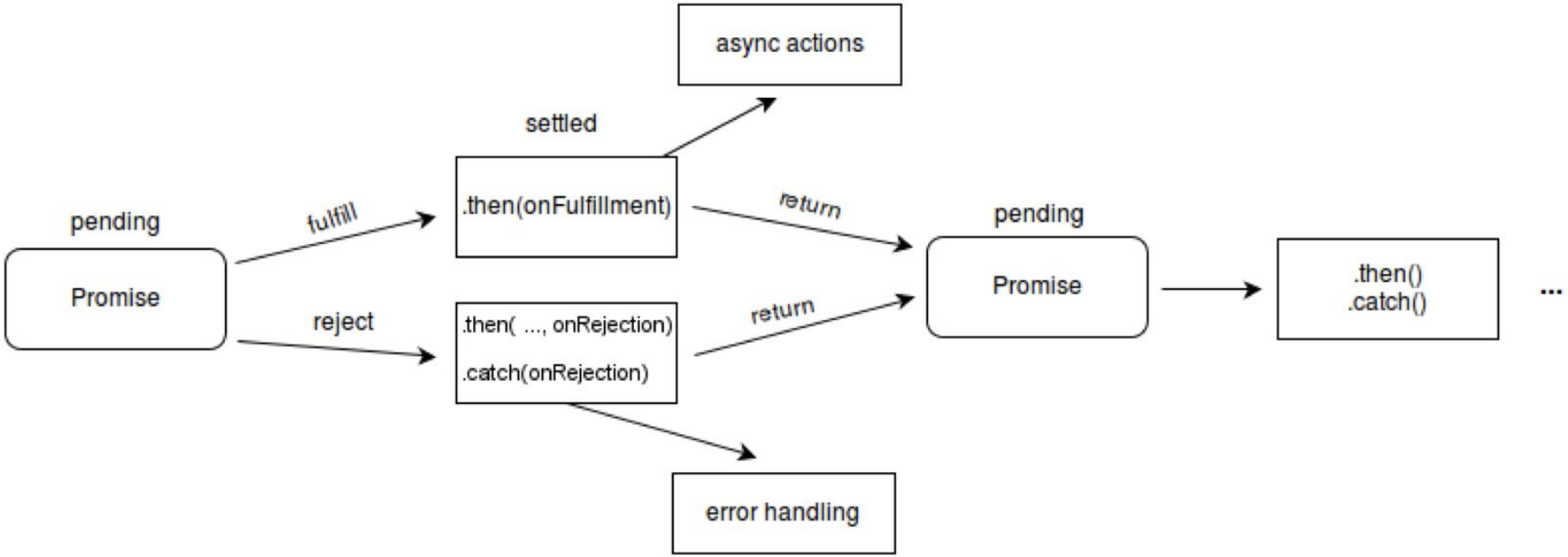
APIs and callback hell

Consider that:

- An API call from the client may under-fetch data...
  (the API is not designed to provide all and only the data the client needs)
- ... so the client will need to make subsequent API calls.
- For example:
  - First, an API call to get a list of student IDs in order to select one ID, then
  - an API call to get the list of courses studied by that student, and then
  - another API call to get further information of specific courses
- This produces a nested (conditional) set of API calls
  - For each call, the client must test whether the call was successful or not

Promises

- The `Promise` object is used for deferred and asynchronous computations.
- Promises allow you to **use synchronous and asynchronous** operations with each other
- A Promise represents an **operation that hasn't completed yet**, but is expected in the future.
  - **pending**: initial state, not fulfilled or rejected.
- Or resolved as either:
  - **fulfilled**: meaning that the operation completed successfully.
  - **rejected**: meaning that the operation failed.


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
.then()
.catch()
...
Promise
reject
.then(..., onRejection)
.catch(onRejection)
return
error handling

Chaining **Promises**

- Each Promise is first **pending**, and then (eventually) either **fulfilled** OR **rejected**
- Chaining Promises allows you to chain dependent asynchronous operations, where each asynchronous operation is itself a Promise
- Each `Promise` represents the completion of another asynchronous step in the chain.
- To chain promises, each `Promise` returns another `Promise`
  - Technically, each `then()` returns a `Promise`
- Chain promises together using `.then()`
- Can have multiple `.then()`s
- Handles rejected state/s with `.catch()` (or `.then(null, callback)`)

ASYNC / AWAIT

async ensures a function returns a `Promise`

(a kind of function wrapper)

```
async function f() {
    return 1;
}
f().then( () => { console.log(result); } );
```

Note:

`async` doesn't execute the function immediately

Babel + JSX + No-Library (pure JS) ▼

```
1  console.log("Start");
2
3  function g(input) {
4    return input + 1;
5  }
6
7  let a_function = g;
8  console.log(a_function);
9  console.log("The result of g(2) is :", g(2));
10
11 async function f(input) {
12   return input + 1;
13 }
14
15 let another_function = f;
16 console.log(another_function);
17 console.log("The result of f(2) is: ", f(2));
18
19 f(10).then(function(result) {
20   console.log("The result of f() is: " + result);
21 });
22
23 another_function(100).then(function(result) {
24   console.log("The result of another_function (f()) is: " + result);
25 });
26
```

Start

function g(input) {
return input + 1;
}

The result of g(2) is : 3

function f(_x) {
return _ref.apply(this, arguments);
}

The result of f(2) is: [object Promise]

The result of f() is: 11

The result of another_function (f()) is: 101

https://jsfiddle.net/dnxquzym/15/

`await` forces JS wait for the `Promise` to resolve

- `await` is only legal inside an `async function`...

- ... and async functions are Promises that commit to a future resolution...
- ... so other code can continue to run

Module dependencies.

module.export / require()

Modular JavaScript files

- **CommonJS**: one specification for managing module dependencies
  - Others exist e.g. RequireJS, ES2015 AMD (Async Module Definition)
- Node.js adopted CommonJS
  - To use CommonJS on front-end, you'll need to use Browserify (or similar)
- A module is defined by a **single JavaScript file**
- Use `module.exports.*` or `exports.*` (but not both) to expose your module's public interface
- Values assigned to `module.exports` are the module's public interface
  - A value can be lots of things e.g. string, object, function, array
- You want to expose something? Add it to `module.exports`
- Import the module using `require()`

Creating modules & reusing existing module

You can create your own modules

```
myModule.js
  module.exports...
```

And then reuse that module :

```
myOtherModule.js
  var something = require('../../myModule.js');
```

Dependency management

- npm is a package manager for node
- npm is designed to be node-specific
- `npm install` installs packages suitable for the CommonJS-like dependency management used by Node, i.e. the exports/requires approach

Creating modules & reusing existing module (npm)

You can reuse existing modules provided by the node ecosystem

First, install the existing module through npm

```
> npm install aModule
```

And then reuse that module :

```
myOtherModule.js
  var something = require(aModule);
```

Note the differences in parameters for node and home-grown modules