CS2003 W07
JSON & Async

Özgür Akgün

Slides adapted from Al Dearle and Saleem Bhatti

read: https://www.w3schools.com/js/js_json_intro.asp

Java Script Object Notation

- JSON is a syntax for storing and exchanging data.
- JSON is text, written with JavaScript object notation.
- Can easily convert JSON Strings into objects
- Can easily convert objects in JSON Strings

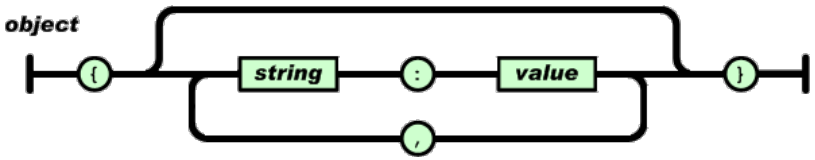
```
JSON.stringify( obj )
JSON.parse( text )
```

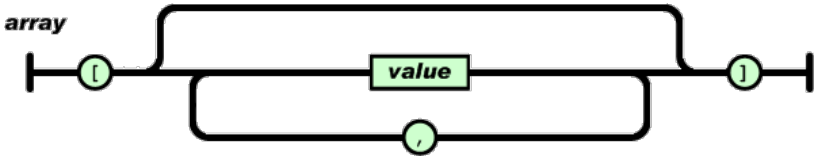
- **Can be sent between browser and a server**
- Can be stored in files - mime type “application/json"
- Language independent, untyped

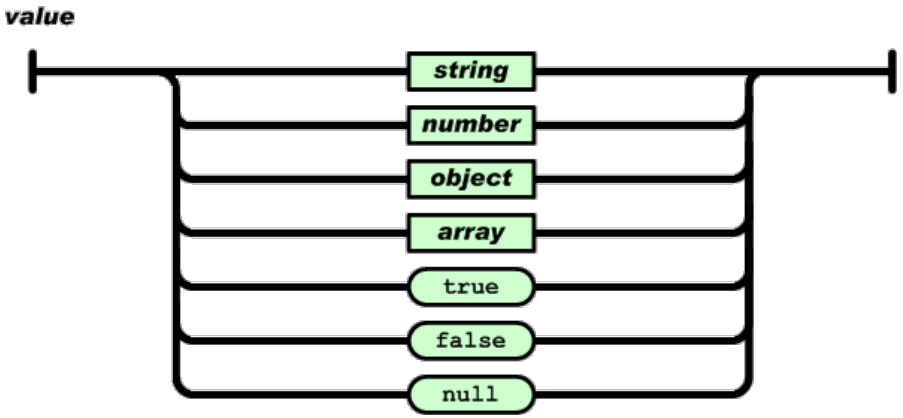
JSON Syntax

- JSON syntax is derived from JavaScript object notation syntax:
  - Data is in name/value pairs
  - Data is separated by commas
  - Curly braces hold objects
  - Square brackets hold arrays
- Values must be one of the following:
  - a string
  - a number
  - an object (JSON object)
  - an array
  - a boolean
  - null

JSON Syntax







from https://www.json.org

06json.js

```
function showPerson( person ) {
   console.log( "person: " )
   console.log( "   " + person.name )
   console.log( "   " + person.age )
   console.log( "   " + person.city)
}

function showJSON( json ) {
   console.log( "JSON: " )
   console.log( "   -->" + json + "<--" )
}

const john = {name: "John", age: 18, city: "St Andrews"};
console.log( john )

var json = JSON.stringify(john);
showPerson( john )
showJSON( json )

json = '{"name":"Betty", "age":21, "city":"St Andrews"}';
var betty = JSON.parse(json);

showPerson( betty )
showJSON( json )
```

JSON and XML

- Both JSON and XML are self describing hierarchical data structures
- Both can be used in XMLHttpRequests
- Both programming language independent formats
- XML is more complex to parse
- JSON has better fit with javascript (not surprisingly)
- Schema in XML is more sophisticated (XMLSchema)

- more at: https://www.w3schools.com/js/js_json_intro.asp
- but not much more..


Web
sockets

WebSockets

- A WebSocket provides full-duplex communication over a TCP connection between a web browser (or other program) and a web server
- IETF as RFC 6455 in 2011
- This is made possible by providing a standardized way for the server to send content to the client without first being first requested by the client
- Thus a two-way conversation can take place between a client and the server.

Client WebSockets API

- Javascript API:
  - client-side and server side (for node.js use “ws” module)
- WebSocket object
- Events:
  - on open
  - on error
  - on close
  - on message
- https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
- API at: https://github.com/websockets/ws/blob/HEAD/doc/ws.md

Server WebSockets API - ws

- **ws** module used in this example:
  https://www.npmjs.org/package/ws
  more examples on this page

Sending and receiving:

```
const WebSocket = require('ws')
const ws = new WebSocket('ws://www.host.com/path')
ws.on('open', function open() {
  ws.send('something')
})
ws.on('message', function incoming(data) {
  console.log(data)
});
```

Examples – CS2003/Examples/web4/

- json-stringy_parse/:
  - encode and decode of JSON objects
- node-events/:
  - events and asynchronous timing
- node-simple_server_client/:
  - HTTP server, TCP echo server, TCP client for echo server
- node-chat_server/:
  - multicast heartbeat and chat, TCP chat
- node-message_board_simple-ws/:
  - WebSockets for client-server communication

Useful materials

- https://www.ecma-international.org/ecma-262/
- https://developers.google.com/web/fundamentals/primers/promises
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing


Async

Synchronous JavaScript

- Javascript does not have threads
- A single locus of control (Denning) always exists in a JavaScript program
- Two bits of Javascript cannot run at the same time; they have to run one after another.
- Typically JavaScript co-exists in an environment that is as rendering the user interface, updating styles, and handling user actions.
- Activity in one of these things delays the others.

Synchronous and asynchronous

- Let us look at an example again:

```
var http = require('http');
http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': ‘text/html’});
 res.end('Hello world\n');
}).listen(10111, '127.0.0.1');

console.log('Server running’);
```

- Why is the console message displayed before the server serves anything?

Look at a modern web interface

- skyscanner.net
- e.g.

- https://www.skyscanner.net/transport/flights/edi/ewr/190930/191004/?adults=1&children=0&adultsv2=1&childrenv2=&infants=0&cabinclass=economy&rtn=1&preferdirects=false&outboundaltsenabled=false&inboundaltsenabled=false&ref=home#/

Asynchronous code

- Many Web interfaces now use asynchronous code to run
- The skyscanner site is an example of this.
- When they fetch a resource (from the network for example) or run a complex query on a server the results are ‘painted’ into the DOM tree asynchronously - Network I/O is **slow**….
- We do not want to have to sequentially wait for each query to be evaluated
- How do you do this when you have a single thread of control?
  - do this, then this, then this, then this …..

Asynchrony in Javascipt

- We will look at the 2 styles of asynchronous code in Javascript:
  1. Callbacks
  2. Promises

Callbacks

- You have seen a few examples of asynchronous behaviour already e.g.
  - http.createServer() takes a function which is called when a message is received
  - event listeners call a function when an event happens

```
  http.createServer(
    function (req, res) {
   // some code here
}
  ).listen(port, address)
```

```
   btn.addEventListener('click', () => { alert('Clicked!');
```

Callbacks

- When we pass a callback function as a parameter to another function, we are only passing the function definition as the parameter
- The callback function is not executed immediately
- It is "called back" (hence the name) asynchronously somewhere inside the containing function's body.
- The containing function is responsible for executing the callback function at the appropriate time (usually in response to an event).
- A callback is therefore "any executable code that is passed as an argument to other code that is expected to call back (execute) the argument at a given time." [Wikipedia]
- Here is a function - call me…

Callback hell (from callbackhell.com)

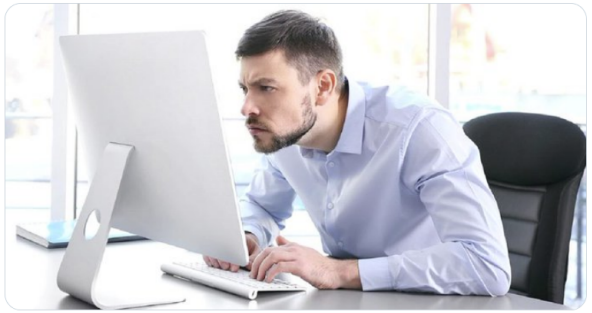
```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

What does this do? - quickly!
Try debugging this!

```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

amy
@ahsurelook

lecturers choosing which meme from 2006 should go into their slides



8:18 PM · Oct 13, 2020 · Twitter for iPhone

832 Retweets **273** Quote Tweets **12K** Likes

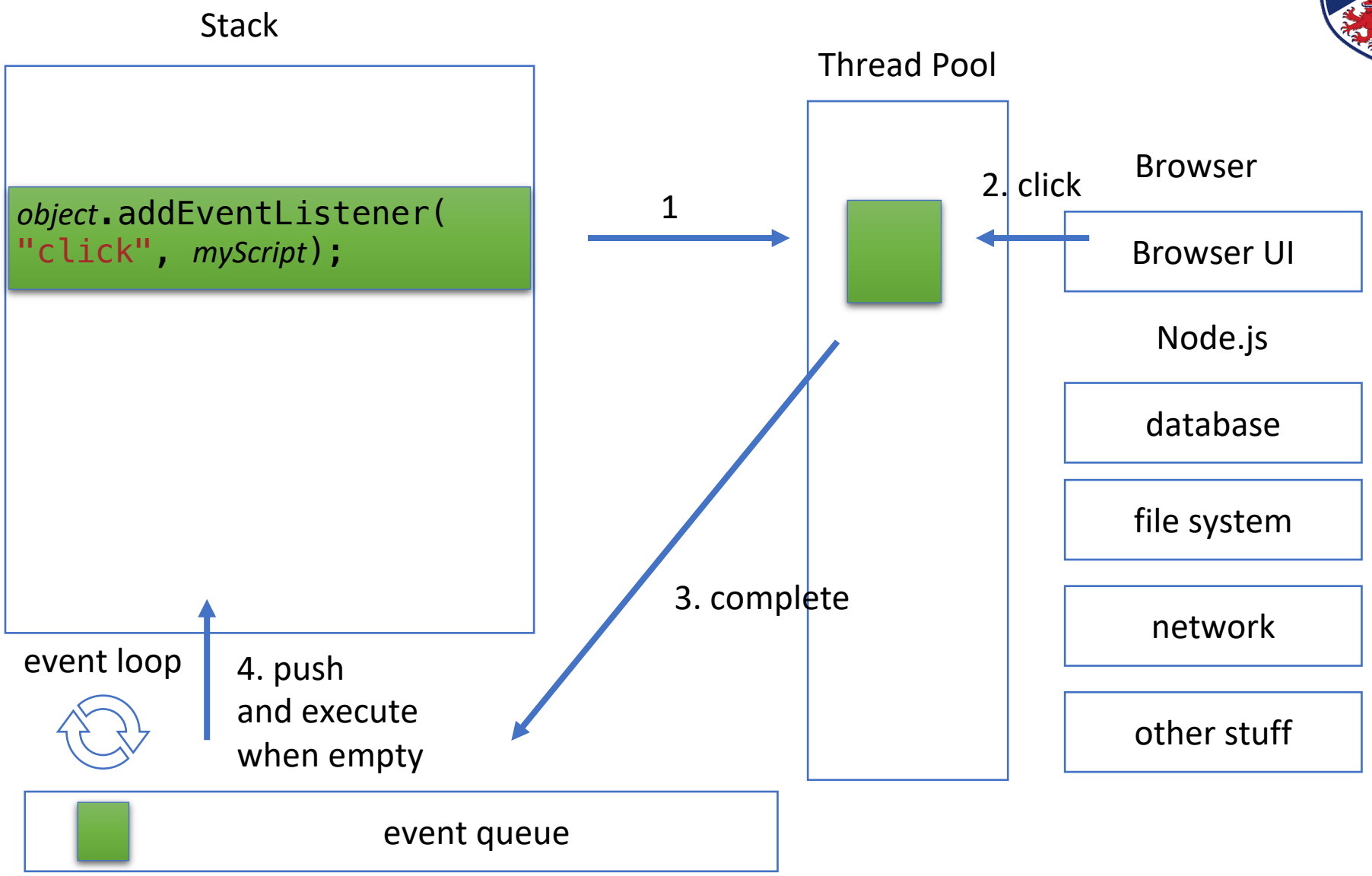
A simple asynchronous example

```
console.log( "See ya " )

setTimeout( function() {
                  console.log( "later")
        }
        , 5000
       )

console.log( "finish" )
```

- What gets printed/when/how?
- Example in 01-timeout.js

How do callbacks work?

- Although the language is single threaded, the runtime environment (browser or node) provides some more concurrency support
- When you call setTimeout() or one of the other functions that take callback functions that code is run asynchronously.
- When the completes (when the user clicks, a http message comes in, the timer finishes etc.) the completed task is put onto a **queue**.
- When the stack is empty, another thread in the run time (the event loop) takes the tasks off the queue and pushes the tasks onto the stack where they are executed.


Stack
object.addEventListener("click", myScript);
Thread Pool
1
2. click
Browser
Browser UI
Node.js
database
file system
network
other stuff
3. complete
event loop
4. push
and execute
when empty
event queue

This is nicely explained in this (good) video

- https://www.youtube.com/watch?v=8aGhZQkoFbQ
- Philip Roberts | JSConf EU
- &yet
- First 20 minutes are very relevant to us here

- Interesting project - Loupe
- https://github.com/latentflip/loupe
- Execution visualiser

So what is wrong with callbacks?

- Ideally you want something like this

```
img1.callThisIfLoadedOrWhenLoaded(function() {
  // loaded
}).orIfFailedCallThis(function() {
  // failed
});


// and…
whenAllTheseHaveLoaded([img1, img2]).callThis(function() {
  // all loaded
}).orIfSomeFailedCallThis(function() {
  // one or more failed
});
```

from https://developers.google.com/web/fundamentals/primers/promises#whats-all-the-fuss-about

Promises

- A Promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation.
- A Promise is an object representing the eventual completion or failure of an asynchronous operation. Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.
- See
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing#Promises
- https://www.ecma-international.org/ecma-262/#sec-promise-objects
- https://developers.google.com/web/fundamentals/primers/promises#whats-all-the-fuss-about

Promises

- A Promise has three fields:
  - The object that is promised
  - A function that is used to resolve/fulfil the promise
  - A function that is used to reject the promise
- Promises are in one of three mutually exclusive states: *fulfilled*, *rejected* and *pending*
- A promise p is **fulfilled** if p.then(f, r) will immediately enqueue a Job to call the function f.
- A promise p is **rejected** if p.then(f, r) will immediately enqueue a Job to call the function r.
- A promise is **pending** if it is neither fulfilled nor rejected.

Consuming Promises

- Promises are **thenable** (sorry for grammar)
- That is they have a then method
- The then method takes two functions - for success and failure - called *onFulfilled* and *onRejected* also known
- Syntax:

```
p.then(onFulfilled[, onRejected]);

p.then(value => {
  // fulfillment
}, reason => {
  // rejection
});
```

The two functions for success an failure

- Once a Promise is fulfilled or rejected, the respective handler function (*onFulfilled* or *onRejected*) will be called asynchronously (scheduled by the current event loop).

see https://www.ecma-international.org/ecma-262/6.0/#sec-promise.prototype.then
material from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

An example - Examples/node/cs2003/web4/02-promise.js and 02a

```
const https = require('https');

function doStuff( b ) {  … return val } // something that takes a while

function createPromise( some_value ) {

  return new Promise( (resolve, reject) =>
              {
                 let result = doStuff( some_value )
                  if( some_value == 3 ) {
                   resolve( result )
                 } else {
                   reject( result )
                 }
              }
            )
}


createPromise(3).then(function(response) {   // this example only accepts 3!!!
  console.log("Success - that worked with 3", response);
}, function(error) {
  console.error("Failed: didn't work with 3", error);
})


createPromise(6).then(function(response) {
  console.log("Success - that worked with 6", response);
  // could write out the response here
  // console.log( response );
}, function(error) {
  console.error("Failed: didn't work with 6", error);
})
```

I have told you everything!

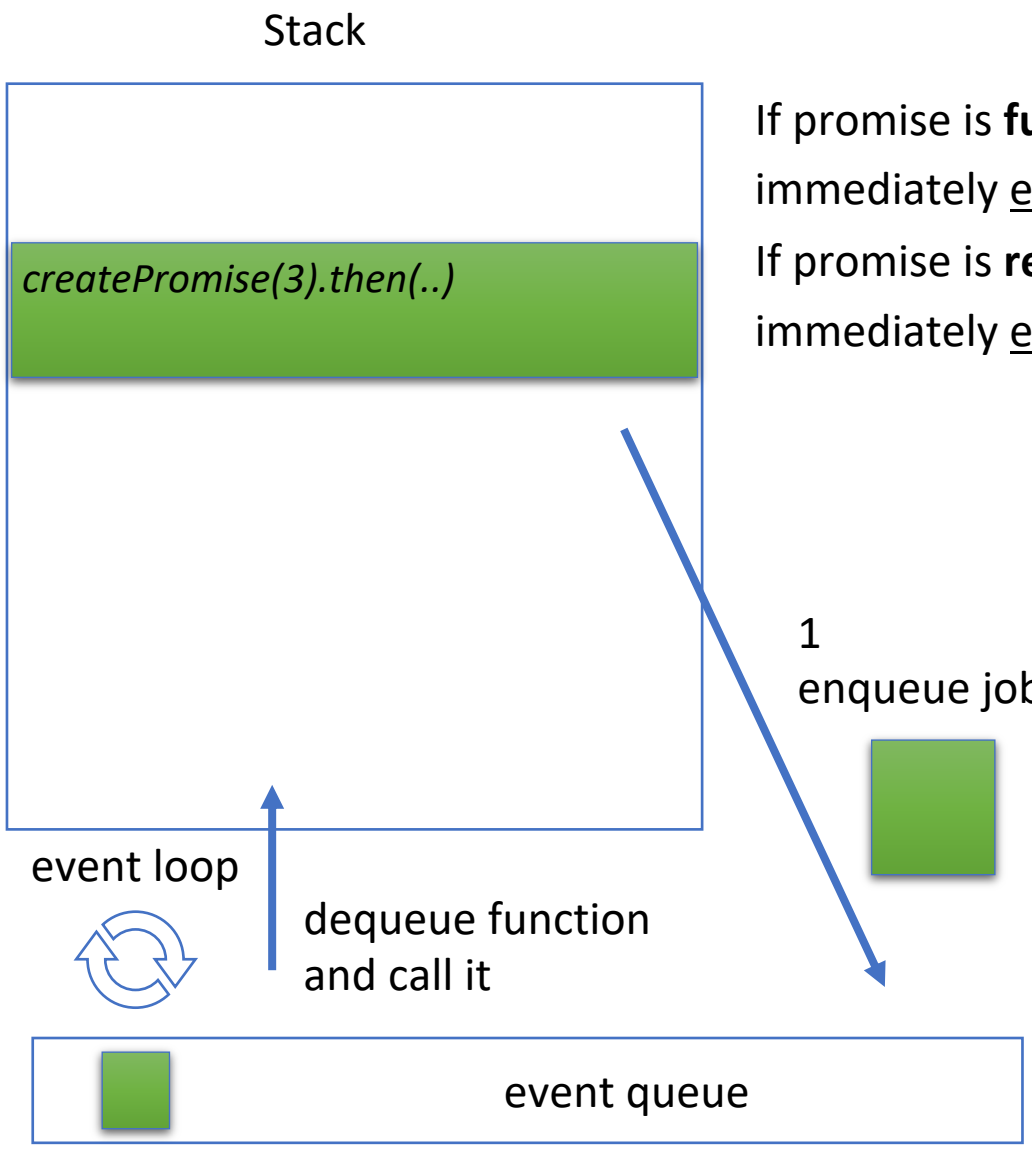
- A promise p is **fulfilled** if p.then(f, r) will immediately enqueue a Job to call the function f.
- A promise p is **rejected** if p.then(f, r) will immediately enqueue a Job to call the function r.
- Once a Promise is fulfilled or rejected, the respective handler function (*onFulfilled* or *onRejected*) will be called asynchronously (scheduled by the current event loop).

- Example 02b-promise.js is an annotated version of the code

```
function createPromise( some_value ) {
  return new Promise( (resolve, reject) =>
              {
                 console.log( "Running the
                                  Promise handler " )
                 let result = doStuff( some_value )
                  if( some_value == 3 ) {
                   console.log( "Resolving the value " )
                   resolve( result )
                 } else {
                   console.log( "Resolving the value " )
                   reject( result )
                 }
              }
            )
}


createPromise(3).then(function(response) {
  console.log("Success - that worked with 3”,
               response);
}, function(error) {
  console.error("Failed: didn't work with 3", error);
}
createPromise(6).then(function(response) {
  console.log("Success - that worked with 6”
               response);
}, function(error) {
  console.error("Failed: didn't work with 6", error);
})

console.log( "done" )
```

```
Running the Promise handler
Running doStuff() 3
Running doStuff() 3
Running doStuff() 3
Running doStuff() 3
Running doStuff() 3
Resolving the value
Running the Promise handler
Running doStuff() 6
Running doStuff() 6
Running doStuff() 6
Running doStuff() 6
Running doStuff() 6
Resolving the value
done
Success - that worked with 3 0
Failed: didn't work with 6 0
```

DEEP DIVE - DON"T WORRY IF YOU DON"T FOLLOW THIS IS MEANT TO HELP!!!



If promise is **fulfilled** if p.then(f, r) will immediately enqueue a Job to call the function f.

If promise is **rejected** if p.then(f, r) will immediately enqueue a Job to call the function r.

2. When stack is empty call the event loop

Catch

- In addition to then() there is also catch()
- The catch() block at the end runs if any of the .then() blocks fail
- This is similar way to synchronous try...catch blocks
- An error object is made available inside the catch() which can be used to report the kind of error that has occurred.

- See: https://www.ecma-international.org/ecma-262/6.0/#sec-promise.prototype.catch

Multiple Promises

- Sometimes you want to download many items or wait for many things to be performed
- There is an API for this -

```
  Promise.all(arrayOfPromises).
then( function(arrayOfResults) {…} )
```

- The all function takes an iterable of Promises and returns a Promise that fulfils only when all of the Promises in the array have completed (or the first reject)
- The result is an array of results - the results of evaluating the promises in the array
- There are other methods in this API - allSettled() (slightly different semantics), race() (any of them resolved) ….

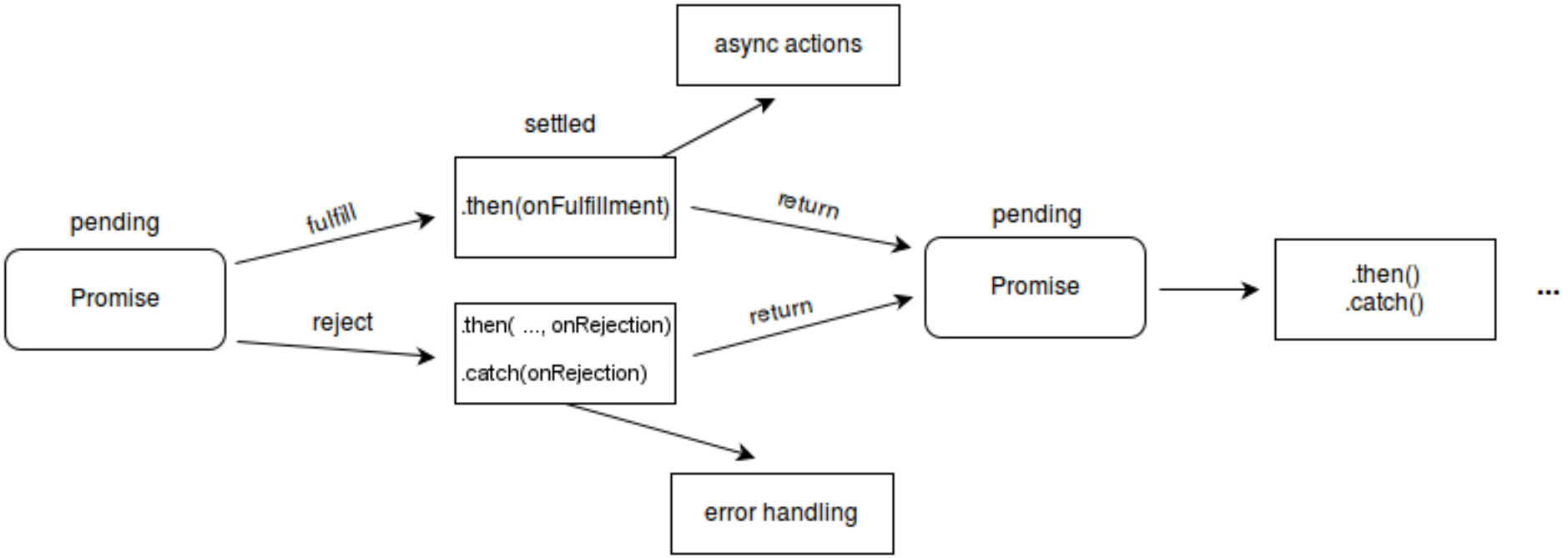
Real life examples

- can be found in:
  - Examples/node/cs2003/web4/03-promise.js
  - Examples/node/cs2003/web4/04-promise.js
- these are a thing of beauty!
- Lots of worked examples here:
  - https://developers.google.com/web/fundamentals/primers/promises#whats-all-the-fuss-about
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

Handler functions

- The behaviour of the handler function follows a specific set of rules. If a handler function
  - returns a value, the promise returned by then gets **resolved** with the returned value as its value;
  - doesn't return anything, the promise returned by then gets **resolved** with an undefined value;
  - throws an error, the promise returned by then gets **rejected** with the thrown error as its value;
  - returns an already fulfilled promise, the promise returned by then gets fulfilled with that promise's value as its value;
  - returns an already rejected promise, the promise returned by then gets rejected with that promise's value as its value;
  - returns another pending promise object, the resolution/rejection of the promise returned by then will be **subsequent** to the resolution/rejection of the promise returned by the handler. Also, the value of the promise returned by then will be the same as the value of the promise returned by the handler.
- All this means that **thens (and catches)** can be **chained together** avoiding call-back hell.

Promise chaining



https://mdn.mozillademos.org/files/15911/promises.png

Asynchronous functions

- The **async** function declaration defines an asynchronous function, which returns an AsyncFunction object
- An asynchronous function is a function which executes asynchronously via the event loop, using an implicit Promise to return its result.
- https://www.ecma-international.org/ecma-262/#sec-async-function-objects
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Aysnchronous examples

- 3 examples in: web4/ 05-async.js 05a-async.js 05b-async.js

```
function resolveAfter1Second() {
 console.log('resolveAfter1Second called');
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('timed out');
      resolve('resolved');
    }, 1000);
  });
}

async function asyncCall() {
  console.log('asyncCall called');
  var result = await resolveAfter1Second();
  console.log(result);
  // expected output: 'resolved'
}

console.log('before call');
asyncCall();
console.log('after call');
```

```
before call
asyncCall called
resolveAfter1Second called
after call
timed out
resolved
```


Cookies
& State

Client-side storage

- For some applications, it is useful to store information at the client side, e.g.:
  - user preferences for a site
  - configuration information
  - returning visitor to a site
  - information to re-use later in the the web application
- Some knowledge for a session:
  - a single use of a document (more on sessions later)
  - HTTP is stateless, but cookies and localStorage (later) add state

Cookies

- Cookies provide a small amount of storage for a client.
- Privacy concerns (more later)
- DOM document.cookie property:
  - manages cookies using Javascript
  - string value, read and write
  - not-secure (plain text)

Cookie – values

- A cookie is a string that looks like this:

  name1=value1;name2=value2;name3=value3;

- Storage is managed locally by the client (browser):
  - no access to a file-level object at the client level.
- Reading document.cookie returns the whole string, i.e. all (name, value) pairs in the cookie.
- https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie

Predefined cookie fields

- path: specifies the web page(s) with which the cookie is associated
- domain: domain associated with the current cookie, default is that for the document, and what pages can see the cookie
- max-age: in seconds
- expires: date of expiry
- secure: if set, can only be used with HTTPS
- max-age:
  - cookie lifetime in seconds
  - cookies are transient by default
  - if max-age is set to be non-zero, cookie is saved in local file system by browser
- expires:
  - GMT string for expiry of cookie

HTML5 – LocalStorage

- HTML5 adds better support for local state: LocalStorage
- SessionStorage:
  - only for that usage of the page, removed when window is deleted
- Cleaner API for manipulating name,value pairs:
  - via a localStorage object
- Example: CS2003/Examples/web/web4/localstorage/localstorage-1.html CS2003/Examples/web/web4/localstorage/localstorage-snooper.html

LocalStorage API

- clear():
  clear cookie for that page
- getItem(key):
  key (name) for which a value has been set
- setItem(key, value):
  set key,value pair
- removeItem(key):
  remove the key,value pair for key
- https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage

Session

- A session can be:
  - the single use of a page
  - an application level session that extends beyond the lifetime of a single transport level connection
  - a long-lived application level interaction for a user
- For SessionStorage, it is for the duration that the page is available in a browser window or tab.
- There is a sessionStorage API (similar API to LocalStorage): https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage

Privacy

- Clearly, cookies and local storage can be used to track users:
  - information between sites.
  - sites visited.
  - information input from a session.
- The cookies are all plain text:
  - just need to know the name-value pairs to read.
- Cookies can be set by server in HTTP response:
  - Set-Cookie: name=newvalue
  - any server of content can set a cookie in a HTTP response
  - pages with content from third-parties (e.g. adverts)

Privacy

- ePrivacy Directive EU 2002/58/EC:
  - directive on Privacy and Electronic Communications
  - https://ec.europa.eu/digital-single-market/en/news/eprivacy-directive
- General Data Protection Regulation, EU 2016/679:
  - https://eur-lex.europa.eu/eli/reg/2016/679/oj
  - https://eugdpr.org
- Opt-in policy for use of tracking cookies:
  - users need to accept the use of cookies
- Web site providers must give details of all cookies used:
  - nature of “opt-in”?
  - how effective is this in reality?