JAVA SCRIPT

Dynamic Web Pages Client Side

- Java Script is a programming language that allows us to attach code to our web pages. The code can change the content/behavior of the page as the code runs.
- Java Script is
  - *Light weight: Limited overhead, types. More similar to Python than Java in many ways.*
  - *Untyped: Variables do not have type, but must be declared with var/let.*
    - Using `var` is not recommended; for new code it is recommended to use `let`
  - *Encapsulation/Inheritance grafted on: Supported, but not in common use.*
  - *Functional: Functions are first class entities*
  - *Interpreted: The goal is that you don't have to compile code but that each platform interprets the code. In practice, speed is an issue and you may compile to byte code or do a just in time compilation of small blocks of code into native machine code.*
  - *Prototype based: Create an instance of an object and use that to create other instances.*
  - *Promises: Asynchronous requests do not have to wait for a response. Can do multiples in parallel.*
- Note: unlike some programming languages (notably Java) the semicolons (;) that terminate lines are optional unless you are putting multiple statements on a single line

Inline

- onclick property has java script that is triggered
- Can reach into elements and replace contents.

```
<!DOCTYPE html>
<html>
<body>
  <h2>Time & Data</h2>
  <button type="button"
 onclick="document.getElementById('demo').innerHTML = Date();
                  alert('date fetched')">
    Click me to display Date and Time.
  </button>
  <p id="demo">This text will be replaced </p>

</body>
</html>
```

Define a function

- Define the function and call as needed.

```
<h2>Time & Dat3</h2>
<button type="button" onclick="dateClick()">
  Click me to display Date and Time.</button>
<p id="demo">This text will be replaced </p>
<script>
  function dateClick() {
    document.getElementById('demo').innerHTML = Date();
    alert('date fetched')
  }
</script>
</body>

</html>
```

Property/CSS

- We can change properties/styles.

```
<p id="demo">We have an image
  <img id="dog-image" width="100" height="100" alt="Picture of a dog"
src="cute.jpg">
</p>
<script>
  function changeSrc() {
    document.getElementById('demo').style = "font-family:Arial"
    document.getElementById('dog-image').src = "ghosted.jpg"
  }
</script>
```

External

- We can keep our code in a js file and use src property of script in the HTML to pull it in.
- The .js file has code but is not HTML, so no tags.

In the HTML:

```
<script src="somecode.js"> </script>
```

This is somecode.js:

```
function changeSrc() {
  document.getElementById('demo').style ="font-family:Arial"
  document.getElementById('dog-image').src ="ghosted.jpg"
}
```

Variables

- Case sensitive
- Untyped – assign anything you want to them
- Data types are Boolean, Number, String, Array and Object.
- Declared with (preferred)
  - `var` – *original just a var – non-lexical scoping.*
  - `let` – *limited scope variable*
  - `const` – *a constant.*
- Vars are local to an entire function or Global (outside a function or missing declaration).
- Local variables shadow global
- Start with $ often indicates methods in a framework. Avoid using.
- Start with _ often indicates "private" variables.

Block Scope and Let

```
var x
function f(k){
    var x = 10
    var x
}
```

Global x is defined outside the function. In this case, the local x declaration inside the function shadows the global and has value 10. Redeclaring a variable has no affect on the value, which will still be 10.

Variables declared inside a block {…} using var are visible outside the block. Let and const have standard lexical (block) scope.

```
{
    var a = 10
    let b =20
}
```

a can be used here outside the block, but not b.

Const

- Const means references can not be changed
- If we have an array or object, we can change the values inside.
- Must declare and set the value in one statement

Assignment and Operators

- Pretty standard.
  - *** is used for exponentiation*
- == checks for equality of value, === checks for type as well.

General statements (C/C++/Java style)

- `if (condition) {code} else if (condition) {code} else {code}`
- `for( init; test; update){ code }`
- `while(condition) { code }`
- `switch(value) {`
  `case literal: code; break;`
  `default:`
- `try{ code } catch(err) { code }`

Output Options

- Change the contents of an HTML element.
- Use `alert(stuff to display)` – we get a pop-up
- Use `console.log(stuff to display)` – The contents are recorded into a log.
- Console is where error messages show up as well.

Confirm/Prompt/Alert

- These all trigger a popup and should be used sparingly. Modal interactions force sequence.
- Examples:

```
alert("A message")
result = confirm("A message; returns true/false")
result = prompt("A message; returns a null/string",
    "default string")
```

Input/Label – HTML forms

- A form binds together a group of input fields where the user can enter values. Values in a form are often sent to a server/service.
- Example:

```
<form>
<label for="form1_age">Age: </label>
<input type="number" id="form1_age" name="form1_age"> <br>
<label for="form1_color">Color: </label>
<input type="text" id="form1_color" name="form1_color">
</form>
```

Select/Button– HTML forms

- You can add a select element that does a dropdown with options
- Example:

```
<form>
<label for="form2_size" Size: </label>
<select type="text" id="form2_size" name="form2_size">
 <option value="sm">Small</option>
 <option value="md">Medium</option>
 <option value="lg">Large</option>
</select>
<button type="button" onclick="form2Handler()">Go</button>
</form>
```

Input – HTML form

- An input element can have a number of different types.
  - *button* has properties: onclick, value
  - *text*
  - *number* has properties: min, max, step, value
  - *password*
  - *url*
  - *date*
  - *radio*
  - *checkbox*
- It should have a name that is used when it is submitted to server side. Value is used to get the input and as the default.
- Useful properties – required, disabled, autofocus

Radio Button Group

- Example:

```
<form>
  <input type="radio" id="rg1_red" name="color" value="red"
    checked >
  <label for="rg1_red">Red</label></br>

  <input type="radio" id="rg1_blue" name="color"
     value="blue">
  <label for="rg1_blue">Blue</label></br>

  <input type="radio" id="rg1_green" name="color"
     value="green">
  <label for="rg1_green">Green</label></br>
</form>
```

Checkbox Group

- Example:

```
<form>
  <input type="checkbox" id="cb1_cheese" name="topping1"
    value="cheese" checked>
  <label for="cb1_cheese">I want cheese</label></br>

  <input type="checkbox" id="cb1_olive" name="topping2"
     value="olive">
  <label for="cb1_olive">I want olives</label></br>

  <input type="checkbox" id="cb1_onion" name="topping3"
     value="onion" checked>
  <label for="cb1_onion">I want onions</label></br>
</form>
```

Objects

- Comma separated property:Value pairs in {}
- Example:
- `{id:1, name:"Fred", wages:2.45, isHappy:true}`

- This not a class, but a dictionary. A class would have named variables that mean something in the context of an instance of the class and are fixed for that class. What we have is a mapping of properties to values. We can add more mappings if we want.
- Keys may have spaces, but must be wrapped in quotes

Object Access

- `let my_object =  {id:1, name:"Fred", wages:2.45, isHappy:true}`

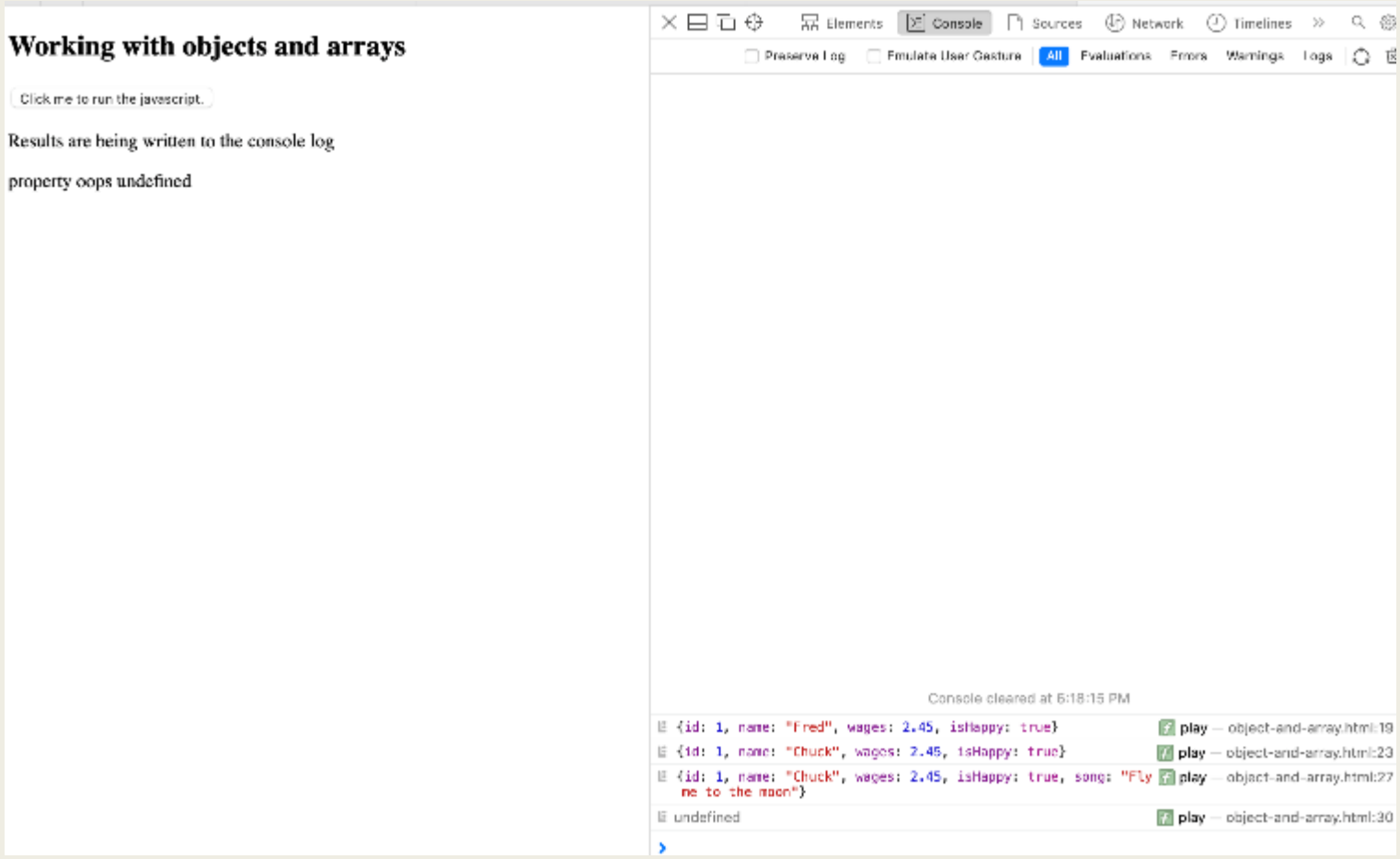
- Two ways we can access the values
  - *`let name = my_object.name`*
  - *`let name = my_object["name"]`*
- We can update similarly
  - *`my_object.name = "Chuck"`*
  - *`my_object["name"] = "Chuck"`*
- We can extend with a new property
  - *`my_object["song"] = "Fly Me To The Moon"`*
- If your key has spaces, you must use the bracket notation to access or update the item

Object Access

- `let my_object =  {id:1, name:"Fred", wages:2.45, isHappy:true}`

- If we try to access a property that is not defined we get undefined
  - *`let name = my_object.oops`*

Object Access



DOM

- Document Object Model – A tree that shows parent child relationships for all the elements in the document.
- Regular way to traverse the tree and expose the elements for access/modification. Effectively replaces jQuery. Warning: $ is the name of a function in jQuery.

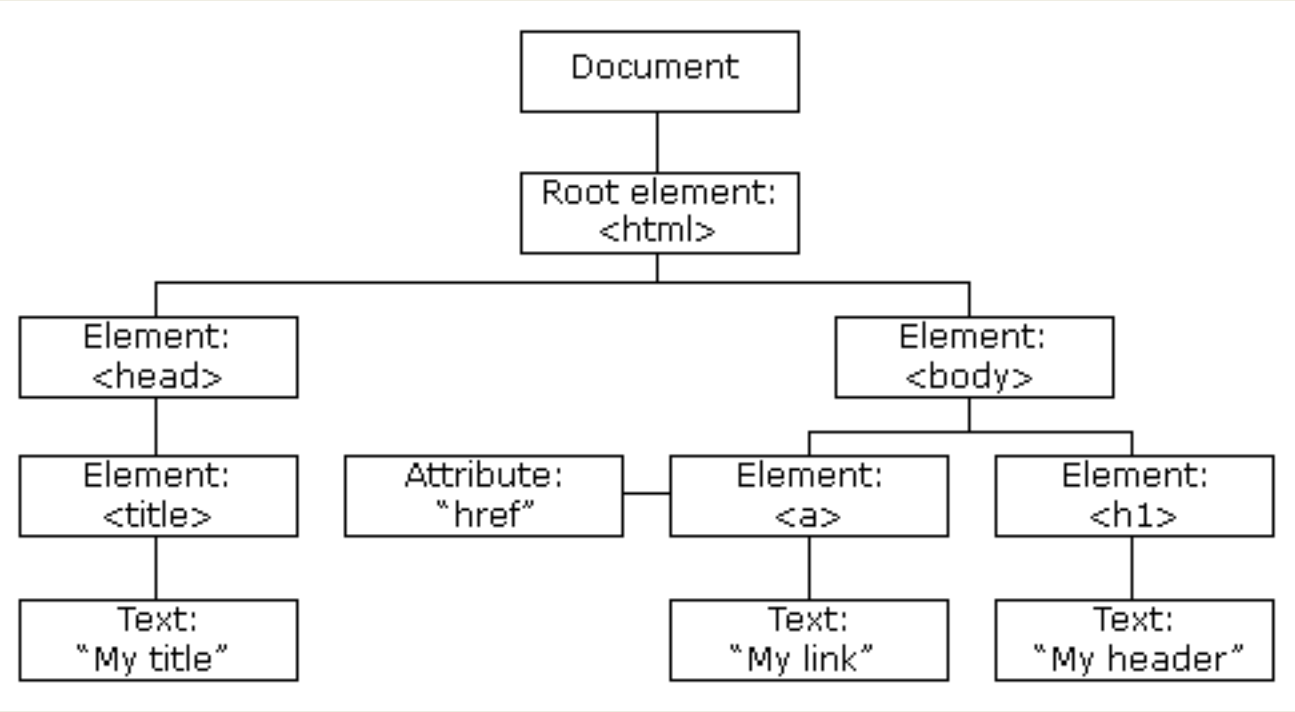
In the HTML:

```
<script src="somecode.js"> </script>
```

This is somecode.js:

```
function changeSrc() {
document.getElementById('demo').style ="font-family:Arial"
document.getElementById('dog-image').src ="ghosted.jpg"
document.body.style.color="green"
}
```

DOM


Document
Root element:
<html>
Element:
<head>
Element:
<body>
Element:
<title>
Attribute:
"href"
Element:
<a>
Element:
<h1>
Text:
"My title"
Text:
"My link"
Text:
"My header"

DOM Basics

- `getElementById(id)` – Method that gets an element with the given id.
- `innerHTML` – Property of an element that is the contents. Allows one to easily change the contents.

```
<script>
 document.getElementById("pick").innerHTML = "New contents"
</script>
```

DOM Basics

- Find an element
  - *getElementsByTagName(name)*
  - *getElementsByClass(class)*

- Change a value
  - *Element.style.property*
  - *Element.attribute*

DOM Basics

- Modifying Structure
  - *`document.removeChild(element)`*
  - *`element.appendChild(element)`*
  - *And others…*
- Event handling
  - *`element.onclick = function(){code}`*
  - *`element.onmouseover = …`*
  - *`element.onmouseout = …`*
  - *`element.addEventListener(event, handler, consume)`*

Event

- ***Listeners*** register for an event with a ***callback*** function.
- ***Event Loop*** waits for events to occur.
  - *When an event occurs appropriate callbacks functions are invoked.*
- In some places a formal callback function isn't needed and we use a chunk of code. (onclick = "...." is an example.)
- Events are associated with the DOM. Typically events are triggered by some mouse, touch, or keyboard occurrence, but there are others. (Native mobile apps have even more events that they can react to like accelerometers for physical movement and low power notifications.)
- Events:
  DOM events

Arrays

- Comma separated values inside [].
- Values don't need to be the same type, but processing is made easier if they are.
- Values are accessed based on position.
- Examples:
  - `let numbers = [1, 2, 3, 4]`
  - `[1, true, "green"]`

Array Access

- Use an index with [].
- Indices are zero based.

```
let letters = ['a', 'b', 'c', 'd']
letters[1] = letters[2]
```

- Changes letters to `['a', 'c', 'c', 'd']`
- Arrays are really objects, so you can set a value at an index that is not currently defined. This can leave empty spaces.
- Use the `.length` property to get the largest index + 1
- Example: `letters.length` would be 4.

Array methods

- `.sort()` sorts the array
- `.reverse()` reorder the array in reverse order
- `.push(value)` adds a value at the end of the array.
- `.pop()` remove and return the value at the end of the array.
- `.toString()` comma separated values in the array.
- `.join(separator)` like toString, but use the separator instead of a comma.
- `.splice(location, number, values…)` – Insert some values at the given location after removing the number of values.
  - *Example :* `stuff.splice(3, 2, "a", "b", "c", "d")` *replaces the values at* `stuff[3]` *and* `stuff[4]` *with* `"a", "b", "c", "d"`
- `.concat(other1, other2, …)` return a new array with the other arrays concatenated at the end.

Iteration

- C style loop
  - `for( i=0; i<array.length; i++) { code using array[i] }`
- For of – Iterate over the values of the structure (can be applied to objects as well).
- For in – iterate over the keys of the structure (can be applied to objects as well)

Iteration Examples

:

```
let numbers = [1, 2, 3, 5]

for(i=0; i<numbers.length; i++){
    console.log("Value in numbers is " + numbers[i])
}


for ( key in numbers) {
 console.log("value is " + numbers[key] )
 /* notice the access */
}

for ( value of numbers) {
  console.log("value is " + value)
}
```

Functionals

- Functional with callbacks. There are more than what are listed here, but these are the common ones.
- `array.forEach(callback)` – apply callback on each value
- `array.map(callback)` – new array with callback applied to each value
- `array.filter(callback test)` – new array with values kept depending on the result of the test
- `array.reduce(start, combine function)` – combine start and values to reduce array to single value

- The callback function takes one or three arguments
  - `function(value)`
  - `function(value, index, array)`

Functional Iteration Examples

:

```
numbers.forEach(logValue)
function logValue(value){
    console.log("Value in numbers is " + value)
}

let doubles = numbers.map(doubleMe)
console.log(doubles)
function doubleMe(value){
    return 2*value
}

let sum = numbers.reduce(0, addUP)
function addUP(accumulator, value){
    return accumulator + value
}
```

Nested Structures

- The values don't have to be primitives but can be structures themselves (objects or arrays).
- Examples:
- ```
  [{"name":"Betty", "age":42},
  {"name":"john, "wages":25.5}]
  ```
- ```
  { "sport":"100 meter",
  "athlete":"Bobby Jones",
  "times":[10.1, 9.75, 13.12, 11.1] }
  ```

Functions

- `function  name( parameters…) { body }`
- Function is exited on a return.
- Returns can send back a value.

- Example: `smaller` is the name, `smaller(2,3)` is an invocation of the function on arguments.

```
function smaller(x,y) {
  if (x<y) return x
  else return y
}
```

Strings

- Use matching pairs of `""` or `''`.
- Backslash for escape (standard C style)
- Iterable
- We can create a primitive string or a string object.

```
let primitiveString = "some string";
let stringObject = new String("some string");
```

- These are ==, but not ===.
- Prefer primitives over objects (Similar advice for Boolean/Number).

Strings

- `.length` for the length.
- `.indexOf(string)` or `.search(regExp)` for searching
- `.slice(index,index)` or `.substr(index,count)`
- `.replace(regExp, str2)` returns
- Regular expressions allow for more complicated pattern matching. Where a regular expression is allowed, you can use a plain string to match. Read about regular expressions here

Number

- We just have one way to represent numbers internally using IEEE 754 64 bit floating point.
- Do not have "infinite" precision integers. Floating point precision is limited and not suitable for all applications.
- If you give an arithmetic operator a string, JavaScript will attempt to convert the string to a number and then perform the operation.
  - *Except for + which is concatenation for strings.*
- NaN – Not a number. Infinity – divide by zero or out of range.
- 0x prefix for a hexadecimal number.

Math

- The math object has predefined constants and methods that we can use.
- Example:
  - `Math.pow(2,3)` – *computes 2^3.*
- See the various scientific methods here and random number methods here.

Cookies

- In the beginning, HTTP was a stateless protocol. Each request for a page opened a new connection and the server did not remember the client/requests that it received.
- But we really want to keep state information to implement things like
  - *Authentication. I log in and then have access to other pages*
  - *Shopping carts. The contents of the cart are remembered as I move between pages.*
  - *Progress. As I move through a resource, where I am is remembered.*

Cookies

- A solution is to use a "cookie" that holds some state information.
  - *A cookie is returned from a server as part of a request*
  - *The cookie is stored on the client side and is associated with a particular domain*
  - *When a request is made to a domain and a cookie exists, it is sent with the request. This allows the server to specialize the response.*
- Cookies hold property value pairs
  - *We expect a unique user or session identifier*
  - *Permanent cookies will have an expiration date*
  - *Hold other personal information like form fields for autofill.*
  - *Limited size – 4K*
  - *Accessible via DOM as document.cookie*
- A session cookie will be removed once the session is finished. Browser closes or we no longer have pages open for that domain.
- A permanent cookie will persist over sessions.

Cookies

- Cookies can be used to track your activity on the web. This can be problematic, though companies will claim that they are only using this information to better serve you as the user. A typical use is to display ads that are targeted using data generated from previous requests.
- First party cookies are used to store information needed to personalize the pages that are provided by the domain for that particular user.
- Third party cookies are typically permanent and are associated with a domain that is not the one serving the page, but is serving some request on the page (often an ad). Usually used to track the sites visited.
- Cookies may not be encrypted and should be considered insecure.
- Discussion of cookies is *here*.

Local Storage

- Another recent option to store data is to use localStorage or sessionStorage objects.
- These are kept on the kept on the client side and are domain specific. Visible to all pages from that domain.
- Can store more information than a cookie.
- Can keep settings/preferences.
- Stores information in pairs, but the value is ***always*** a string.

Local Storage - Example

```
<script> function addIn() {
  let localCount = localStorage.clicks
  if(localCount == undefined){
    localCount = 1;
  } else {
    localCount = Number(localCount) + 1
  }
localStorage.clicks = localCount

displaylocal()
}
```

Clicks was a property I added.

localCount will be a string
Conversion is required so
the + is not a concatenation

Anonymous Functions

- Sometimes it is useful to define a quick function without having to give it a name. Often as a callback function being passed as an argument.
- Examples

```
function (arg) {
  return "anonymous" + arg
}
```

Inside function "this" is the object that called me

```
arg => {
  return "anonymous" + arg
}
```

Using arrow notation, "this" is usually undefined.

Arrow notations:
() =>
arg =>
(arg) =>
(arg1 arg2) =>

setTimeout Example

```
function buttonPress() {
  setTimeout( function () {
     document.getElementById('target').style.color = "red"
  } , 3000
)}
```

Example: Add an element

```
function makeNewDiv(){
  /* make the div and its text */
  let element = document.createElement("div")
  element.innerHTML = "an item after 3"
  element.style.color = "orange"

  /* add it to the container */
  let parent = document.getElementById('target')
  parent.appendChild(element)
}
```

createElement of type, then set its properties
Finally, append it into the children of the target.
If we acquire a target element, we can replace it
Or remove it.

Asynchronous Functions

- We often want our web apps to feel interactive even while they are performing work
  - *Network I/O*
  - *Heavy computation*
- Javascript functions are *blocking*; they will not continue executing the next line until the current line has completed
- Calling a function asynchronously allows us to run a task in the background while still doing other tasks
  - *This improves the interactivity of your web app*

Promises

- Promises enforce sequencing with an asynchronous method.
- Ends in one of two states fulfilled and rejected, until then it is pending.
- When in an end state, it has a result which is either result value or an error object.
  - `my_promise.then(successCB, errorCB)`
  - `my_promise.then(callback).catch(errorCB)`
- Then/catch/finally return a promise so we can chain
  - `my_promise.then(cb1).finally(cb2).then(cb3)`
- Finally adds code to be executed at the end of the chain.

Async/Await

- Another way of handling asynchronous calls.
  - *Async makes a function return a promise.*
  - *Await makes a function wait for a promise*
- Await is only allowed inside an async function.
- resolve() and reject() are predefined for a promise.
- The following are the same:
- ```
  async function h(){
    return “done”}
  ```
- ```
  function h() {
    return Promise.resolve(“done”)}
  ```

Async/Await Example

```
async function itsAPromise() {
  if(good)
    return Promise.resolve("Happy")
  else
    return Promise.reject(
      new Error("Fail") )
)

itsAPromise().then(
   successCB,
   errorCB )
```

Async/Await Example

```
async function myDisplay() {
  let myPromise =
    new Promise(function(myResolve, myReject) {
       myResolve("I love You !!");
  });

  document.getElementById("demo").innerHTML
     = await myPromise;
}

myDisplay();
```

Only in the context of an async function

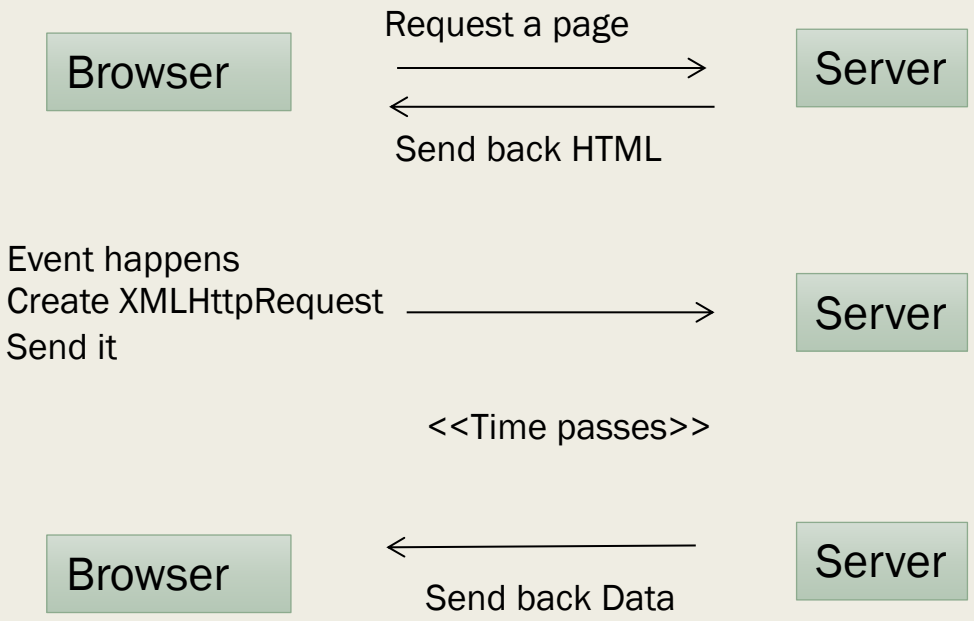
Requests

- Key idea: As pages get more complicated, we want to allow for the possibility that we request pieces of information from the server and then use that information to change/add an element using javaScript.
- Faster because we don't have to get a full HTML document.
- Response is usually XML or JSON

AJAX

- Asynchronous JavaScript And XML.
- Based on the class XMLHttpRequest
- "GET" or "POST" type for http request
- The name is a lie; AJAX is often used to describe any request (that may return JavaScript instead of XML)

AJAX


Request a page
Browser
Server
Send back HTML
Event happens
Create XMLHttpRequest
Send it
Server
<<Time passes>>
Browser
Server
Send back Data
JS Callback is invoked
Process data
Update page

AJAX Example

```
function loadDoc() {
  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
     document.getElementById("demo").innerHTML =
                                        this.responseText;
    }
  };

  xhttp.open("GET", "ajax_info.txt", true);
  xhttp.send();
}
```

Set a call back with
An anonymous fn

Request for ajax_info.tex using GET asynchronously. Send it.

AXIOS

- Promise based library
- Can make XMLHttpRequests from the browser
- Works with node.js (more on this later) to support http requests
- Must be installed as a dependency.
- Automatically transforms JSON responses

AXIOS methods

- Different methods corresponding to the different kinds of requests, E.g. GET, POST, …
- URI as first argument, send payload as second if needed
- May build a configuration and use that with axios.

AXIOS response

- data – the payload (xml/JSON)
- status – HTTP code (200 is ok, 404 page not found) More status codes here.
- statusText
- headers
- config – the request configuration
- request - the request object

AXIOS Example

```
const axios = require("axios")
```

Dependency: require to use

```
axios.get('http://some.place.com/service').then(resp => {
  console.log(resp.data);
  document.getElementById("target").innerHTML =
                                  process(resp.data);
}).catch(err => {
   console.log(err);
   document.getElementById("target").innerHTML =
                                   processErr(err);
});
```

axios.get() returns a promise. We supply two anonymous callback functions to handle success and failure on the get. We could have done this using an asynch function with await.

Fetch

- Built-in API.
  - *No need to install*
  - *Not required as a dependency*
  - *Promise based*

Fetch Example

```
fetch('http://some.place.com/service')
    .then( resp => {
           console.log(resp)
          return resp.json()})
     .then ( data => {
           document.getElementById("target").innerHTML =
                process(data)
});
```

The body of the Response object is a stream which is consumed by .json()

fetch() returns a promise that is a Response object. We deserialize the body and send it onward to be processed and used. Could add in a catch for error.

References

- W3 Schools DOM
- More on promises