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.*
  - *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.*

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 & Data</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:

```
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.*
- Local to 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 undefined outside the function. Inside the function the local x 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

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.

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:

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.

Object Access

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

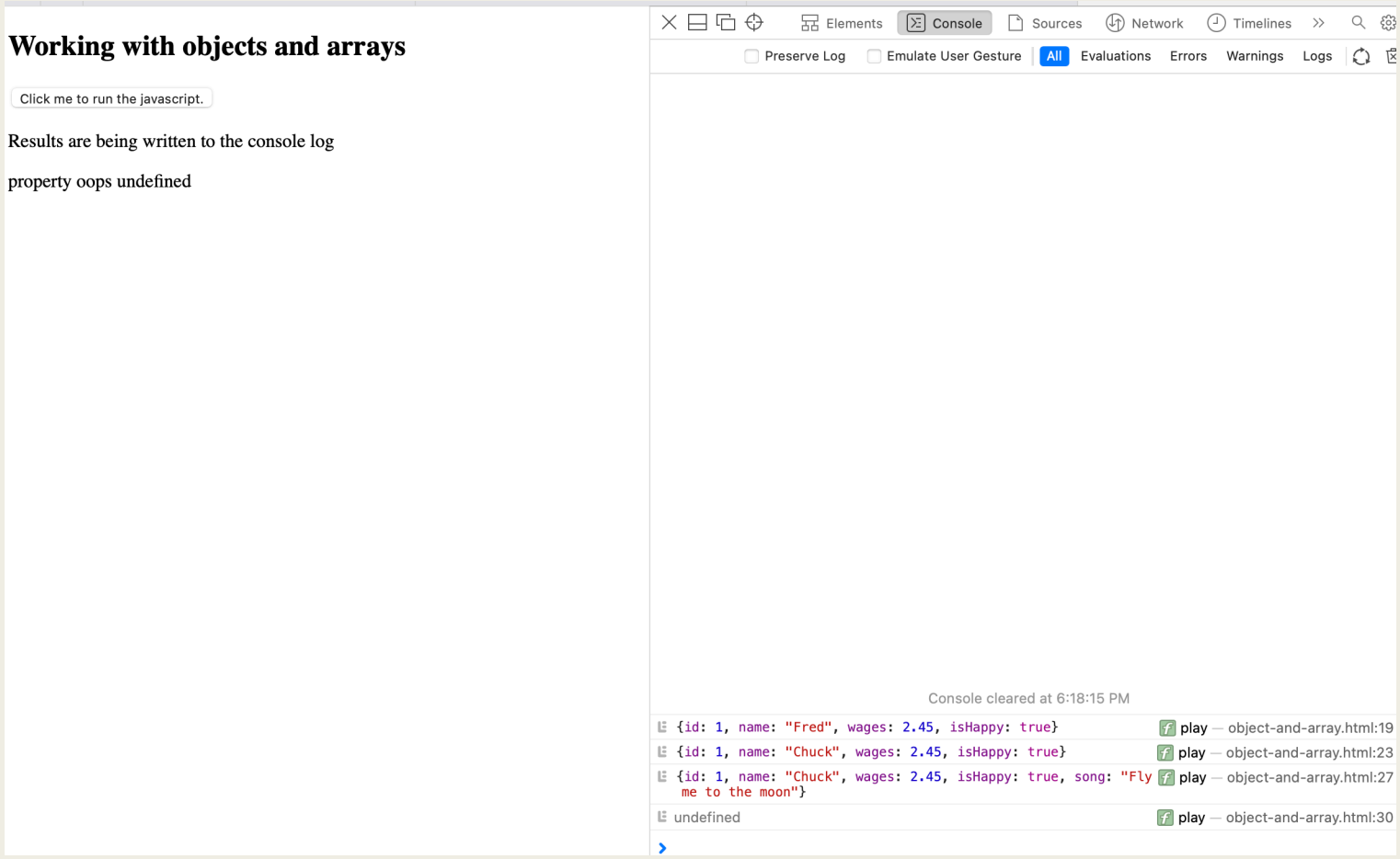
- Two ways we can access the values
  - *let name = object.name*
  - *let name = object[“name”]*
- We can update similarly
  - *object.name = “Chuck”*
  - *object[“name”] = “Chuck”*
- We can extend with a new property
  - *object[“song”] = “Fly Me To The Moon”*

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.

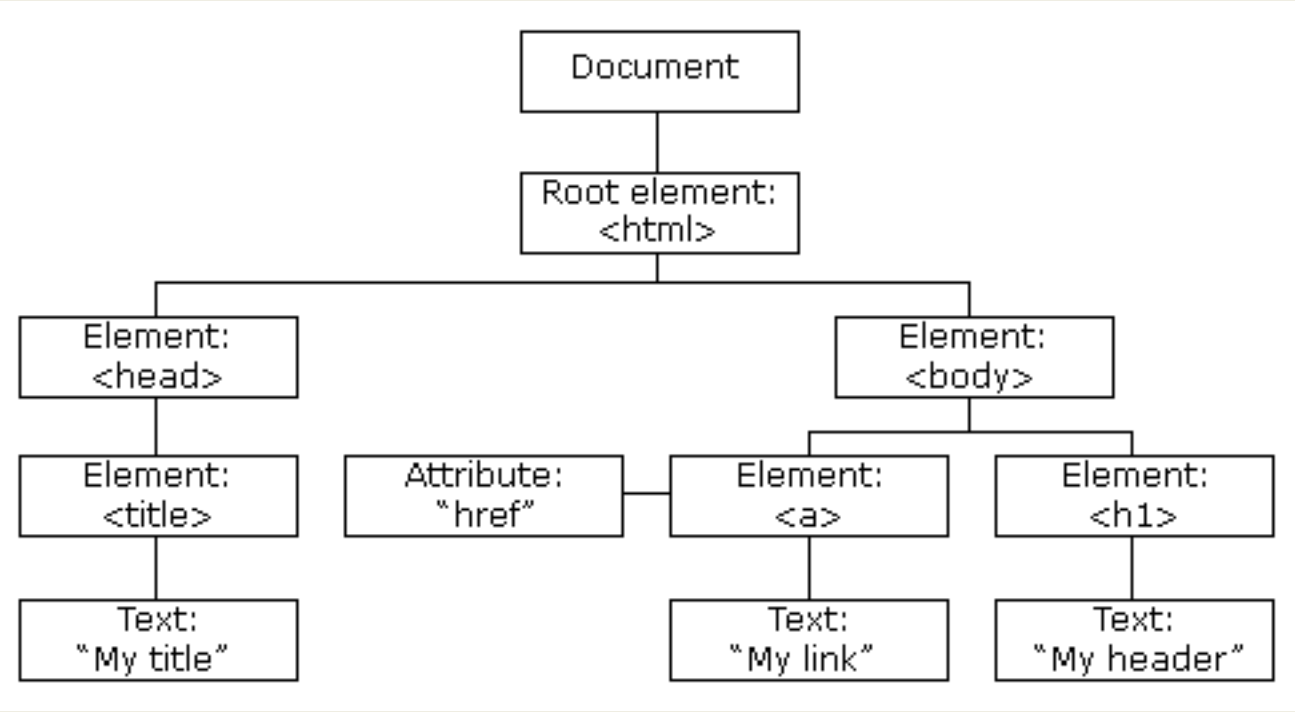
In the HTML:

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

This is somecode.js:

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

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(orther, other, …) 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)
- .
- 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**(combine function) – combine accumulator and value to reduce array to single value

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

Iteration Examples

:

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

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**(combine function) – combine accumulator and value 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.map(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.

Promises

- https://javascript.info/promise-basics

References

- W3 Schools DOM