

Javascript

Because ECMAScript sounds horrible

Javascript

- Javascript is a general purpose programming language.
- It usually runs within a browser
 - Node.js runs Javascript in a server / application context
- Developed in the mid nineties as a simple way to provide interactivity to web pages.
- Originally developed by Brendan Eich working at Netscape
- Submitted to ECMA standards body in 1996
- ECMAScript 5.1 released in 2011

Javascript In A Browser

- REPL
- Read-Eval-Print Loop
- All major browsers have a Javascript REPL system in the console

Javascript In A Browser

```
Elements Network Sources Timeline Profiles Resources Audits |Console|   
```

```
< top frame> ▾
> 1 + 2
< 3
> a = "hello"
< "Hello"
> b = ['a', 'b', 'c', 'd']
< ["a", "b", "c", "d"]
> b[2]
< "c"
> o = {name: 'Mark', class: 'CS337'}
< Object {name: "Mark", class: "CS337"}
> o.name
< "Mark"
> document
< #document
<!DOCTYPE html>
  <html>
    <head>
      <body>
        </body>
    </html>
>
```

Documentation

http://ecma262-5.com/ELS5_HTML.htm

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>

Data Types

- Basic Data Types
 - number
 - boolean
 - string
 - object

Data Types

- `typeof` unary operator
- lets us know what we're dealing with
- If you're evaluating a complex operation, you need parenthesis. Not because `typeof` is a function, but to make sure that there's only one argument to `typeof`

A screenshot of the Chrome Developer Tools' JavaScript console. The output shows the result of the `typeof` operator for different variables:

```
<top frame>
> s = "Hello"
<"Hello"
> typeof s
<"string"
> typeof true
<"boolean"
> typeof 4
<"number"
> typeof 3.1415
<"number"
> typeof("Hello")
<"string"
> typeof(5/2)
<"number"
> typeof 5/2
<"number"
> Nah
<
```

Numbers

- Javascript has a single number datatype to deal with all numbers.
- No distinction between integers, floats, doubles, etc.
- All numbers are represented as floating point numbers, but if the fractional part is zero, they're shown as integers.

A screenshot of the Chrome Developer Tools' JavaScript console. The output shows the result of the `typeof` operator for variables containing floating-point numbers:

```
<top frame>
> a = 10/3
<3.3333333333333335
> b = a - 3
<0.3333333333333335
> c = a - b
<3
> typeof a
<"number"
> typeof c
<"number"
>
```

Numbers

- Numbers stored in variables are converted objects when needed, to have methods and properties
- `Number.toString()`
- `Number.toPrecision()`

A screenshot of the Chrome Developer Tools' JavaScript console. The output shows the result of calling `toString` and `toPrecision` on a number variable:

```
<top frame>
> n = 42
<42
> n.toString()
<"42"
> f = 3.1415
<3.1415
> f.toFixed()
<"3"
> f.toPrecision(3)
<"3.14"
>
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

Strings

Chrome

- A series of zero or more characters.
- Unicode support is pretty good.
- Browser support for full unicode support is spotty.

Safari

The screenshot shows the Chrome Developer Tools console. It displays the following code and its execution results:

```
> a = "is for Apple";
< "is for Apple"
> r = "José Núñez";
< "José Núñez"
> a = "U an apple!";
< "U an apple!"
```

Below the console, tabs for Resources, Timelines, Debugger, and Console are visible. The title bar indicates "Developer Tools - http://www.d...

The screenshot shows the Safari Developer Tools console. It displays the following code and its execution results:

```
> a = "U an apple!";
< "U an apple!"
>
```

Below the console, tabs for Console, Search, Emulation, and Rendering are visible. The title bar indicates "Developer Tools - ht...

Strings

- String variables are also converted to objects as needed.
- `String.toUpperCase()`
- `String.substring(start, end)`
- Note the difference between `.substring()` and `.length`
 - One is a method, one is a property

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Boolean

- Boolean for `true` and `false`.
- Comparisons
- Coerce other datatypes into Boolean.
- Note the behavior of the Boolean value for strings.
 - Empty string is `false`
 - Other strings are `true`. Even "`false`"!

The screenshot shows the Chrome Developer Tools console. It displays the following code and its execution results:

```
> typeof true
< "boolean"
> 8 < 4
< false
> Boolean(false)
< false
> Boolean(0)
< false
> Boolean("")
< false
> Boolean("true")
< true
> Boolean("false")
< true
```

Below the console, tabs for Elements, Network, and Sources are visible. The title bar indicates "Developer Tools - f...

Variables

- Variable names can be any combination of letters, numbers, an underscore (_), or \$
- Variable names cannot start with a number.
- Variables do not need to be declared.
- The `var` keyword can be used to declare and scope variables.

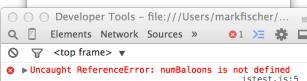
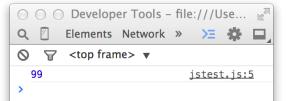
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var>

Variables

- Variables have global scope unless `var` is used to declare a variable.

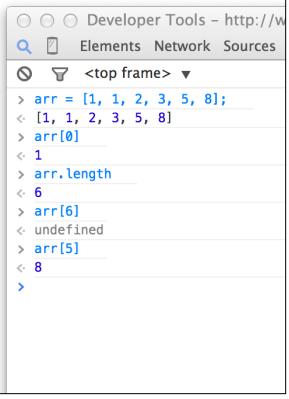
```
var foo = function() {  
    numBalloons = 99;  
}  
foo();  
console.log(numBalloons);
```

```
var foo = function() {  
    var numBalloons = 99;  
}  
foo();  
console.log(numBalloons);
```



Arrays

- Collection of values
- Created with `[n, n+1,...k-1]` syntax
- Array access with brackets: `n[]`
- Length property
- Standard Zero based indexing



Arrays

- Arrays can be collections of many different datatypes.

The screenshot shows a browser developer tools console window. At the top, there are three circular icons (yellow, green, blue) followed by the text "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples/...". Below the tabs are buttons for Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Console tab is active, indicated by a blue border. The console output shows the following code and its execution:

```
> var f = function() { return 1; }
< undefined
> f
< function () { return 1; }
> arr = [1, "two", {name: "three"}, f];
< [1, "two", <Object {name: "three", value: 1}>, function () {return 1;}]
    name: "three"
    > __proto__: Object
>
```

Arrays From Strings

- `String.split()` to create an array from a string.

The screenshot shows a browser developer tools console window. At the top, there are three circular icons (yellow, green, blue) followed by the text "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples/...". Below the tabs are buttons for Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Console tab is active. The console output shows the following code and its execution:

```
> s = "983,Name,ID,OrderStats,9,15";
< "983,Name,ID,OrderStats,9,15"
> orderDetails = s.split(",");
< ["983", "Name", "ID", "OrderStats", "9", "15"]
> typeof orderDetails
< "object"
> orderDetails[1]
< "Name"
>
```

Array Methods

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

- Lots of useful array methods.
- `.contains(<some value>)` // returns true or false
- `.join(<glue string>)` // joins all elements together with glue and returns a string.
- `.toString()` // Quick string representation of the array
- `.pop() .push() .shift() .unshift()` // Standard array methods
- `.sort()` // Sorts elements according to criteria
- `.splice()` // Adds or removes elements from an array

Array Assignment

- Assigning an array to another variable assigns a reference of the array to the variable, not a copy.

A screenshot of a browser's developer tools console. The code shown is:

```
> arr1 = ["apples", "bananas"];
< ["apples", "bananas"]
> arr2 = arr1
< ["apples", "bananas"]
> arr1.push("kiwi")
< 3
> arr1
< ["apples", "bananas", "kiwi"]
> arr2
< ["apples", "bananas", "kiwi"]
>
```

Array Assignment

- To make a copy of an array, use the `.slice(0)` method.

A screenshot of a browser's developer tools console. The code shown is:

```
> arr1 = ["apples", "oranges"];
< ["apples", "oranges"]
> arr2 = arr1.slice(0);
< ["apples", "oranges"]
> arr1.push("bananas");
< 3
> arr1
< ["apples", "oranges", "bananas"]
> arr2
< ["apples", "oranges"]
>
```

undefined

developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined

- Javascript has a special value for things that are not defined: `undefined`
- Out of bounds requests
- Un-initialized variables
- `undefined` is a property of the *global object*. Its type is `undefined`.

A screenshot of a browser's developer tools console. The code shown is:

```
> arr = [1, 2, 3]
< [1, 2, 3]
> arr[4]
< undefined
> typeof b
< "undefined"
> q
✖ > ReferenceError: q is not defined
> b.toString()
✖ > ReferenceError: b is not defined
> u = undefined
< undefined
> typeof u
< "undefined"
>
```

Objects

- Objects are very flexible data structures.

- A basic object:

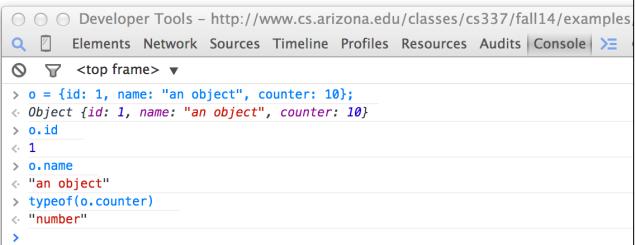
```
o = {id: 1, name: "an object", counter: 10};
```

- Create property names and values using **key: value** syntax.
- Separate multiple properties by commas.

Objects

```
o = {id: 1, name: "an object", counter: 10};
```

- Access properties via dot syntax



```
Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples
Elements Network Sources Timeline Profiles Resources Audits | Console | >≡
< top frame > ▾
> o = {id: 1, name: "an object", counter: 10};
< Object {id: 1, name: "an object", counter: 10}
> o.id
< 1
> o.name
< "an object"
> typeof(o.counter)
< "number"
>
```

Objects

```
o = {id: 1, name: "an object", counter: 10};
```

- Act as “Associative Arrays” or “Key / Value” arrays, or “Dictionary” array
- arr["key"] syntax



```
Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples
Elements Network Sources Timeline Profiles Resources Audits | Console | >≡
< top frame > ▾
> o
< Object {id: 1, name: "an object", counter: 10}
> o["id"]
< 1
> typeof(o["name"])
< "string"
>
```

Objects

- Assigning to undefined properties creates them

A screenshot of the Developer Tools Console window. The title bar says "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples/...". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Console tab is selected. The console output shows:

```
> o
< Object {id: 1, name: "an object", counter: 10}
> o.desc = "A description"
< "A description"
> o
< Object {id: 1, name: "an object", counter: 10, desc: "A description"}
>
```

null

developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null

- Null is a literal value representing an “empty” or non-existent value.

A screenshot of the Developer Tools Console window. The title bar says "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples/...". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Console tab is selected. The console output shows:

```
> o = {id: 1, name: "an object", callback: null}
< Object {id: 1, name: "an object", callback: null}
> o.callback
< null
>
```

Operators

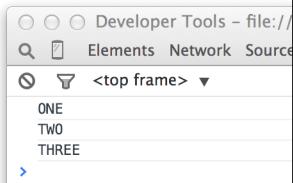
- Arithmetic Operators: + - * % ++ --
- String concatenation: +
- Logical Operators: && || !
- Comparisons: < > <= >=
- Ternary Operator: condition ? true expr : false expr
- Bitwise Operators: << >> ^ ~

Control Structures

- if (condition) { stmt1 } else { stmt2 }
- while (condition) { statements }
- for (i = 0; i < 10; i++) { statements }
- Pretty much work like every other C or Java style language

Control Structures: forEach

```
a = ["one", "two", "three"];
a.forEach(function(element, index, arr) {
  console.log( element.toUpperCase() );
});
```



- Arrays have a special **forEach** method for performing some action relating to each element of the array
- The **forEach** method takes a *function* as an argument.

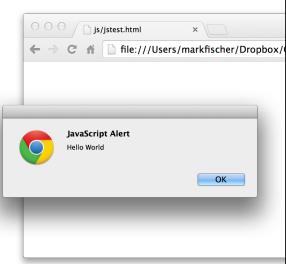
Basic I/O

- Alerts
- Log to Console
- Confirms
- Prompt
- DOM Manipulation
- Debugger
- No Direct Local File I/O!

alert()

```
alert("Hello World");
```

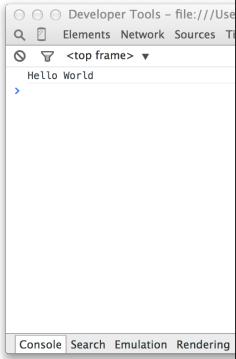
- Display a modal dialog box with the specified text.
- Pauses execution of Javascript until dialog is dismissed.



console.log()

```
console.log("Hello World");
```

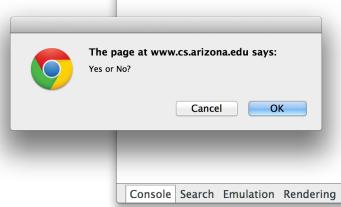
- Quick way to get some debugging out.
- Doesn't block execution, so usually a better choice for debugging and testing than `alert()`.



confirm()

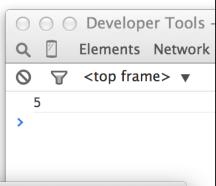
```
confirm("Yes or No?");
```

- Ask for a `true` or `false` response from the user.

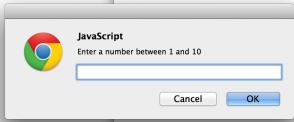


prompt()

```
prompt("Enter a number between 1 and 10");
console.log(i);
```

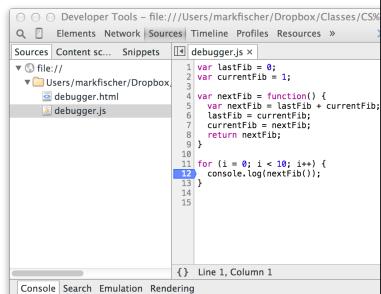


- Ask for user input as a text string.



Debugger

- Most browsers have a full featured interactive debugger built in.
- Breakpoints, watched expressions, step through execution, etc.
- Example.



Functions

```
function echo(a) {
    return a;
}

echoTwo = function(a) {
    return a;
}

var echoThree = function(a) {
    return a;
}

console.log( echo("one") );
console.log( echoTwo("two") );
console.log( echoThree("three") );
```

- Multiple ways to define a function

Functions

Declares a named function without requiring assignment

```
function echo(a) {  
    return a;  
}  
  
echoTwo = function(a) {  
    return a;  
}  
  
var echoThree = function(a) {  
    return a;  
}  
  
console.log( echo("one") );  
console.log( echoTwo("two") );  
console.log( echoThree("three") );
```

Declares a *global* variable echoTwo and assigns an anonymous function to it

Declares a *local* variable echoThree and assigns an anonymous function to it

Functions

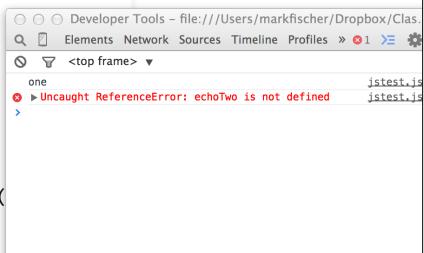
- Does any of this matter?
- What if we call the functions before they're declared?

```
console.log( echo("one") );  
console.log( echoTwo("two") );  
console.log( echoThree("three") );  
  
function echo(a) {  
    return a;  
}  
  
echoTwo = function(a) {  
    return a;  
}  
  
var echoThree = function(a) {  
    return a;  
}
```

Functions

```
console.log( echo("one") );  
console.log( echoTwo("two") );  
console.log( echoThree("three") );
```

```
function echo(a) {  
    return a;  
}  
  
echoTwo = function(a) {  
    return a;  
}  
  
var echoThree = function(  
    return a;  
}
```



Functions

- The first style has a symbol table entry created for it at parse time. So it can be referenced immediately during runtime.
- The other two have symbol table entries created at runtime, so aren't available until after they've been executed.

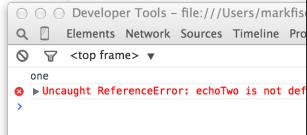
[javascriptweblog.wordpress.com/2010/07/06/
function-declarations-vs-function-expressions/](http://javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions/)

```
console.log( echo("one") );
console.log( echoTwo("two") );
console.log( echoThree("three") );

function echo(a) {
    return a;
}

echoTwo = function(a)
{
    return a;
}

var echoThree = function(a)
{
    return a;
}
```



Functions

```
//Function Declaration
function add(a,b) {return a + b;}
//Function Expression
var add = function(a,b) {return a + b};
```

- So should we always use Function Declarations?
- Well, it depends...

Functions

- What is the console output here?
- ```
function echo(a) {
 return a;
}

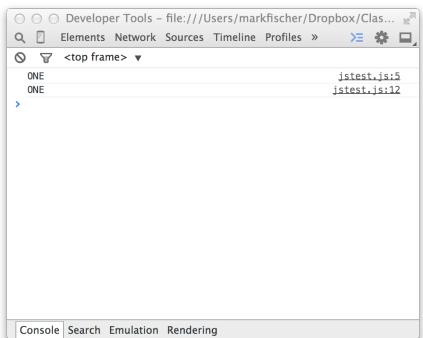
console.log(echo("one"));

function echo(a) {
 return a.toUpperCase();
}

console.log(echo("one"));
```

# Functions

- Hmm, maybe not what we were expecting.
  - Function Declarations are ‘hoisted’ to the top at parse time, so when executed, the last declared version wins.



# Function Declarations

- Can only appear as block level elements.
  - Are ‘hoisted’ to the top at parse time, before run time.
  - Cannot be nested within non-function blocks.
  - Are scoped by where they are declared, like `var`

# Function Expressions

- Can be used anywhere an expression is valid.
    - Can be more flexible because of this.
  - Are evaluated and assigned at run time.

# Objects and Functions

- Functions can be added to objects as property variables.
- Many object “methods” are really properties with functions assigned to them.

---

---

---

---

---

---

---

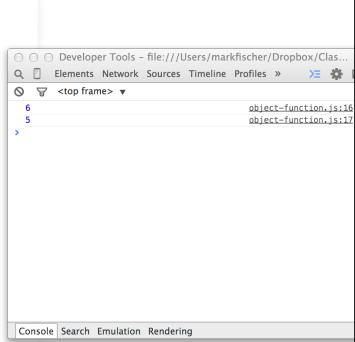
# Objects and Functions

```
var doubleMe = function(x) {
 return 2 * x;
}

var halveMe = function(x) {
 return x/2;
}

var myLib = {
 version: 0.3,
 name: "My Test Library",
 double: doubleMe,
 half: halveMe
}

console.log(myLib.double(3));
console.log(myLib.half(10));
```



---

---

---

---

---

---

---

# Objects and Functions

- Using anonymous function expressions instead.

```
var myLib = {
 version: 0.4,
 name: "My Test Library",
 double: function(x) { return 2 * x; },
 half: function(x) { return x/2; }
}

console.log(myLib.double(3));
console.log(myLib.half(10));
```

---

---

---

---

---

---

---

# Javascript in HTML

- Where does our Javascript live?
- Inline in an HTML document inside a `<script>` element
- Included in an external file via a `<script>` element.

# Javascript in HTML

- The `<script>` element with inline content
- Within the `<script>` element, we're parsing Javascript, not HTML

```
<!doctype html>
<head>
 <title>js/jstest.html</title>

 <script>
 var answer = 42;
 function calculateAnswer() {
 return answer;
 }
 console.log(calculateAnswer());
 </script>
</head>

<body>
 <div></div>
 <div></div>
</body>
</html>
```

# Javascript in HTML

- The `<script>` element with `src` attribute.
- Includes an external file with Javascript in it.
- No wrapping `<script>` tags within external files.

```
<!doctype html>
<html>
<head>
 <title>js/jstest.html</title>
 <script src="jstest.js"></script>
</head>

<body>
 <div></div>
</body>
</html>
```



```
var answer = 42;
function calculateAnswer() {
 return answer;
}
console.log(calculateAnswer());
```

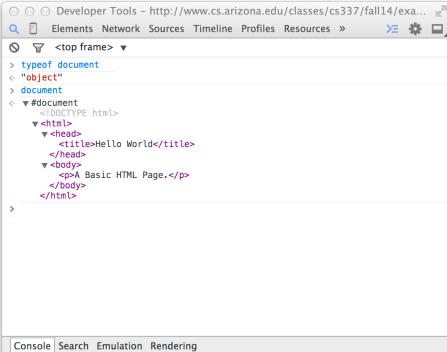
# The document Object

This is all well and good, but how about something involving a web page?

# The document Object

- Browsers parse the HTML and CSS of a page, and build an object model in memory.
- The browser exposes this object to us for use with our Javascript as the **document** object.

# The document Object



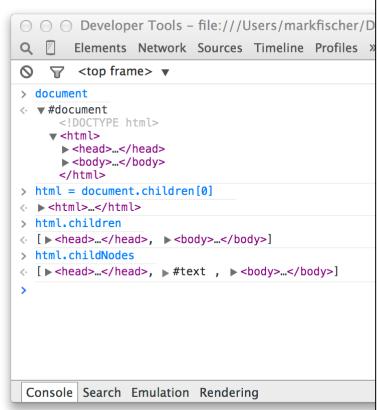
A screenshot of the Chrome Developer Tools Elements tab. The page URL is http://www.cs.arizona.edu/classes/cs337/fall14/exa... . The DOM tree shows the following structure:

```
> <html>
 <head>
 <title>Hello World</title>
 </head>
 <body>
 <p>A Basic HTML Page.</p>
 </body>
</html>
```

At the bottom of the tools, there are tabs for Console, Search, Emulation, and Rendering.

# The document Object

- The document object represents the root element of our DOM tree.
- It has child nodes, and each node has various attributes.
- Note the difference between `.children` and `.childNodes`



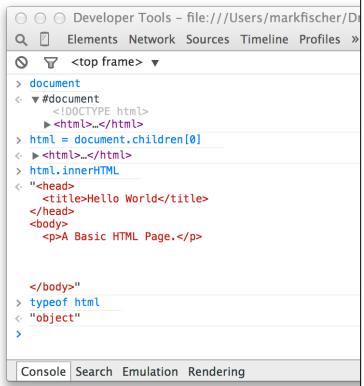
A screenshot of the Developer Tools Elements tab. The tree view shows the following structure:

```
<top frame>
 > document
 <> #document
 <> <!DOCTYPE html>
 > <html>
 > <head>...</head>
 > <body>...</body>
 </html>
 > html = document.children[0]
 <> <html>...</html>
 > html.children
 <> [<head>...</head>, <body>...</body>]
 > html.childNodes
 <> [<head>...</head>, #text , <body>...</body>]
 >
```

At the bottom of the developer tools window, there are tabs for Console, Search, Emulation, and Rendering.

# The document Object

- `document` elements are *objects*, so accessing their properties is done with the dot syntax
- `object.property`
- `html.innerHTML` for example



A screenshot of the Developer Tools Elements tab. The tree view shows the following structure:

```
<top frame>
 > document
 <> #document
 <> <!DOCTYPE html>
 > <html>...</html>
 > html = document.children[0]
 <> <html>...</html>
 > html.innerHTML
 <> "<head>
 <title>Hello World</title>
 </head>
 <body>
 <p>A Basic HTML Page.</p>
 </body>"
```

At the bottom of the developer tools window, there are tabs for Console, Search, Emulation, and Rendering.

# The document Object

- The `document` object is *NOT* part of the Javascript language.
- It is an API defined by the W3C to interact with HTML and XML documents.

[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)

# DOM Selection

- Starting with the `document` root and drilling down via `.children` is tedious. Can we get at elements some other way?
- `document.getElementById("main")`
- `document.getElementsByTagName("p")`
- `document.getElementsByClassName("error")`

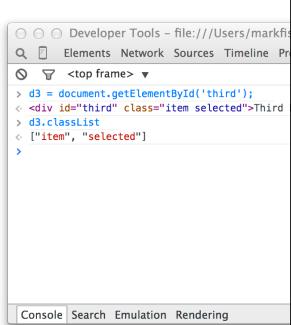
## getElementById

- Gets an `HTMLElement` object from the document based on an ID.
- Since ID must be unique, this method returns a single element, not an array of elements.

## getElementById

```
<!doctype html>
<head>
 <title>js/getElementById.html</title>
 <link rel="stylesheet" type="text/css"
 href="getElements.css" />
</head>

<body>
 <div id="main">
 <div id="first" class="item">
 First Block
 </div>
 <div id="second" class="item">
 Second Block
 </div>
 <div id="third" class="item selected">
 Third Block
 </div>
 </div>
</body>
</html>
```



# Updating the DOM

- Now that we can get an element, can we do something with it?

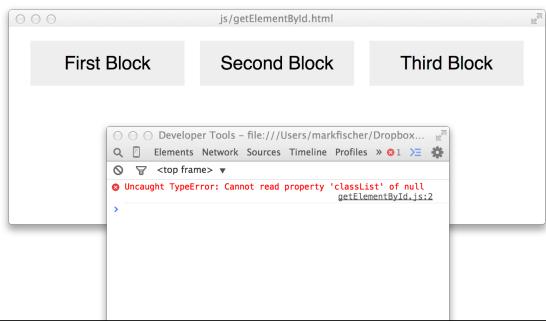
```
<!doctype html>
<head>
 <title>js/getElementById.html</title>
 <link rel="stylesheet" type="text/css"
 href="getElements.css" />
 <script src="getElementById.js"></script>
</head>

<body>
 <div id="main">
 <div id="first" class="item">First Block</div>
 <div id="second" class="item">Second Block</div>
 <div id="third" class="item">Third Block</div>
 </div>
</body>
</html>
```

The diagram illustrates the execution flow of the JavaScript code. A pink box highlights the line `d2.classList.add("selected");`. A curved arrow points from this line to the `classList` property of the element `d2`, which is the second `div` with `id="second"`. Another curved arrow points from the `document.getElementById('second')` call in the code back to the `div` element itself.

# Updating the DOM

- Hmm nothing happened. Why? Check the console.



# Updating the DOM

- Uncaught TypeError: Cannot read property 'classList' of null?? But how can d2 be null?

```
d2 = document.getElementById('second');
d2.classList.add("selected");
```

A screenshot of developer tools showing the same error message: "Uncaught TypeError: Cannot read property 'classList' of null" at the line `d2.classList.add("selected");`. Below the error, the DOM structure is shown with the `div` element having `id="second"` highlighted. The right side of the tools shows the original HTML code.

# Waiting for the DOM to load

- The browser waits for no DOM
- The browser parses the file, loads the `getElementById.js` file, and executes it all before the rest of the HTML is parsed and the DOM is created.

```
<!doctype html>
<head>
 <title>js/getElementById.html</title>
 <link rel="stylesheet" type="text/css" href="getElements.css" />
 <script src="getElementById.js"></script>
</head>

<body>
 <div id="main">
 <div id="first" class="item">First Block</div>
 <div id="second" class="item">Second Block</div>
 <div id="third" class="item">Third Block</div>
 </div>
</body>
</html>
```

# Waiting for the DOM to load

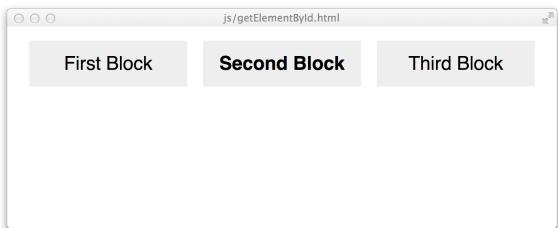
- What if we just move the `<script>` element down to the bottom?

```
<!doctype html>
<head>
 <title>js/getElementById.html</title>
 <link rel="stylesheet" type="text/css" href="getElements.css" />
</head>

<body>
 <div id="main">
 <div id="first" class="item">First Block</div>
 <div id="second" class="item">Second Block</div>
 <div id="third" class="item">Third Block</div>
 </div>
<script src="getElementById.js"></script>
</body>
</html>
```

# Waiting for the DOM to load

- Works!



## Waiting for the DOM to load

- That seems... hackish. Isn't there a "right" way to do this?
- Well, it's perfectly valid. `<script>` elements do not have to go in the `<head>`, although they frequently do.
- However, `<script>` elements that aren't in the `<head>` tend to get overlooked later, so we try to put them there if we can.

## Events

- The web browser is an Event Driven application.
- Documents load, links are clicked, HTTP requests are made and completed.
- Each of these is an event, and we can register event listeners (function) which will be called as these events occur.
- These are called *callbacks*.

## Events

- `object.addEventListener('event', callback);`
- The object can be any object that responds to event listeners, such as an Element, the Document, or maybe the Window.

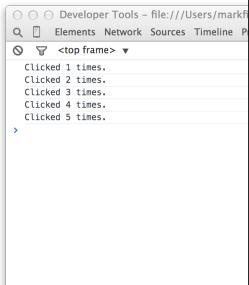
# Events

- A basic example of a 'click' event handler.

```
<!doctype html>
<head>
 <title>js/events.html</title>
 <link rel="stylesheet" type="text/css"
 href="getElements.css" />
</head>

<body>
 <div id="main">
 <div id="first" class="item">First Block</div>
 <div id="second" class="item">Second Block</div>
 <div id="third" class="item">Third Block</div>
 </div>

 <script>
 clickCount = 0;
 d1 = document.getElementById('first');
 d1.addEventListener('click', function() {
 console.log("Clicked " + ++clickCount + " times.");
 });
 </script>
</body>
</html>
```



# Events

- Is it really that simple? What about IE, doesn't that always mess us up?
- Well, yes. Of course it does.
- `object.addEventListener()` didn't come to IE until 9
- Earlier methods for adding event listeners were directly in markup, or via `object.event = callback;`

```
Link
```

# window load Event

- There's also a `window` object that the DOM API provides for us.
- The Window object supports the `load` event, and we can register our own callback with this.
- The `load` event fires once the DOM has completed loading.

## window load Event

```
<!doctype html>
<head>
 <title>js/window-load.html</title>
 <link rel="stylesheet" type="text/css"
 href="getElements.css" />
 <script src="window-load.js"></script>
</head>

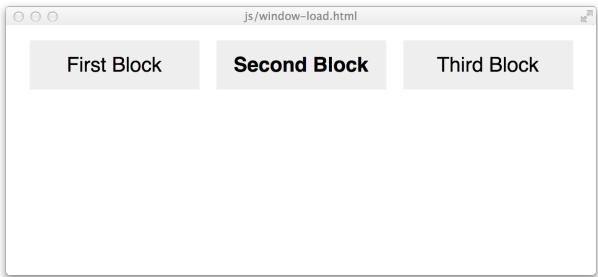
<body>
 <div id="main">
 <div id="first" class="item">First Block</div>
 <div id="second" class="item">Second Block</div>
 <div id="third" class="item">Third Block</div>
 </div>

</body>
</html>
```

```
window.addEventListener('load', function()
{
 d2 = document.getElementById('second');
 d2.classList.add('selected');
});
```

## window load Event

- Works!



## window load Event

- Since addEventListener doesn't work with IE 8 or older, to provide a more robust solution you'd have to do browser capabilities detection.

```
window.addEventListener('load', function()
{
 d2 = document.getElementById('second');
 d2.classList.add('selected');
});
```

# window load Event

- IE 8 supported a different method, the `object.attachEvent` method.
- Even older browsers only support a single "onload" property.
- If only someone would write a library that did all this for us...

```
var ready = function(myFunciton) {
 if (window.attachEvent) {
 window.attachEvent('onload', myFunciton);
 console.log("IE");
 } else if (window.addEventListener) {
 window.addEventListener('load', myFunciton);
 console.log("Modern");
 } else {
 console.log("Legacy");
 if(window.onload) {
 var curr.onload = window.onload;
 var new.onload = function() {
 curr();
 myFunciton();
 };
 window.onload = new.onload;
 } else {
 window.onload = myFunciton;
 }
 }
}
```

## Putting Pieces Together



Demo

## click-count.html

```
<!doctype html>
<head>
 <title>js/click-count.html</title>
 <link rel="stylesheet" type="text/css"
 href="click-count.css"/>
 <script src="click-count.js"></script>
</head>

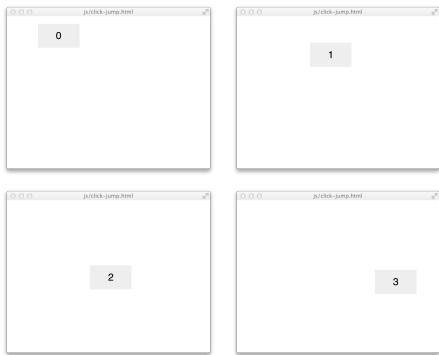
<body>
 <div id="main">
 </div>
</body>
</html>
```

## click-count.html

```
var addCount = function(event) {
 var curCount = Number(this.textContent);
 curCount++;
 this.textContent = curCount.toString();
}

window.addEventListener('load', function() {
 var numBoxes = 9;
 main = document.getElementById('main');
 for (i = 0; i < numBoxes; i++) {
 var newBox = document.createElement("div");
 newBox.textContent = "0";
 newBox.addEventListener('click', addCount);
 main.appendChild(newBox);
 }
});
```

## click-jump.html



## click-jump.html

```
var addCount = function(event) {
 var curCount = Number(this.textContent);
 curCount++;
 this.textContent = curCount.toString();

 if (curCount == 1) {
 this.style.position = "absolute";
 }

 var max_x = window.innerWidth - 110;
 var max_y = window.innerHeight - 60;
 var newX = Math.random() * max_x;
 var newY = Math.random() * max_y;
 newX = Math.floor(newX);
 newY = Math.floor(newY);

 this.style.top = newY.toString() + "px";
 this.style.left = newX.toString() + "px";
}
```

# Inspiration

- Case study on copying stuff from other people.
- <https://account.arizona.edu/welcome>

# Updating Styles

- Many ways of updating DOM elements involves changing its CSS Style Attributes
  - Positioning an element somewhere
  - Changing font styles
  - Changing colors, borders, etc

# element.style

```
element.style.color = "#cccccc";
```

- An HTMLElement object has a style property
- The style property is itself an object, having properties for all the CSS properties appropriate to that element

## element.style

```
size = 2;
element.style.fontSize = size.toString() + "em";
```

- All values are strings. If you have to assign numerical values, you need to convert them to strings.
- You also need to make sure the value is a complete and valid value for the property, including any units

## element.style

```
element.style.fontSize = "2em";
```

- You can't have javascript variable names that contain a dash... its the subtraction operator.
- CSS properties with dashes in their name become camelCased

```
font-size -> fontSize
```

## Timing Events

- Browsers implement Javascript in a threaded environment.
- Events can be queued to fire at a later time.
- `window.setTimeout()`
- `window.setInterval()`

<https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers>

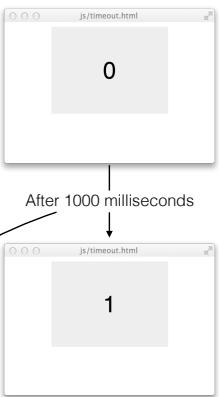
# setTimeout()

```
<!doctype html>
<head>
 <title>js/timeout.html</title>
 <link rel="stylesheet" type="text/css"
 href="timeout.css" />
</head>

<body>
 <div id="main">0</div>

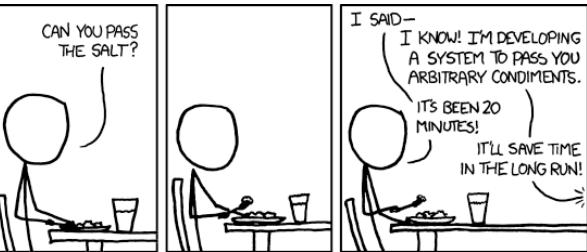
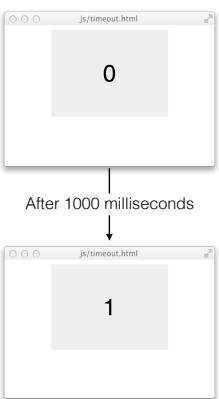
 <script>
 var counter = function() {
 var d = document.getElementById('main');
 var curCount = Number(d.textContent);
 curCount++;
 d.textContent = curCount.toString();
 }

 window.setTimeout(counter, 1000);
 </script>
</body>
</html>
```



# setInterval()

- `setTimeout()` only fires a single time.
- To fire on an interval, use `setInterval()`, or continually call `setTimeout()`.
- Demo



# Classes

Oops, sorry, there are no classes.

---

---

---

---

---

---

---

## Class Like Thingies

- Javascript has no “Class” concept.
- Objects are based on building on a prototype.
- “Instances” are not tied to a particular static Class definition.
- functions?

---

---

---

---

---

---

---

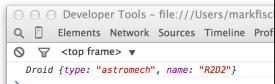
## functions and new

- Classes are just functions!
- Create new instances with the new keyword.

```
function Droid(type, name) {
 this.type = type;
 this.name = name;
}

var r2 = new Droid('astromech', 'R2D2');
var c3 = new Droid('protocol', 'C3PO');

console.log(r2);
```



---

---

---

---

---

---

---

# prototypes

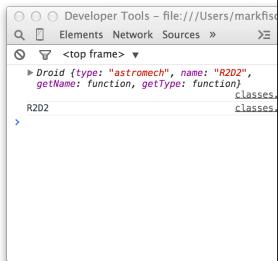
- Methods can be added through the special `.prototype` property of objects.

```
function Droid(type, name) {
 this.type = type;
 this.name = name;
}

Droid.prototype = {
 getName: function() { return this.name; },
 getType: function() { return this.type; }
}

var r2 = new Droid('astromech', 'R2D2');
var c3 = new Droid('protocol', 'C3PO');

console.log(r2);
console.log(r2.getName());
```



# prototypes

- Don't like the behavior of something? Re-define it on the fly

```
function Droid(type, name) {
 this.type = type;
 this.name = name;
}

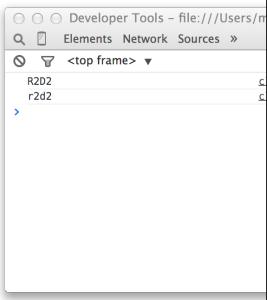
Droid.prototype = {
 getName: function() { return this.name; },
 getType: function() { return this.type; }
}

var r2 = new Droid('astromech', 'R2D2');
var c3 = new Droid('protocol', 'C3PO');

console.log(r2.getName());

Droid.prototype.getName =
 function() { return this.name.toLowerCase(); };

console.log(r2.getName());
```



# myQuery

- jQuery is a very popular Javascript toolkit which abstracts away some of the underlying complexity.
- Can we build our own simple toolkit?
- Of course we can...
- jQuery doesn't own \$

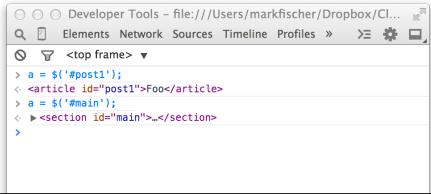
# Basic Selection

- Using `document.getElementById()` isn't too bad, but it sure is a lot of typing.
- Can we use the `$( 'selector' )` pattern?

```
var $ = function myQuery(selector) {
 // See if selector starts with a #. If so we're looking for an ID
 if (selector[0] == '#') {
 // Strip off the # sign
 var selector = selector.substring(1, selector.length);
 var element = document.getElementById(selector);
 return element;
 }
}
```

# Basic Selection

```
var $ = function myQuery(selector) {
 // See if selector starts with a #. If so we're looking for an ID
 if (selector[0] == '#') {
 // Strip off the # sign
 var selector = selector.substring(1, selector.length);
 var element = document.getElementById(selector);
 return element;
 }
}
```



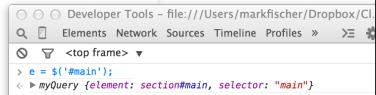
# Returning Objects

```
function myQuery(selector) {
 this.element = null;
 this.selector = selector;

 // See if selector starts with a #.
 // If so we're looking for an ID
 if (selector[0] == '#') {
 // Strip off the # sign
 var selector = selector.substring(1, selector.length);
 var element = document.getElementById(selector);

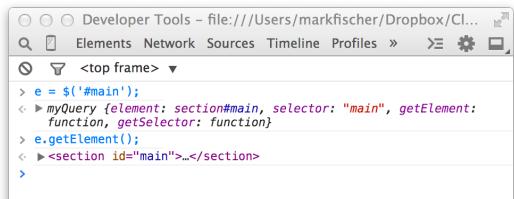
 myQobj = new myQuery(selector);
 myQobj.element = element;
 return myQobj;
 }

 var $ = myQuery;
}
```



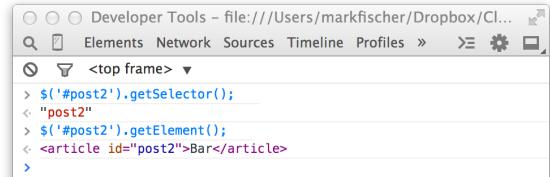
# prototype Methods

```
myQuery.prototype = {
 getElement: function() {
 return this.element;
 },
 getSelector: function() {
 return this.selector;
 },
}
```



## Function Chaining

- Supports function chaining.
- The return value from the function call is an object, which has methods we can call.
- Don't need intermediate variables.



## jQuery

- This is basically what jQuery does.
- More methods and selector types.
- There's a lot more edge cases handled, and checks made.
- jQuery 'plugins' just add their own function calls to the jQuery `prototype` property.

<http://code.jquery.com/jquery-2.2.1.js>

And now for something  
moderately different