

# JavaScript Crash Course

Sumner Evans and Sam Sartor

November 10, 2016



Colorado School of Mines  
Linux Users Group

# JavaScript is **NOT** Java <sup>1</sup>

- JavaScript was written was created in 10 days in May 1995 by Brendan Eich.
- JavaScript was originally called Mocha and was renamed to LiveScript before being renamed again to JavaScript.
- Why **JavaScript**? Because Java happened to be popular then (that was before people realized how much Java sucks in a browser) and JavaScript looks syntactically similar at a glance.
- JavaScript is standardized<sup>2</sup> by Ecma International and there have been a number of ECMAScript versions. The latest is ECMAScript 6, but it is not fully supported by any browsers, including Firefox which only has partial support.

---

<sup>1</sup> Lots of this slide's information is from: [https://www.w3.org/community/webed/wiki/A\\_Short\\_History\\_of\\_JavaScript](https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript)

<sup>2</sup> JavaScript standards aren't actually that standard.

# JavaScript is **NOT** Java <sup>1</sup>

- JavaScript was written was created in 10 days in May 1995 by Brendan Eich.
- JavaScript was originally called Mocha and was renamed to LiveScript before being renamed again to JavaScript.
- Why **JavaScript**? Because Java happened to be popular then (that was before people realized how much Java sucks in a browser) and JavaScript looks syntactically similar at a glance.
- JavaScript is standardized<sup>2</sup> by Ecma International and there have been a number of ECMAScript versions. The latest is ECMAScript 6, but it is not fully supported by any browsers, including Firefox which only has partial support.

---

<sup>1</sup> Lots of this slide's information is from: [https://www.w3.org/community/webed/wiki/A\\_Short\\_History\\_of\\_JavaScript](https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript)

<sup>2</sup> JavaScript standards aren't actually that standard.

# JavaScript is **NOT** Java <sup>1</sup>

- JavaScript was written was created in 10 days in May 1995 by Brendan Eich.
- JavaScript was originally called Mocha and was renamed to LiveScript before being renamed again to JavaScript.
- Why **Java**Script? Because Java happened to be popular then (that was before people realized how much Java sucks in a browser) and JavaScript looks syntactically similar at a glance.
- JavaScript is standardized<sup>2</sup> by Ecma International and there have been a number of ECMAScript versions. The latest is ECMAScript 6, but it is not fully supported by any browsers, including Firefox which only has partial support.

---

<sup>1</sup> Lots of this slide's information is from: [https://www.w3.org/community/webed/wiki/A\\_Short\\_History\\_of\\_JavaScript](https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript)

<sup>2</sup> JavaScript standards aren't actually that standard.

# JavaScript is **NOT** Java <sup>1</sup>

- JavaScript was written was created in 10 days in May 1995 by Brendan Eich.
- JavaScript was originally called Mocha and was renamed to LiveScript before being renamed again to JavaScript.
- Why **Java**Script? Because Java happened to be popular then (that was before people realized how much Java sucks in a browser) and JavaScript looks syntactically similar at a glance.
- JavaScript is standardized<sup>2</sup> by Ecma International and there have been a number of ECMAScript versions. The latest is ECMAScript 6, but it is not fully supported by any browsers, including Firefox which only has partial support.

---

<sup>1</sup> Lots of this slide's information is from: [https://www.w3.org/community/webed/wiki/A\\_Short\\_History\\_of\\_JavaScript](https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript)

<sup>2</sup> JavaScript standards aren't actually that standard.

# Objects & Primitives

- Everything is either a primitive or an object.
- Objects are ALWAYS passed by reference
- Primitives are ALWAYS passed by value
- Objects in JavaScript are mutable keyed collections/dictionaries.
- JavaScript is *pseudoclassical*.
- JavaScript uses *prototypes* for inheritance.
- There is no such thing as a *class* in JavaScript.<sup>1</sup>

---

<sup>1</sup>ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.

# Objects & Primitives

- Everything is either a primitive or an object.
- Objects are ALWAYS passed by reference
- Primitives are ALWAYS passed by value
- Objects in JavaScript are mutable keyed collections/dictionaries.
- JavaScript is *pseudoclassical*.
- JavaScript uses *prototypes* for inheritance.
- There is no such thing as a *class* in JavaScript.<sup>1</sup>

---

<sup>1</sup>ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.

# Objects & Primitives

- Everything is either a primitive or an object.
- Objects are ALWAYS passed by reference
- Primitives are ALWAYS passed by value
- Objects in JavaScript are mutable keyed collections/dictionaries.
- JavaScript is *pseudoclassical*.
- JavaScript uses *prototypes* for inheritance.
- There is no such thing as a *class* in JavaScript.<sup>1</sup>

---

<sup>1</sup>ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.



# Objects & Primitives

- Everything is either a primitive or an object.
- Objects are ALWAYS passed by reference
- Primitives are ALWAYS passed by value
- Objects in JavaScript are mutable keyed collections/dictionaries.
- JavaScript is *pseudoclassical*.
- JavaScript uses *prototypes* for inheritance.
- There is no such thing as a *class* in JavaScript.<sup>1</sup>

---

<sup>1</sup>ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.

# Objects & Primitives

- Everything is either a primitive or an object.
- Objects are ALWAYS passed by reference
- Primitives are ALWAYS passed by value
- Objects in JavaScript are mutable keyed collections/dictionaries.
- JavaScript is *pseudoclassical*.
- JavaScript uses *prototypes* for inheritance.
- There is no such thing as a *class* in JavaScript.<sup>1</sup>

---

<sup>1</sup>ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.

# Objects & Primitives

- Everything is either a primitive or an object.
- Objects are ALWAYS passed by reference
- Primitives are ALWAYS passed by value
- Objects in JavaScript are mutable keyed collections/dictionaries.
- JavaScript is *pseudoclassical*.
- JavaScript uses *prototypes* for inheritance.
- There is no such thing as a *class* in JavaScript.<sup>1</sup>

---

<sup>1</sup>ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.

# Objects & Primitives

- Everything is either a primitive or an object.
- Objects are ALWAYS passed by reference
- Primitives are ALWAYS passed by value
- Objects in JavaScript are mutable keyed collections/dictionaries.
- JavaScript is *pseudoclassical*.
- JavaScript uses *prototypes* for inheritance.
- There is no such thing as a *class* in JavaScript.<sup>1</sup>

---

<sup>1</sup>ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.

# Primitives: Types<sup>1</sup>

JavaScript has six primitive types:

- Boolean
- Null
- Undefined (yes, this is a type)
- Number (can be a number between  $-(2^{53} - 1)$  and  $2^{53} - 1$ , NaN, -Infinity, or Infinity).
- String (single or double quotes declares a string literal<sup>2</sup>)
- Symbol (new in ECMAScript 6)

---

<sup>1</sup>Info on this slide from: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data\\_structures](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)

<sup>2</sup>Single quotes are recommended by Douglas Crockford because HTML normally uses double quotes and to avoid conflicts when manipulating DOM objects, single quotes should be used.

# Objects: Inheritance and the Prototype Chain

- Every JavaScript object is linked to a *prototype*. If a member is not found in an object (i.e. if `obj.fooBar == undefined`) then the prototype is searched. It defines a sort of "default" set of values for the object.
- "Empty" objects start with `Object.prototype` defined as their prototype.
- You can set the prototype of an object to another object (or to undefined) by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is just another object, it too can have a prototype. Hence the prototype *chain*. When you access a property of an object, the whole prototype chain is searched for it.
- The prototype relationship is a dynamic relationship. If a property is added to the prototype, it is automatically visible to all objects based on that prototype.

# Objects: Inheritance and the Prototype Chain

- Every JavaScript object is linked to a *prototype*. If a member is not found in an object (i.e. if `obj.foobar == undefined`) then the prototype is searched. It defines a sort of "default" set of values for the object.
- "Empty" objects start with `Object.prototype` defined as their prototype.
- You can set the prototype of an object to another object (or to undefined) by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is just another object, it too can have a prototype. Hence the prototype *chain*. When you access a property of an object, the whole prototype chain is searched for it.
- The prototype relationship is a dynamic relationship. If a property is added to the prototype, it is automatically visible to all objects based on that prototype.

# Objects: Inheritance and the Prototype Chain

- Every JavaScript object is linked to a *prototype*. If a member is not found in an object (i.e. if `obj.foobar == undefined`) then the prototype is searched. It defines a sort of "default" set of values for the object.
- "Empty" objects start with `Object.prototype` defined as their prototype.
- You can set the prototype of an object to another object (or to undefined) by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is just another object, it too can have a prototype. Hence the *prototype chain*. When you access a property of an object, the whole prototype chain is searched for it.
- The prototype relationship is a dynamic relationship. If a property is added to the prototype, it is automatically visible to all objects based on that prototype.



# Objects: Inheritance and the Prototype Chain

- Every JavaScript object is linked to a *prototype*. If a member is not found in an object (i.e. if `obj.foobar == undefined`) then the prototype is searched. It defines a sort of "default" set of values for the object.
- "Empty" objects start with `Object.prototype` defined as their prototype.
- You can set the prototype of an object to another object (or to undefined) by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is just another object, it too can have a prototype. Hence the prototype *chain*. When you access a property of an object, the whole prototype chain is searched for it.
- The prototype relationship is a dynamic relationship. If a property is added to the prototype, it is automatically visible to all objects based on that prototype.

# Objects: Inheritance and the Prototype Chain

- Every JavaScript object is linked to a *prototype*. If a member is not found in an object (i.e. if `obj.foobar == undefined`) then the prototype is searched. It defines a sort of "default" set of values for the object.
- "Empty" objects start with `Object.prototype` defined as their prototype.
- You can set the prototype of an object to another object (or to undefined) by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is just another object, it too can have a prototype. Hence the prototype *chain*. When you access a property of an object, the whole prototype chain is searched for it.
- The prototype relationship is a dynamic relationship. If a property is added to the prototype, it is automatically visible to all objects based on that prototype.

# Objects: Syntax

---

```
1  var myObj = { // this is an object literal
2      a: 3,
3      'b': 'JavaScript',
4      'is-awesome?': true,
5      doSomething: function () {
6          console.log(this.a); // 3
7          console.log(a); // error
8      }, // trailing commas are allowed
9  };
10 myObj.doSomething();
11 console.log(myObj.b, myObj['is-awesome?']);
```

---

## Output:

---

```
1  3
2  error: a is undefined
3  JavaScript true
```

---

# Objects: Arrays

JavaScript arrays are basically vectors (and are also objects, remember?).

---

```
1 var arr = [1, 'a', {}, [], true];
2 arr[0] = 'not a number';
3 arr.push('this is basically a vector');
4 console.log(arr);
```

---

## Output:

---

```
1 [ 'not a number', 'a', {}, [], true, 'this is basically a vector' ]
```

---

*Note that the elements of an array do not have to be the same type.*

# Variables

JavaScript is an **untyped** language. I don't know what that means and I don't think that Brendan did either when he wrote the language.

Variables are declared using the `var` keyword<sup>1</sup>.

## Examples:

- `var name;` - creates variable `name` of type `undefined`.
- `var name = 'Sumner';` - string literal
- `var age = 18;` - declaring a number literal
- `var hasFriends = false;` - declaring a boolean
- `var significantOther = null;`

---

<sup>1</sup>Sometimes you don't need to use `var` as I have described above.

# Functions

- Functions are just objects with two special properties: a context (scope) and the function code.
- Functions can be defined anywhere where an object can be defined and can be stored in variables.
- Functions can access all arguments passed to a function via the `arguments` variable.
- Functions can access the callee of a function (`callee.func()`) via the `this` variable.
- Functions can also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return `undefined`.

# Functions

- Functions are just objects with two special properties: a context (scope) and the function code.
- Functions can be defined anywhere where an object can be defined and can be stored in variables.
- Functions can access all arguments passed to a function via the `arguments` variable.
- Functions can access the callee of a function (`callee.func()`) via the `this` variable.
- Functions can also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return `undefined`.

# Functions

- Functions are just objects with two special properties: a context (scope) and the function code.
- Functions can be defined anywhere where an object can be defined and can be stored in variables.
- Functions can access all arguments passed to a function via the `arguments` variable.
- Functions can access the callee of a function (`callee.func()`) via the `this` variable.
- Functions can also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return `undefined`.



# Functions

- Functions are just objects with two special properties: a context (scope) and the function code.
- Functions can be defined anywhere where an object can be defined and can be stored in variables.
- Functions can access all arguments passed to a function via the `arguments` variable.
- Functions can access the callee of a function (`callee.func()`) via the `this` variable.
- Functions can also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return `undefined`.

# Functions

- Functions are just objects with two special properties: a context (scope) and the function code.
- Functions can be defined anywhere where an object can be defined and can be stored in variables.
- Functions can access all arguments passed to a function via the `arguments` variable.
- Functions can access the callee of a function (`callee.func()`) via the `this` variable.
- Functions can also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return `undefined`.

# Functions

- Functions are just objects with two special properties: a context (scope) and the function code.
- Functions can be defined anywhere where an object can be defined and can be stored in variables.
- Functions can access all arguments passed to a function via the `arguments` variable.
- Functions can access the callee of a function (`callee.func()`) via the `this` variable.
- Functions can also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return `undefined`.

# Functions: Callback

Since JavaScript functions are objects, they can be passed just like other objects.

---

```
1  function doStuff(callback) {  
2      // do a bunch of processing  
3      var x = 3;  
4      console.log('in doStuff');  
5      callback(x);  
6  }  
7  
8  doStuff(function(x) {  
9      console.log(x * 3);  
10 })
```

---

## Output:

---

```
1  in doStuff  
2  9
```

---

# Functions: New

JavaScript functions can be invoked with the `new` keyword, mimicking traditional class-based languages:

---

```
1 function Thing(val) {  
2     this.v = val;  
3 }  
4  
5 var t = new Thing(12);  
6 console.log(t.v); // prints 12
```

---

But don't be fooled. Really that is just equivalent to:

---

```
1 ...  
2  
3 var t = {};  
4 t.prototype = Thing.prototype;  
5 t.Thing(12); // the important bit!  
6 console.log(t.v); // prints 12
```

---

# Scope

There are two scopes in JavaScript: global and function.<sup>1</sup>

Variables declared outside of a function are automatically in the global scope.

Variables declared within a function *without* the `var` keyword are also in the global scope.

---

```
1  var a = 2;
2  (function() {
3      b = 3
4      var c = 5;
5  })(); // this creates and invokes the function
6      // immediately
7
8  console.log(a); // logs 2
9  console.log(b); // logs 3
10 console.log(c); // error since c is undefined
11               // in global scope
```

# Global Abatement

Because your code could coexist with other people's code, on the same HTML page, it is recommended that you reduce your *global footprint* by only creating a few global objects and then putting all assets into that object.

---

```
1 myGlobal = (function() {
2     var myInternalData = 10;
3     return {
4         data: 5,
5         subObject: {
6             cool: 'things',
7         },
8         fn: function() { return myInternalData; },
9     };
10 })();
```

---

Since you can add properties to objects at will, you can still split your code into multiple files.

# Private Variables

You can simulate private variables the same way:

---

```
1  var Dog = function(name) {
2      var gender = 'male';
3      this.name = name;
4      this.isBoy = function () {
5          return gender == 'male';
6      };
7  };
8
9  var myDog = new Dog('Sebastian');
10 console.log(myDog.gender); // logs undefined
11 console.log(myDog.name);   // logs 'Sebastian'
12 console.log(myDog.isBoy()); // logs true
```

---



# Syntax: Control Statements

---

```
1  // if statement syntax is identical to C++
2  if (condition) {
3  } else if (condition) {
4  } else {
5  }
6
7  // ternary syntax is just like C++
8  var a = condition ? val_if_true : val_if_false;
9
10 for (initializer; condition; incrementor) {
11     // for loop syntax is identical
12 }
13
14 for (var prop in obj) {
15     obj[prop].doThing(); // prop is the key
16                         // could be a number or a string
17 }
```

---

# Pitfalls: Variable Hoisting

Variables are *hoisted* to the top of the function they are declared in. Thus, the following is entirely valid.

---

```
1 function scopeEx() {  
2     b = 5;  
3     console.log(b); // logs 5  
4     var b = 3  
5     console.log(b); // logs 3  
6 }
```

---

This is confusing. Just declare your variables before you use them.

---

<sup>1</sup>In ES6, variables declared with `let` are actually block scope.

# Pitfalls: Truthy, Falsy and `==` vs `===`

JavaScript has the notion of being *truthy* and *falsy*.

The following values are always falsy: `false`, `0`, `""`, `null`, `undefined`, `NaN`.

Do not expect all falsy values to be equal to each other (`false == null` is false).

JavaScript has two equality operators:

- `==` compares without checking variable type. This will cast then compare.
- `===` compares and checks variable type.

# DOM Manipulation

The *Document Object Model* is an API used by JavaScript to interact with the elements of an HTML document.<sup>1</sup>

jQuery is great for simple DOM manipulation.

---

```
1 <div id="cool">Cool</div>
2 <div class="myCls">jQuery Demo</div>
```

---

---

```
1 var coolDiv = document.getElementById('cool'); // pure JS
2 coolDiv.style.background = 'blue';
3
4 var coolDiv = $('#cool'); // jQuery
5 coolDiv.css('background-color', 'blue');
```

---

jQuery does a ton of other useful things as well, but that's what the docs are for.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Document\\_Object\\_Model](https://en.wikipedia.org/wiki/Document_Object_Model)

# Canvas Manipulation

While many JS games (like 2048) use lots of HTML and CSS to draw the game, with some JS and DOM/JQuery-stuff for the logic. However, you can also draw the game directly using a Canvas. All you need then is a few lines of HTML and the rest can happen in your script. You can even create 3D stuff with WebGL or a 3rd party library like Three.js.

---

```
1 var c = document.getElementById("myCanvas");
2 var ctx = c.getContext("2d");
3 ctx.moveTo(0,0);
4 ctx.lineTo(200,100);
5 ctx.stroke();
```

---

# Libraries

- DOM Manipulation (HTML and CSS stuff)
  - ✓ JQuery (Yep)
- HTML5 Canvas (Direct drawing from JS)
  - ✓ EaselJS (Nice interaction callbacks)
  - bHive (Never used it, but other people like it)
  - ✓ Paper.js (Good vector and shape drawing)
  - WebGL (3D Graphics if you can OpenGL the things)
    - \* ✓ Three.js (3D Graphics if you can't Opengl the things)
    - \* BabylonJS (Looks pretty I guess)
- WebSockets (TCP, multiplayer, experimental, good luck)
  - Sockets.io (talk to your Node.JS server?)
- Audio Stuff
  - SoundJS (Again, never used)
  - Google (you are smart, figure it out)

## Additional Resources

A lot of this presentation was based off of *JavaScript: The Good Parts* by Douglas Crockford. This is an essential read for anyone interested in learning JavaScript for anything more than writing a few simple scripts.

MDN is the best resource for JavaScript documentation (<https://developer.mozilla.org/en-US/>).

**JSHint** (<http://jshint.com/about/>) is a tool which checks JavaScript syntax and helps prevent bugs in your code. JSHint has plugins for most IDEs and text editors. Here's a SO article on the Vim plugin: <http://stackoverflow.com/questions/473478/vim-jshint/5893447>