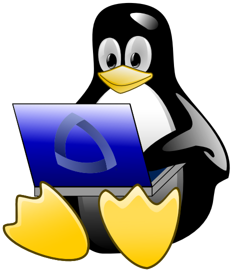
JavaScript Crash Course

Sumner Evans and Sam Sartor

November 10, 2016



Colorado School of Mines
Linux Users Group

JavaScript is **NOT** Java [1]

- 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[2] 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.

[1] Lots of this slide's information is from: `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`

[2] JavaScript standards aren't actually that standard.

JavaScript is **NOT** Java [1]

- 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[2] 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.

[1] Lots of this slide's information is from: `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`

[2] JavaScript standards aren't actually that standard.

JavaScript is **NOT** Java [1]

- 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[2] 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.

[1] Lots of this slide's information is from: `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`

[2] JavaScript standards aren't actually that standard.

JavaScript is **NOT** Java [1]

- 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[2] 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.

[1] Lots of this slide's information is from: `https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript`

[2] 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.[1]

[1] 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.[1]

[1] 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.[1]

[1] 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.[1]

[1] 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.[1]

[1] 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.[1]

[1] 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.[1]

[1] ECMAScript 6 added support for classes, but JavaScript classes are just wrappers around the underlying prototype-based structure.

Primitives: Types[1]

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[2])
- Symbol (new in ECMAScript 6)

[1] Info on this slide from: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures`

[2] 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

```
var myObj = { // this is an object literal
    a: 3,
    'b': 'JavaScript',
    'is-awesome?': true,
    doSomething: function () {
        console.log(this.a); // 3
        console.log(a); // error
    }, // trailing commas are allowed
};
myObj.doSomething();
console.log(myObj.b, myObj['is-awesome?']);
```

Output:

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

Objects: Arrays

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

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

Output:

```
[ '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[1].

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;`

[1] 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.

```
function doStuff(callback) {
    // do a bunch of processing
    var x = 3;
    console.log('in doStuff');
    callback(x);
}

doStuff(function(x) {
    console.log(x * 3);
});
```

Output:

```
in doStuff
9
```

Functions: New

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

```javascript
function Thing(val) {
    this.v = val;
}

var t = new Thing(12);
console.log(t.v); // prints 12
```

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

```javascript
...

var t = {};
t.prototype = Thing.prototype;
t.Thing(12); // the important bit!
console.log(t.v); // prints 12
```

Scope

There are two scopes in JavaScript: global and function.[1]

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.

```javascript
var a = 2;
(function() {
    b = 3
    var c = 5;
})(); // this creates and invokes the function
      // immediately

console.log(a); // logs 2
console.log(b); // logs 3
console.log(c); // error since c is undefined
                // 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.

```
myGlobal = (function() {
    var myInternalData = 10;
    return {
        data: 5,
        subObject: {
            cool: 'things',
        },
        fn: function() { return myInternalData; },
    };
})();
```

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:

```javascript
var Dog = function(name) {
    var gender = 'male';
    this.name = name;
    this.isBoy = function () {
        return gender == 'male';
    };
};

var myDog = new Dog('Sebastian');
console.log(myDog.gender);  // logs undefined
console.log(myDog.name);    // logs 'Sebastian'
console.log(myDog.isBoy()); // logs true
```

Syntax: Control Statements

```javascript
// if statement syntax is identical to C++
if (condition) {
} else if (condition) {
} else {
}

// ternary syntax is just like C++
var a = condition ? val_if_true : val_if_false;

for (initializer; condition; incrementor) {
    // for loop syntax is identical
}

for (var prop in obj) {
    obj[prop].doThing(); // prop is the key
                         // could be a number or a string
}
```

Pitfalls: Variable Hoisting

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

```javascript
function scopeEx() {
    b = 5;
    console.log(b); // logs 5
    var b = 3
    console.log(b); // logs 3
}
```

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

[1] 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.

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-jslint/5893447`

Additional Resources: Libraries

There are **lots** of JavaScript libraries. One of the most widely used is jQuery (`http://jquery.com/`). It has good documentation and is really good for DOM manipulation.

DOM Manipulation

The *Document Object Model* is an API used by JavaScript to interact with the elements of an HTML document.[1]

[1] `https://en.wikipedia.org/wiki/Document_Object_Model`

Canvas Manipulation

Sources

I relived heavily on *JavaScript the Good Parts* by Douglas Crockford in preparing this presentation. In fact, almost every slide contains some information I got from that book.