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 awful it is) 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 awful it is) 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 awful it is) 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 awful it is) 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

- Everything is either a primitive or an object.
- Objects in JavaScript are mutable keyed collections.
- 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

- Everything is either a primitive or an object.
- Objects in JavaScript are mutable keyed collections.
- 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

- Everything is either a primitive or an object.
- Objects in JavaScript are mutable keyed collections.
- 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

- Everything is either a primitive or an object.
- Objects in JavaScript are mutable keyed collections.
- 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

- Everything is either a primitive or an object.
- Objects in JavaScript are mutable keyed collections.
- 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: Inheritance and the Prototype Chain

- Every JavaScript object is liked to a *prototype*. Objects inherit the properties from their prototypes.
- Object literals inherit from `Object.prototype` which is defined by the JavaScript language.
- You can set the prototype of an object to another object by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is itself an object, the prototype will have a prototype.
- 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 liked to a *prototype*. Objects inherit the properties from their prototypes.
- Object literals inherit from `Object.prototype` which is defined by the JavaScript language.
- You can set the prototype of an object to another object by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is itself an object, the prototype will have a prototype.
- 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 liked to a *prototype*. Objects inherit the properties from their prototypes.
- Object literals inherit from `Object.prototype` which is defined by the JavaScript language.
- You can set the prototype of an object to another object by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is itself an object, the prototype will have a prototype.
- 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 liked to a *prototype*. Objects inherit the properties from their prototypes.
- Object literals inherit from `Object.prototype` which is defined by the JavaScript language.
- You can set the prototype of an object to another object by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is itself an object, the prototype will have a prototype.
- 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 liked to a *prototype*. Objects inherit the properties from their prototypes.
- Object literals inherit from `Object.prototype` which is defined by the JavaScript language.
- You can set the prototype of an object to another object by calling `myObj.prototype = otherObj;`
- Since the prototype of an object is itself an object, the prototype will have a prototype.
- 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.

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 also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return undefined.
- Functions invoked using the new command construct objects. The default return value is `this`.

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 also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return undefined.
- Functions invoked using the new command construct objects. The default return value is `this`.

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 also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return undefined.
- Functions invoked using the new command construct objects. The default return value is `this`.

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 also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return undefined.
- Functions invoked using the new command construct objects. The default return value is `this`.

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 also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return undefined.
- Functions invoked using the new command construct objects. The default return value is `this`.

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 also have named parameters.
- Functions always return a value. If no return is explicitly specified, the function will return undefined.
- Functions invoked using the new command construct objects. The default return value is `this`.

Functions: Callback

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

Example:

```
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
```

Scope I

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

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

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

console.log(b); // error since b is undefined
```

This is confusing. It is recommended that you declare all of your variables at the top of your functions (one exception to this rule is counter variables).

[1] In ES6, variables declared with `let` are actually block scope.

Scope II

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.

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

console.log(a); // logs 2
console.log(b); // logs 3
```

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 creating only a few global objects and then putting everything into that.

Functions: Closure

Arrays

JavaScript arrays are basically vectors.

Example:

```
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’;` - you can initialize a variable when you declare it.

[1] Sometimes you don't need to use `var`.

Literals

It is extremely easy to declare object and array literals.

Types[1]

JavaScript has six primitive types:

- Boolean (`true` or `false`)
- 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 is recommended by Douglas Crockford because HTML normally uses double quotes and to avoid conflicts when manipulating DOM objects, single quotes should be used.

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

JavaScript has the notion of being *truthy* and *falsy*.
JavaScript has two equality operators:

- == compares without checking variable type.
- === 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.