Chapter 1: JavaScript

Where did it come from? Created in one day by Brendan Eich for Netscape in 1995 and is inspired by Java, C, Self, Perl... It's Multi-paradigm

- Imperative
- Functional
- Object-oriented
- Prototype-driven
- Event-driven
- Embeddable

JavaScript is best known for it's weird type casting behavior, this leads to a lot of debugging issues.

The reason why we keep using it is because it runs everywhere:

- Browsers
- Servers
- PDFs

It's also fast on modern browsers, they can compile javascript to machine code.

Running JS on a webpage

You can run JavaScript within the `<script>` tags inside a html file.

```
<script>
var someJS = "JS in a <script> tag!";
</script>
<!-- you can embed oneliners within HTML! -->
<button onClick="alert('Stuck in JS factory,
↪  send help!');">
  Test me!</button>
```

You can also put JavaScript at a separate URL with the `<script>` tag as well.

```
<script src="myscript.js"></script>
<!--You have to put the closing </script> tag
because script isn't a void tag!-->
```

This allows the html content to be cached while the content of `myscript.js` can change without affecting the caching behavior of the source html file.

Functions

- Functions can return values
- Functions can have parameters
- Functions can access all available scope

```
// Function one with no parameters
function one() {
    return 1;
}
// A function with 1 param
function f(x) {
    return 2*x;
}
// How to call the function
f(2) == 4.0;
// Show on the console each click event
// Unnamed function
button.onclick = function(e) {
    console.log(e);
};
```

Comments

```
// This is a line comment
/*
This is a block comment
*/
```

Closure

A closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
// I'm some javascript
/* also a comment */
function outer(text) {
    function inner(x) {
        alert(text);
    }
    return inner;
}
f = outer("Hi mom I'm on the projector");
f(1);
```

The code segment before will give an alert to the user with `Hi mom I'm on the projector`

Another example:

```
function p(a) {
    function q() {
        console.log("a: " + a);
        console.log("arguments: " +
        ↪  arguments[0]);
    }
    return q;
}
r = p(1);
s = p(2); // now p is finished running, what
↪  happens to a?
r(3); // print 1
s(4); // print 2
```

Even though `r` will print 1 and `s` will print 2 the arguments that `r` and `s` will print is 3 and 4 respectively.

This is because `function p` will return `function q` to the caller that is `r` and `s`. When give parameters to `r` and `s` those arguments will go to `function q`. However, `function q` doesn't take in any parameters. What gives? In JavaScript giving additional parameters to a functions is stored in the `arguments` variable as a list. Where the first additional argument is stored in `arguments[0]`, the next additional argument is stored in `arguments[1]` and so on.

Scope

By default creating a variable makes globals. To create a local variable use

- `'use strict'`
- `let`
- `const` for constants
- `var`

- `let` and `const` scopes variables/constants to the enclosing block
- `var` scopes variables to the enclosing function

```
"use strict";
function f() {
    let a = 1;
    for (let i = 1; i<10; i++) {
        var c = i; // var defines it inside
        ↪  the function
        let b = i; // let only defines it
        ↪  inside the for loop
    }
    console.log(c); // 9
    console.log(b); // Reference Error
}
f();
console.log(a); // Reference Error
```

Note

`let` should be used over `var` because regardless of where a `var` variable is created it's is hoisted to the top of it's scope. While `let` behaves similar to a regularly defined variable.

Type Coercion

Python does a lot of implicit type conversion

```
"1" + 1 //-> "11"
```

There is a bias towards strings and floats for conversions.

this

Like `self` in Python, except: You don't have to list in the arguments. It's not automatically bound to an object.

In most cases, the value of `this` is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called.

In non-strict mode the value of `this` is always a reference to an object. In `'use strict'` mode it can be any value.

```
class Quiz {
  write() {
    console.log(this);
  }
}
quiz = new Quiz();
w = quiz.write
w()
```

In JavaScript the w method isn't bound to the quiz object unless we call it explicitly like `quiz.write`, `w.call(quiz)` or bind it with `w.bind(quiz)`

ES5 introduced the `bind()` method to set the value of a function's this regardless of how it's called, and ES2015 introduced arrow functions which don't provide their own this binding (it retains the this value of the enclosing lexical context).

Anonymous Functions

JavaScript allows anonymous functions by returning a function definition.

```
var outer = function(text) {
    return function(x) {
        console.log(this);
        alert(text);
    }
}
var f = outer("Hi mom I'm on the projector");
f.call({}, 1);
```

Arrow Function =>

Like anonymous functions but they always keep the `this` from when they were created.

```
var outer = function(text) {
    return (x) => {
        console.log(this);
        alert(text);
    }
}
var f = outer("Hi mom I'm on the projector");
f.call({}, 1); // {} is ignored because it's
↪  an arrow function
```

Names

- Start with a letter followed by underscores, letters or numbers.
- Can't be a reserved word like `break` or `case` or `for` or `function` or `if` or `in` etc.
- Convention is to use `camelCase` like Java not `snake_case` like Python.

```
var aString = "Strings";
var break = "not allowed!";
var BREAK = "This is allowed!";
var BrEAK = "Try not to abuse case
↪  sensitivity";
```

Numbers

Everything is a double, write integers, decimals, or decimals with an exponent.

```
var aNumber = 10;
var aNumber = 11.11;
var aNumber = 1e-100;
var aNumber = 1E+100;
var nan = NaN;
var inf = Infinity;
var negativeInfinity = -Infinity;
```

Casting Numbers

There are many different ways to cast numbers

Casting to an "integer"

```
Math.floor(0.7)
Math.ceil(0.7)
Math.round(0.7)
Math.trunc(0.7)
```

Casting to a float

```
parseFloat("127")
Number("0x7F")
+"0x7F" // Unary plus is the same as Number
```

JavaScript also accepts string hexadecimal for conversions to numbers

Rounding Errors

Since everything's a double, you get rounding errors.

```
a = 0.1
b = 0.2
a + b == 0.3 // Should be true but false due
↪  to a rounding error
```

Strings

- Unicode by default
- Use ' ' or " "
- Any characters except control characters and " or '

```
var aNumber = 10;
var aString = "";
var anotherString ="Hi how are you";
var escapesString = "\r\n\t\f\b\/\\\\'\"";
var snowMan = "\u2603";
snowMan.length === 1;
aString.length === 0;
```

Casting Strings

There are a ways to convert things to a string

```
"" + 1;
1 + "";
String(1);
(1).toString();
String(null); // "null"
null.toString(); // Error
(127).toString(16); // "7F"
```

Booleans

- `true` or `false`
- Unfortunately conditional expressions have lot of truthy and falsey values
- False values:

```
false
null
undefined
''
0
NaN
```

- Everything else is true plus `true` itself.

Equality

- `==` the *abstract* equality operator
- `===` the *strict* equality operator i.e., type must match.

```
3=="3" // true
3==="3" // false
1==true // true
1===true // false
undefined == null // true
undefined === null // false
NaN==NaN // false
NaN===NaN // false
isNaN(NaN) // true
```

Generally you should use `===`.

Arrays

Arrays in JavaScript are object-oriented and fill of methods.

Unlike MATLAB arrays are 0-indexed.

```
var empty = [];
var arrayInitialized = [1,2,3,4,'5'];//mixed!
var arr = new Array(10);
arr[0] === undefined;
arr[0] = 'Assigned';
'Assigned' === arr[0];
arrayInitialized[4] === '5';
arrayInitialized.length === 5;
arrayInitialized.splice(3,1); // delete 4
↪  from the array (slow)
```

Iterating over an array

You can use `for ... of` to iterate over iterable objects, including; strings, arrays, and array-like objects.

```
let a = [1, 2, 3, 4, 5];
for (let i of a) {
  console.log(i);
}
```

If you use `for ... in` on an iterable object, like strings, arrays, and array-like objects then you will get the *indices* of those iterated elements rather than the elements themselves.

Objects

Everything is an object except for these primitives

- Booleans
- numbers
- strings

Although those primitives still have methods, **objects** have properties. These properties are named by a string and property values can be anything including `undefined`. Objects don't have a class and objects are pass by reference.

Example

```
var empty = {};
var abram = {
   "name":"Abram Hindle",
   "job":"Throwing Down JS",
   "favorite tea":"puerh"
};
var dog = {
   paws: 4 // note I didn't quote paws
};
dog.paws === 4;
abram["favorite tea"] === "puerh";
abram.name === "Abram Hindle";
abram["favorite tea"] = "oolong";
```

More examples

```
undefined.property; // Throws a type error
undefined && undefined.property // returns
↪  undefined
var empty = {};
empty.property === undefined;
var abram = {
   "name":"Abram Hindle",
   "job":"Throwing Down JS",
   "favorite tea":"puerh"
};
keys(abram); // produces
↪  ["name","job","favorite tea"]
//prototype!
var abramChild = Object.create(abram)
keys(abramChild); // produces []
abramChild.name === "Abram Hindle"; //
↪  inherits keys from abram
```

Prototypes

All javascript objects inherit properties and methods from a prototypes. Prototype provides inheritance in JavaScript where objects can have a prototype object, which acts as a template object that it inherits methods and properties from.

```
var abram = {
   "name":"Abram Hindle",
   "job":"Throwing Down JS",
   "favorite tea":"puerh",
   "sayName": function() {
      alert(this.name);
   }
};
abramChild = Object.create(abram);
abramChild.name = "Child";
function doit() {
    abram.sayName();
    abramChild.sayName();
}
```

In this case `abramChild` is a object created via the `abram` prototype. It inherited all the properties from abram as well as their default values, but can change the value of the properties after initialization. In this case change `abramChild.name` to `Child` instead of `Abram Hindle`.

Looping over Properties

The `for ... in` statement iterates over all enumerable properties of an object that are keyed by strings, including inherited enumerable properties from the prototype.

```
let author = {
   "name":"Unknown Slide Author",
   "job":"Making Slides",
   "sayName": function() {
      alert(this.name);
   }
};
let hazel = Object.create(author);
hazel.name = "Hazel Campbell";
for (let property in hazel) {
  alert(property + ": " + hazel[property]);
}
```

`Object.keys(object)`

If you want the properties or keys of the object and **exclude their inherited properties** then use the `Object.keys(object)` method to iterate over the object's own properties and not their inherited properties.

Classes

Classes are in essence, "special functions", and and just as you can define function expressions and

function declarations, the class syntax has two components: class expression and class declarations.

JavaScript has several ways of creating `classes`,

- ECMAScript 2015 classes
- Constructor functions

ECMAScript 2015 Classes

The proper way of creating classes is using the ECMAScript 2015 classes.

```
class Pokemon {
  constructor(name, level) {
    this.name = name;
    this.level = level;
  }
  levelUp() {
    this.level += 1;
  }
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

This is similar to C++ where you have a constructor method within the class definition that initializes a class instance. Class methods are defined in a similar way to the constructor.

Constructor Function

You might see this in older code

```
// classes start with a capital letter by
↪  convention
function Pokemon(name,level) {
  this.name = name;
  this.level = level;
  this.levelUp = function() {
    this.level += 1;
  }
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

Where instead of a `class` keyword and definition you have a function that will define and initialize its data members and define the class methods within the body of the function constructor.

```
// classes start with a capital letter by
↪  convention
function Pokemon(name,level) {
  this.name = name;
  this.level = level;
  this.levelUp = function() {
    this.level += 1;
  }
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

Both instances uses new to create a new class instance.

Adding a method to a prototype

If you want to add a new or overwrite a method to an already defined class you can assign the new function on the class's prototype.

```
// classes start with a capital letter by
↪  convention
function Pokemon(name,level) {
  this.name = name;
  this.level = level;
}
// have to include "prototype" here so "this"
↪  works in the method
Pokemon.prototype.levelUp = function() {
  this.level += 1;
}
pikachu = new Pokemon("Pikachu", 1);
pikachu.levelUp();
```

You can also use the `Object.Create(prototype)` convention to create a new object from a "default" object. Though this is awkward and rarely used.

```
// classes start with a capital letter by
↪  convention
var Pokemon = {
  name: null,
  level: null,
  levelUp: function() {
    this.level += 1;
  }
}
pikachu = Object.create(Pokemon);
pikachu.name = "Pikachu";
pikachu.level = 1;
pikachu.levelUp();
```

DOM Manipulation

You can use JavaScript to manipulate what's on the page. The browser turns HTML into the DOM or Document Object Model.

- Document: the stuff on your page, the content
- Object: gets turned into objects accessible by JS
- Model: it's a tree with children nodes

DOM elements from HTML

```
<p>A paragraph</p>
<div>
  Hi!
  <a href="https://google.ca">Click me!</a>
</div>
```

The DOM elements for the html document is

- document (it's a tree with children nodes!)
- Root Element: HTML (document.children[0])
- Element: Head (document.children[0].children[0])
- Element: Body (document.children[0].children[1])
- Element: p (document.children[0].children[1].children[0])

- #text: A paragrah
- Element: div
- Text: Hi!
- Element: a ; attribute href
- Text: Click me!

Recursive

Because the DOM is a tree you can recursively traverse the in whatever ways you see fit.

```
function domRecurse(start, depth) {
  var out = "";
  for (child of start.childNodes) {
    indent = "    ".repeat(depth);
    out += indent + "* " + child.nodeName;
      if (child.nodeName === "#text") {
        out += " " +
        ↪  JSON.stringify(child.wholeText);
      }
    out += "\n";
    out += domRecurse(child, depth+1);
  }
  return out;
}
```

querySelector

You can also jump straight to an element using `querySelector`. It uses CSS-style selectors for the query. You don't need jQuery.

```
function domRecurseExample() {
  var start =
  ↪  document.querySelector("#domexample");
  alert(domRecurse(start, 0));
}
```

You can use other selectors than `querySelector` to find specific element(s).

```
// Get all DIVs
var divs =
↪  document.getElementsByTagName("div");
// gets all elements with class divider
var dividers =
↪  document.getElementsByClassName("divider");
// get the element with the ID main
var main = document.getElementById('main');
// get the element by name
var ups = document.getElementsByName('up');
```

jQuery

jQuery's functionality is now available from APIs built-in to browsers.

Old projects that need backwards-compatibility or already use jQuery:

- Using jQuery is totally cool!

New projects using ECMA 2016 and later:

- Better to just use the tools the browser gives you.

Changing the DOM

You can change the DOM by creating elements on the `document`

Filling a div

This will add

> Here's some text in that div! Let's make a link too!

The `Let's make a link too!` is an `a.href` to `https://google.ca/`.

```
<div id="fillme"></div>

function fillExample() {
  fillme = document.getElementById("fillme");
  fillme.textContent = "Here's some text in
  ↪  that div! ";
  a = document.createElement("a"); // create
  ↪  an a element
  a.textContent = "Let's make a link too!";
  a.href = "https://google.ca/";
  fillme.appendChild(a);
}
```

Adding a colour border

This will add a randomly colored border to each of the two `div` elements.

```
<div class="styleme">I'm text in a div!</div>
<div class="styleme">I'm text in another
↪  div!</div>

function styleExample() {
  var divs =
  ↪  document.getElementsByClassName("styleme");
  divs = Array.prototype.slice.call(divs); //
  ↪  convert HTMLCollection to Array
  divs.map( (div) => {
    console.log(div);
    div.style.border = "5px solid";
    div.style.borderColor = "rgb(" +
    ↪  (256*Math.random())
      + ", " + (256*Math.random())
      + ", " + (256*Math.random()) + ")";
  });
}
```

This will apply a random border color to both divs.

```
function styleExample2() {
  old =
  ↪  document.getElementById("styleme2sheet");
  if (old) {
    old.parentNode.removeChild(old);
  }
  var css = "div.styleme2 { border: 5px solid
  ↪  "
      + "rgb(" + (256*Math.random())
      + ", " + (256*Math.random())
```

```
        + ", " + (256*Math.random()) + "); }";
    style = document.createElement('style');
    style.type = "text/css";
    style.id = "styleme2sheet";
    style.textContent = css;
    document.head.appendChild(style);
}
```

Review

Running JS in HTML.

- Use the `<script>` tag to either inline JS code or even better
- Set the `src` property to the JavaScript file so you don't invalidate the page's cache.

Function.

- Functions can return values.
- Functions can have parameters.
- If parameters are not defined you can still pass arguments to the function by passing them regardless and use the `arguments` array in the function body to access them.
- Function can access all available scope.
- Everything is passed by reference.

Closure.

- Gives you access to an other function's scope from an inner function.
- Closures are created every time a function is created.
- You can call a function that returns another function (let's call it function 2) and calling that returned function (function 2) again will allow that second function (function 2) to access data from the first function. called.

Scope.

- Default creating a variable makes it global.
- For local variable, use `var`, `const` for constants, `let`, or `'use strict'`.
- `let` and `const` scopes variables to the enclosing block.
- `var` scopes variable to the enclosing function.
- `var` also declares and set variable to undefined at the start of the enclosing function.
- Use `let` over `var`

Type Coercion.

- For numbers there is a bias towards floats.
- For everything else there is a bias towards strings, and strings takes precedence.

This.

- Like Python's `self`
- Don't need to list it in the argument so no `def func(self, ...)`.
- Not automatically bound to an object, you have to explicitly define this to a variable or
- call the function using the `call()` or `bind()` method.

Anonymous Functions.

- Defined by just writing `function()` with no identifier.
- The `=>` operator also creates an anonymous function but defines `this` from when they were created.

Names.

- Start with a letter followed by `[a-zA-Z0-9_]`.
- Cannot be a reserved word like `break`, etc
- Use camelCase.

Numbers.

- Everything is a double
- Casting to int can be done using the `Math` module.
- Casting to float can be done using `parseFloat` or `Number` methods and it can take in both integers as well as hexadecimals.
- Numerical errors still apply like in C and C++.

Strings.

- Unicode by default
- You can use the `String` constructor or `(2).toString()` method on the primitive number type or just implicit casting.

Booleans.

- `true` or `false`
- There are falsey types which var keywords and values that default to false. For example an empty string is false.

Equality.

- == abstract equality operator, will try and implicit type cast before evaluating.
- === strict equality operator, will evaluate if the types match as well.
- Use the strict equality operator over the abstract equality operator.
- Also `NaN != NaN` for both operators, use `isNaN(NaN)`.

Arrays.

- Arrays are object oriented and have methods like
- length, splice
- index starts at 0.
- index operator is `[]`
- Range based iteration of objects is done using `for (... of ...)`
- Range based iteration of the indices is done using `for (... in ...)`

Objects.

- Everything is an object except for Numbers, Booleans and strings
- They are like JSON format with key-value pairs.
- The keys can be either strings or regular identifier with a colon `:`
- Passed by reference.
- Object's attributes can be accessed using the object indexing operator `object["string"]` or as a direct attribute so long as it doesn't have any spaces `object.name`.

- `Object.keys(object)` will produce the keys of the object.
- Iterating though the object using `for (... in object)` will iterate through all the keys including its prototype.

Classes.

- Constructor which is the newer standard way
- Constructor functions which is the older way where everything is defined in the function's body.
- You can add a method to a prototype by creating a new attribute to the `object.prototype` attribute.

DOM manipulation.

- Change the content of a page using JS
- You can get elements by using `document.querySelector()` or get specific elements using `document.getElementBy<selector>()` where the `<selector>` is what you want.
- You can create an element using `document.createElement()` and assign predefined attributes to it.