ECMAScript 2015

Were you expecting something else?

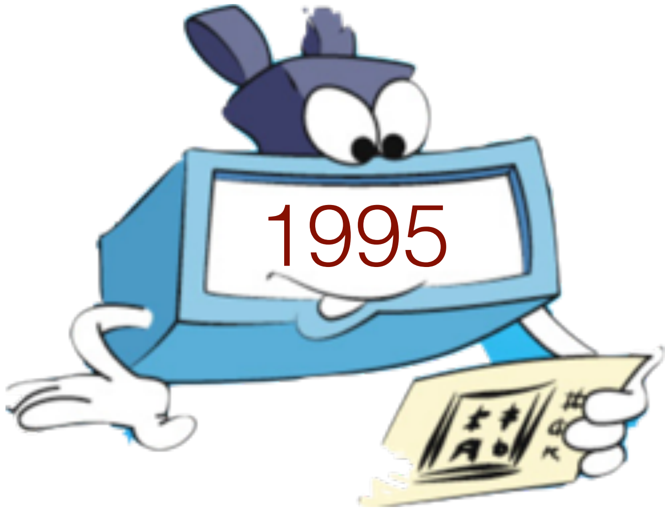
A History Lesson

A History Lesson

May - Mocha is invented in Netscape by Brendan Eich.

September - Renamed to LiveScript

December - Renamed to Javascript (Because Java was popular)



1995

A History Lesson

1996 - JavaScript is taken to standardization in ECMA.

From now on ECMA is the spec. Javascript is an implementation (ActionScript is another implementation)

1997 - ECMA-262 (ECMAScript)

1998 - ECMAScript 2

1999 - ECMAScript 3

A History Lesson

2005 - Mozilla and Macromedia started Work on ECMAScript 4. Feature rich and a very major leap from ECMAScript 3.

Douglas Crockford (Yahoo) And Microsoft opposed the forming standard.

ECMAScript 3.1 was the compromise.

A History Lesson

2009 - Opposing parties meet in Oslo and achieve an agreement.

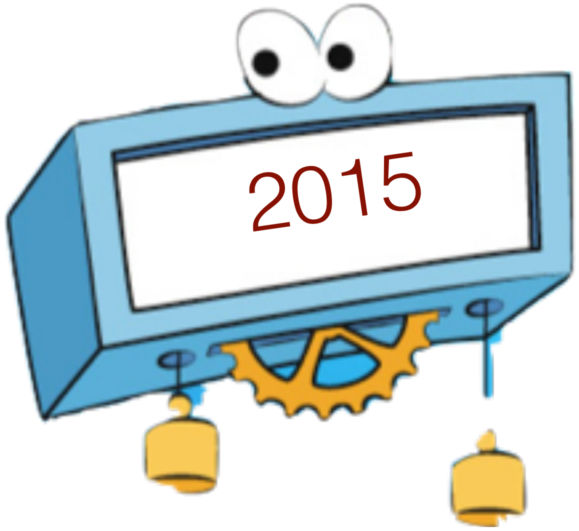
ES3.1 is renamed to ES5 (Which we have today)

In the spirit of peace and agreement, the new Javascript long term agenda is named “Harmony”

A History Lesson

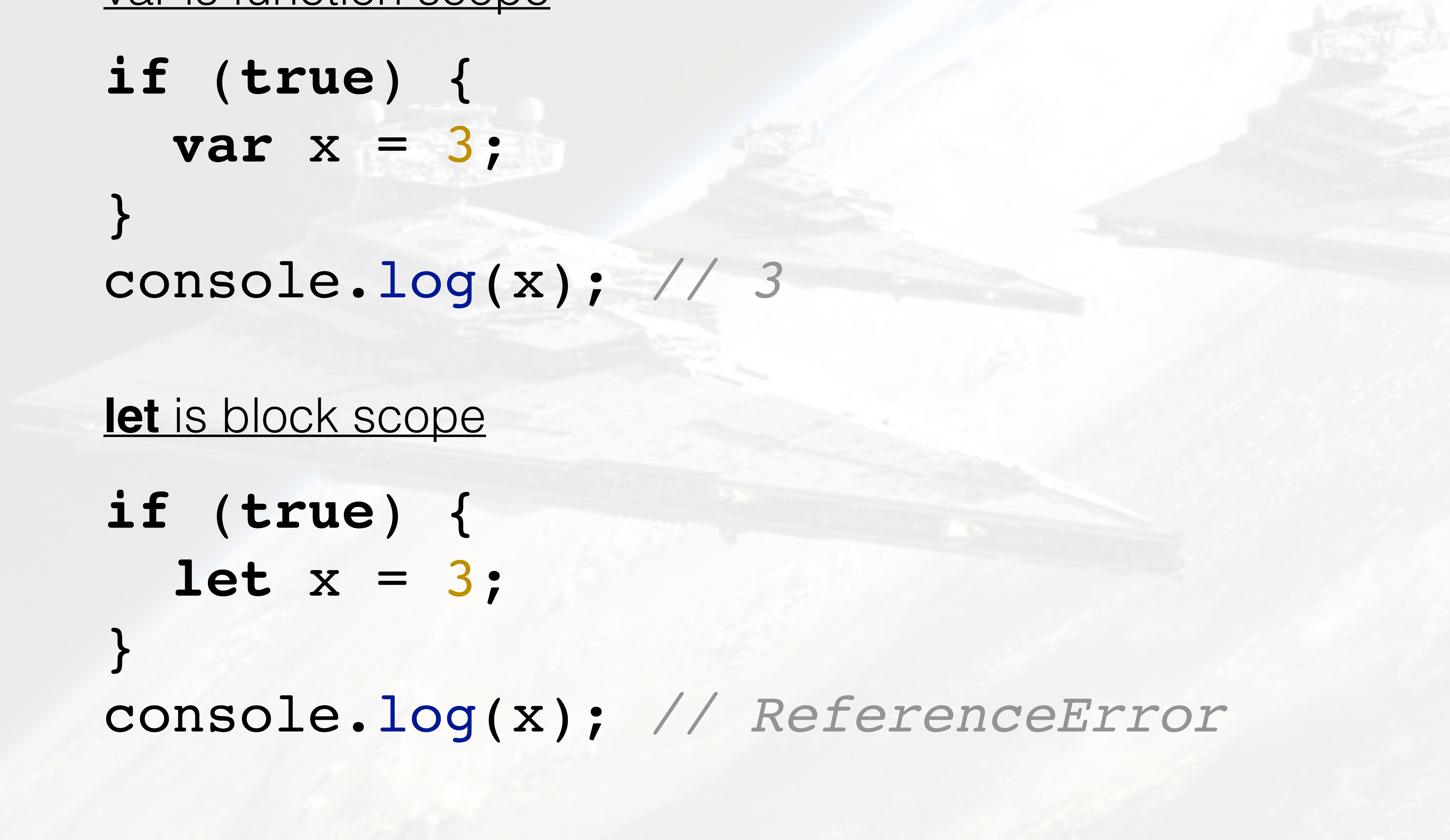
2015 - ES6 (Part of the “Harmony” umbrella) will be released.

Starting with ES6 - Version names will be based on the year of release. so ES6 is ES2015 and ES7 should be ES2016



Declaration and Scoping

let - A New ~~Hope~~ Scope



var is function scope

```
if (true) {
  var x = 3;
}
console.log(x); // 3
```

let is block scope

```
if (true) {
  let x = 3;
}
console.log(x); // ReferenceError
```

Loop scoping

With *let* You get a fresh iteration per loop

```
let vals = [];
for (let x = 0; x < 4; x += 1) {
  vals.push(() => x);
}
console.log(vals.map(x => x()));
// [0, 1, 2, 3]
```

With *var* You would get *[4, 4, 4, 4]* as a result

- we’re using functions to store by reference otherwise it won’t prove the point. Full Example here

const

const makes variables immutable

```
const obj = { par: 3 };
obj = 4; // TypeError
```

but it doesn't stop you from changing the object values

```
obj.par = 12; // Fine
```

you can use Object.freeze() to achieve that.

```
Object.freeze(obj);
obj.par = 10; // TypeError
```

or the Object.seal() which blocks changing object structure

```
console.seal(obj); // 17
obj.newParam = 3 // TypeError
```

String Templates

String Templates

This is syntactic sugar for string concatenation.

```
let name = 'John';
`Hi ${name},
 Did you know that 5 * 3 = ${5*3}?`

/*
"Hi John,
 Did you know that 5 * 3 = 15?"
 */
```

Anything in *${}* is evaluated
Back-ticks are used to enclose the string.
New lines are retained

Classes

Class

this is a class

```
class Jedi {
  constructor() {
    this.forceIsDark = false;
  }
  toString() {
    return (this.forceIsDark ? 'Join' :
      'Fear is the path to' ) +
     ' the dark side';
  }
}
```

you cannot define a property inside a class, but getters/ setters can create one to the outside)

Extends

extends works as you’d expect

```
class Sith extends Jedi {
  constructor() {
    super();
    this.forceIsDark = true;
  }
}
```

super() in a ctor calls the parent ctor

Class

```
let yoda = new Jedi();
let darth = new Sith();

console.log(yoda.toString());
// Fear is the path to the dark side
console.log(darth.toString());
// Join the dark side

console.log(darth instanceof Sith); //
true
console.log(darth instanceof Jedi); //
true
```

Static

you can declare static functions

```
class Util {
  static inc(x) { return x + 1 },
  static dec(x) { return x - 1 }
}

console.log(Util.inc(3)); // 4
```

it’s the same as placing them on the prototype object

Class Get/Set

you can define get/set just like in ES5 object literals

```
class O {
  constructor() {
    this.mx = 'initial';
  }
  get x() {
    return  this.mx;
  }
  set x(val) {
    console.log('x changed');
    this.mx = val;
  }
}
```

Class Get/Set

and to the user of the class it would appear like a property

```
let o = new O();
console.log(o.x); //initial

o.x = 'newval';
// x changed
console.log(o.x); //newval
```

Libraries

String

String got some new functions

```
"Hello".startsWith('Hell');
"Goodbye".endsWith('bye');
"Jar".repeat(2); // JarJar
"abcdef".includes("bcd");
```

Firefox still uses *contains* instead of *includes*. That should change

Number

New Number constants and methods

```
Number.EPSILON
Number.MAX_SAFE_INTEGER
Number.MIN_SAFE_INTEGER

Number.isNaN()
Number.isFinite()
Number.isInteger()
Number.isSafeInteger()
Number.parseFloat()
Number.parseInt()
```

Array.from()

from array-like objects (objects that have indices and length)

```
let arrayLike = {
  0: 'zero',
  1: 'one',
  2: 'two',
  3: 'three',
  'length' : 4
};

Array.from(arrayLike);
// Array ["zero", "one", "two", "three"]
```

Reminder… *

genTen is a custom iterable that generates 0 to 10

```
function

      [Symbol.iterator]() {



          done: (i > n) ?
          value: i++ };
        }
      };
    }
  };
}
```

Array.from()

from iterables

```
Array.from(gen(6));
// Array [ 0, 1, 2, 3, 4, 5, 6 ]
```

second parameter is a mapping function

```
Array.from(gen(6), x => x*x);
//Array [ 0, 1, 4, 9, 16, 25, 36 ]
```

Array.of()

a better way to create arrays

```
Array.of(1, 2, 4, 3, 4);
//Array [ 1, 2, 4, 3, 4 ]
```

you should use Array.of over Array constructor because:

```
new Array(2.3);
//RangeError: invalid array length
```

Array.prototype.*

Array.prototype.keys()

```
['a','b','c'].keys()
// Array Iterator {  }
[...['a','b','c'].keys()]
// Array [ 0, 1, 2 ]
```

Array.prototype.entries()

```
Array.from(['a','b','c'].entries())
// [[0,"a"],[1,"b"],[2,"c"]]
```

Notice how keys() and entries() return an iterator and not an array

Array.prototype.*

Array.prototype.find()

```
[4, 100, 7].find(x => x > 5);
// 100
```

Array.prototype.findIndex()

```
[4, 100, 7].findIndex(x => x > 5);
// 1
```

Array.prototype.fill()

```
(new Array(7)).fill(2).fill(3, 2, 5);
// Array [ 2, 2, 3, 3, 3, 2, 2 ]
```

Object

Object.assign()

```
let x = {a:1};
Object.assign(x, {b:2});
x; // {a:1, b:2}
```

object.is() checks if two values are the same

```
Object.is('y', 'y'); // true
Object.is({x:1}, {x:1}); // false
Object.is(NaN, NaN); // true
```

different than === in (+0 !== -0) and (NaN === NaN)

also - doesn’t coerce values (as == does)

Arrow Functions

The Basics

```
function inc(x) {
  return x + 1;
}
```

is equivalent to:

```
let inc = x => x + 1;
```

2 parameters:

```
let inc = (x, y) => x + y;
```

no parameters

```
let inc = () => 7;
```

The Basics

```
function inc(x) {
  return x + 1;
}
```

more than 1 statement

```
let inc = (x) => {
  console.log(x);
  return 7;
}
```

Lexical this

Arrow functions capture the this value of the enclosing context. In other words, no more `that`, `self` or `bind(this)`

```
this.x = 7;
setTimeout(() =>
  console.log(this.x),
  2000);

// wait for it....


// 7
```

Default Values

default value is used if match on src is undefined (either missing or actual value)

```
let [a,b = 3, c = 7] = [1, undefined];
// a === 1, b === 3, c === 7
```

works on both arrays and objects

```
let {x:x = 1 ,y:y = 2,z:z = 3} =
    { x: undefined, z: 7};
// x === 1, y === 2, z === 7
```

Default Values

default values are lazily evaluated

```
function con() {
  console.log('TEST');
  return 10;
}

let [x = con()] = [];
// TEST
// x === 10

let [x = con()] = [5];
// x === 5
```

Default Values

default values evaluation is equivalent to a list of let declarations

this fails

```
let [x = y, y = 0] = [];
// ReferenceError
```

this works

```
let [x = 7, y = x] = [];
// x === 7, y === 7
```

Modules

Modules

Modules allow you to load code on demand (async) and to provide a level of abstraction

```
//*****  Shape.js  ******
export default class Shape {
  getColor() { /*...*/ }
}

//*****  main.js  ******
import Shape from 'Shape';
let shape = new Shape();
```

this method exports a single default value for the module

Modules

you can also export by name

```
//*****  Circle.js ******
export function diameter(r) {
  return 2 * Math.PI * r;
}
export let area = r =>
  Math.pow(Math.PI*r, 2);
```

Modules

and then import by name

```
//*****  main.js  ******
import {area as circleArea, diameter}
  from 'Circle';
import area as cubeArea from 'Cube';
import * as tri from 'Triangle';

circleArea(5);
cubeArea(4);
tri.area(3);
```

Modules

- modules structure is statically defined.
- dependencies can be resolved statically.
- you can still load modules programatically if you choose

```
System.import('module')
.then(module => {})
.catch(error => {});
```

the end

whaddya wanna know?