

COMP3123 - Week 1

ES6

ES6

JS

Topics

- **Evolution of JavaScript**
- **Main goals of JavaScript & ES6**
- **ES6 Features**

Definitions



<https://www.ecma-international.org/>

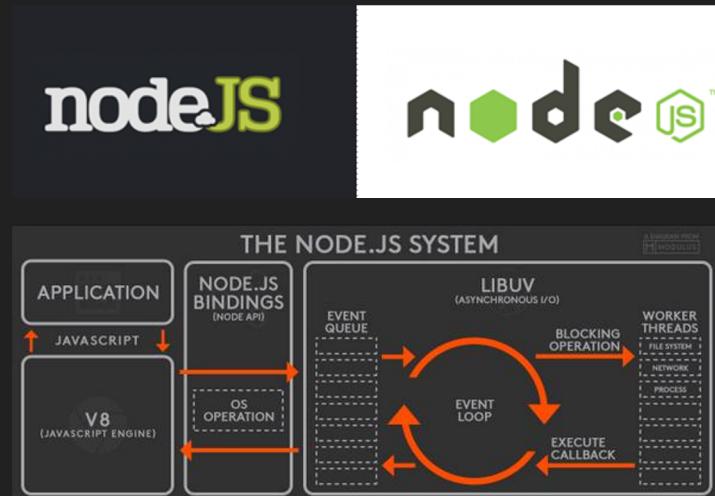
- **JavaScript (JS)** - a high level, dynamic, untyped and interpreted programming language created original for web browsers
- **ECMA International** - an international non-profit standards organization (*European Computer Manufacturers Association*)
- **ECMAScript (ES)** - scripting-language specification standardized by ECMA International. (*Implemented well known languages such as JavaScript, JScript and ActionScript*)
- **ES2015 (ES6)** - the newest version of ECMAScript

1990's



- 1995: Netscape creates Mocha
 - 1995: Mocha > LiveScript > JavaScript
 - 1996: ECMA adopts JavaScript
 - 1997: ECMA-262 (ES1)
 - 1998: ES2
 - 1999: ES3 (regex, try/catch)
- * 1990's browser wars IE vs Netscape

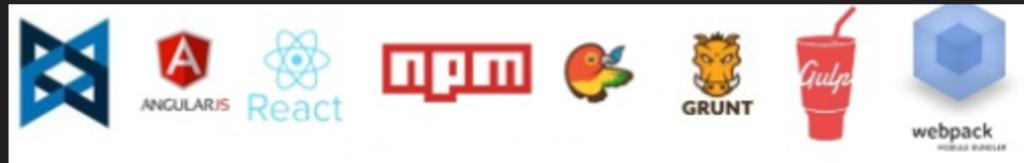
2006-2009



- 2008: ECMAScript 4 (*abandoned*)
- 2009: ECMAScript 3.1 5 (strict, JSON, Reflect)
- 2009: Emergence of Server Side JavaScript environment => Node.js

2010-2015

- frameworks continue to evolve, no longer just DOM & AJAX helpers
- JS Packet Manager ie. npm, bower
- solutions for keeping code in modules (node.js, CommonJs, AMD, Browserify)
- JavaScript preprocessors (Grunt, Gulp, Webpack..)



JavaScript Pros/Cons

Pros

- easy syntax
- functions are objects
- the only native web browser language
- independently driven

Cons

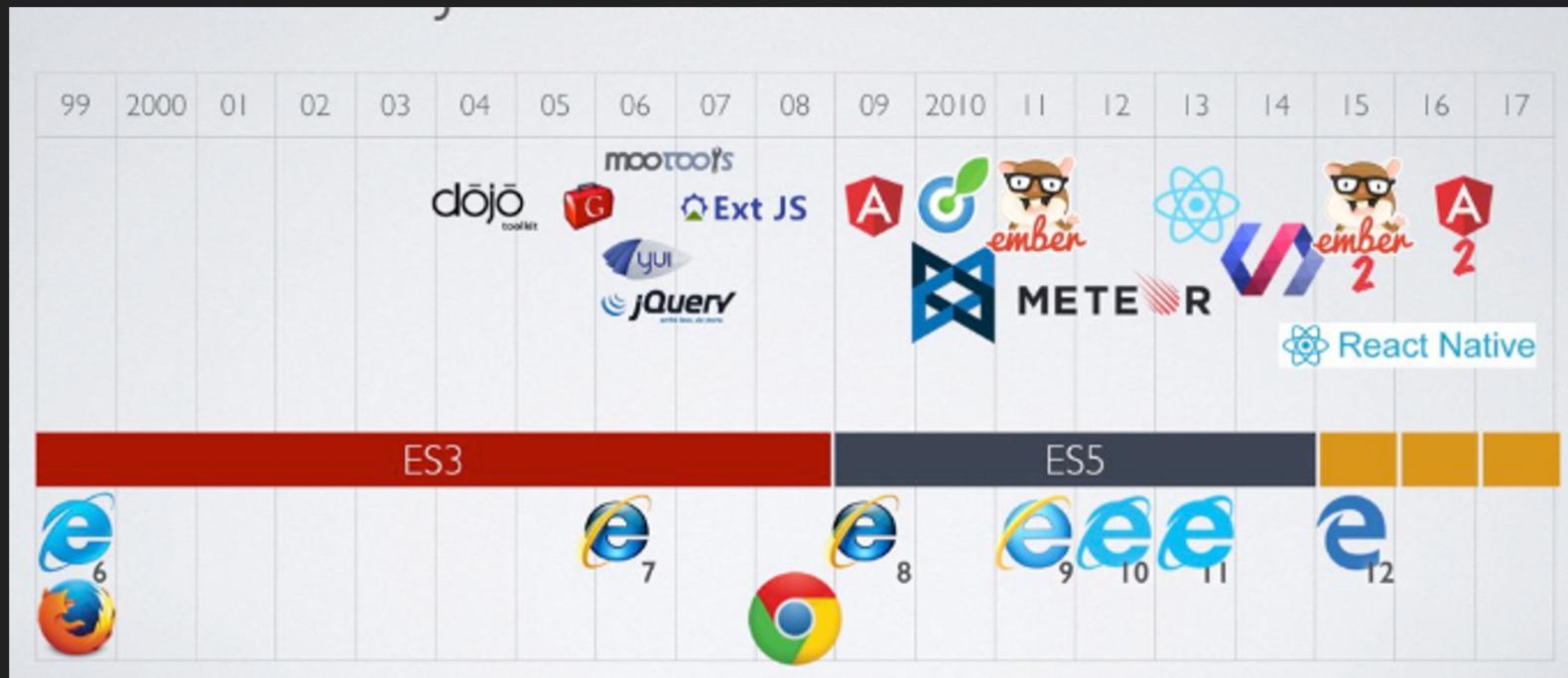
- not many clean code practices
- each framework = new practices, enforcing bad practices overall
- very rapid development often makes tools and frameworks obsolete fast

Why should you learn vanilla JavaScript before frameworks?

- If you master JavaScript fundamentals, your **only challenge** when learning new JS frameworks will be scoped to their **specific syntax**.
- Understanding JavaScript's core engineering principles is paramount, if you want to build yourself a decent web career.
- In the past 5 years, more than 10 frontend JS frameworks made the news. Guess how many will do the same in the next 5-10 years?
- "JQuery developers" are trying to catch up on Angular. Tomorrow, they'll be trying to catch up on React/Vue, and the loop will continue as technology advances..

JavaScript UI Frameworks Timeline

<http://www.evolutionoftheweb.com/>



ES6 Main Goals

- Fix (some of) ES5 problems
- backwards compatibility (Babel is a compiler to translate ES6 code to make valid ES5)
- modern syntax
- better suited for bigger complex application (ie. native modules)
- new features in the standard library (ie. native class support, arrow functions, modules)

let statement

- The **let** statement declares a block scope local variable, optionally initializing it to a value.

```
let x = 1;

if (x === 1) {
  let x = 2;

  console.log(x);
  // expected output: 2
}

console.log(x);
// expected output: 1
```

```
1 var foo = 'OUT'
2
3 {
4   var foo = 'IN'
5 }
6
7 console.log(foo) //IN
```

ES5 var statement

const statement

Constants are block-scoped, much like variables defined using the let statement. The value of a constant cannot change through reassignment, and it can't be redeclared.

```
'use strict'

const foo = function() {
  console.log('original')
}

foo = function() { // Error
  console.log('hijacked')
}

foo();
```

```
foo = function() {
^

TypeError: Assignment to constant variable.
  at Object.<anonymous> (/Users/veedzk/es6/
example.js:7:5)
  at Module._compile (module.js:399:26)
  at Object.Module._extensions..js (module.
js:406:10)
  at Module.load (module.js:345:32)
  at Function.Module._load (module.js:302:1
2)
  at Function.Module.runMain (module.js:431
:10)
  at startup (node.js:141:18)
  at node.js:977:3
```

for in... statement

- Use for...in to iterate over the properties of an object (the object keys)
- Also use for...in to iterate over the index values of an iterable like an array or a string

```
let oldCar = {  
    make: 'Toyota',  
    model: 'Tercel',  
    year: '1996'  
};  
  
for (let key in oldCar) {  
    console.log(`#${key} --> ${oldCar[key]}`);  
}  
  
// make --> Toyota  
// model --> Tercel
```

Iterating over object

```
let str = 'Turn the page';  
  
for (let index in str) {  
    console.log(`Index of ${str[index]}: ${index}`);  
}  
  
// Index of T: 0  
// Index of u: 1
```

Iterating over string

for of... statement

- Use for...of to iterate over the values in an iterable ie. array, string
- Also can iterate over maps, sets, generators, DOM node collections and the arguments object available inside a functions.

```
let str = 'abcde';

for (let char of str) {
  console.log(char.toUpperCase().repeat(3));
}

// AAA
// BBB
// ...
```

template literals

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them. (*ES5 => template strings*)

```
var a = 5;
var b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

ES6: Template Literals

```
var a = 5;
var b = 10;
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b)
// "Fifteen is 15 and
// not 20."
```

ES5: Embedded Expressions with normal strings

Arrow functions

```
var elements = [  
  'Hydrogen',  
  'Helium',  
  'Lithium',  
  'Beryllium'  
];  
  
elements.map(function(element) {  
  return element.length;  
}); // [8, 6, 7, 9]  
  
elements.map(element => {  
  return element.length;  
}); // [8, 6, 7, 9]  
  
elements.map(element => element.length); // [8, 6, 7, 9]  
  
elements.map(({ length }) => length); // [8, 6, 7, 9]
```

- An arrow function expression has a shorter syntax than a function expression and does not have its own `this`, `arguments`, `super`, or `new.target`.
- These function expressions are best suited for non-method functions, and they cannot be used as constructors.

Destructuring.

- Destructuring syntax allows you to extract data from arrays and objects with more ease and less syntactic clutter.
- We can extract properties from the object and assign them to new const variables

Object Destructuring

```
const names = {cat: 'Bob', dog: 'Fred', alligator: 'Benedict'};  
  
const {cat, dog, alligator} = names;
```

Array Destructuring (*use comma to skip items*)

```
const names = ['Bob', 'Fred', 'Benedict'];  
  
const [cat, , alligator] = names;
```

Default Parameters

- Default values can be defined for your function parameters in JavaScript. The default value will be used when an argument is missing or it evaluates to undefined.

Value 3 is used when y is not provided or when undefined is provided

```
function add(x, y = 3) {  
  console.log(x + y);  
}
```

```
add(3, 9); // 12  
add(3) // 6  
add(12, undefined) // 15
```

Default params can ensure you have an empty array or object literal

```
function addToGuestList(guests, list = []) {  
  console.log([...guests, ...list]);  
}
```

Object & Arrays

Spread (...) operator

- It lets you use the spread (...) operator to copy enumerable properties from one object to another

When we don't use spread operator

```
var mid = [3, 4];
var arr = [1, 2, mid, 5, 6];

console.log(arr);
```

Outputs

```
[1, 2, [3, 4], 5, 6]
```

When we use spread operator

```
var mid = [3, 4];
var arr = [1, 2, ...mid, 5, 6];

console.log(arr);
```

Outputs

```
[1, 2, 3, 4, 5, 6]
```

Spread (...) operator cont.

- Math operator wants to find maximum value of multiple numbers, you can't use array as input.
- Instead of using apply, we can use the spread syntax to expand our array elements and inputs each element into the Math.max() method

When we use apply method

```
var arr = [2, 4, 8, 6, 0];

function max(arr) {
    return Math.max.apply(null, arr);
}

console.log(max(arr));
```

When we use spread method

```
var arr = [2, 4, 8, 6, 0];
var max = Math.max(...arr);

console.log(max);
```

ES5 recap: map

- The `array.prototype.map()` method creates a new array with the results of calling a provided function on every element in the calling array.

Array.map()

```
var array1 = [1, 4, 9, 16];

// pass a function to map
const map1 = array1.map(x => x * 2);

console.log(map1);
// expected output: Array [2, 8, 18, 32]
```

Array.map() vs for loop

```
let arr = [1, 2, 3]

let duplicatedArr = arr.map(function(el) {
  return el * 2
}) // [2, 4, 6]
```

```
let duplicatedArr = []
for (let i=0; i< arr.length; i++) {
  duplicatedArr.push(arr[i] * 2)
}
```

ES5 recap: Filter

- The `array.prototype.filter()` method creates a new array with all elements that pass the test implemented by the provided function.

`Array.filter()`

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

```
let arr = [1, 2, 3]

let evenArr = arr.filter(function(el){
    return el % 2 === 0
}) // [2]
```

ES5 recap: Reduce

- The `array.prototype.reduce()` method executes a reducer function (that you provide) on each member of the array resulting in a single output value.

Array.reduce()

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;
```

```
// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

Array.reduce() vs for loop

```
let arr = [1, 2, 3]

let sum = arr.reduce(function(sumSoFar, el){
  return sumSoFar + el
}, 0) // 6
```

```
let sum = 0
for (let i=0; i< arr.length; i++) {
  sum = sum + arr[i]
}
```

Map

- Map holds key-value pairs. It's similar to an array but we can define our own index.
- All indexes are unique and we can use any value as key or value.

Map keys can be any value

```
var map = new Map();
map.set('name', 'John');
map.set('name', 'Andy');
map.set(1, 'number one');
map.set(NaN, 'No value');

map.get('name'); // Andy. Note John is replaced
map.get(1); // number one
map.get(NaN); // No value
```

Other useful methods used in Map

```
var map = new Map();
map.set('name', 'John');
map.set('id', 10);

map.size; // 2. Returns the size of the map.

map.keys(); // outputs only the keys.
map.values(); // outputs only the values.

for (let key of map.keys()) {
  console.log(key);
}
```

Object property declaration

- Shorthand key names can now be used to define an object, when key's have same name as variables passed in as properties.

ES6 Syntax

```
let cat = 'Miaow';
let dog = 'Woof';
let bird = 'Peet peet';

let someObject = {
  cat,
  dog,
  bird
}
```

ES5 Syntax

```
var cat = 'Miaow';
var dog = 'Woof';
var bird = 'Peet peet';

var someObject = {
  cat: cat,
  dog: dog,
  bird: bird
}
```

Classes

Classes

- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance.
- The class syntax **does not** introduce a new object-oriented inheritance model to JavaScript.
- * Function declaration are hoisted, class declaration are not.

Class declaration

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
}
```

Class Expression

```
let Rectangle = class {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
};
```

Sub classing with extends

- The **extends** keyword is used in class declarations or class expressions to create a class as a child (sub class) of another class.

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  speak() {  
    console.log(this.name + ' makes a noise.');//  
  }  
}
```

```
class Dog extends Animal {  
  constructor(name) {  
    super(name); // call the super class constructor  
  }  
  
  speak() {  
    console.log(this.name + ' barks.');//  
  }  
}
```

```
let d = new Dog('Mitzie');  
d.speak(); // Mitzie barks.
```

Super class calls with super

- The **super** keyword is used to call corresponding methods of super class. This is one advantage over prototype-based inheritance.

sub class

```
class Lion extends Cat {  
  speak() {  
    super.speak();  
    console.log(`${this.name} roars.`);  
  }  
}
```

output

```
let l = new Lion('Fuzzy');  
l.speak();  
// Fuzzy makes a noise.  
// Fuzzy roars.
```

super class

```
class Cat {  
  constructor(name) {  
    this.name = name;  
  }  
  
  speak() {  
    console.log(`${this.name} makes a noise.`);  
  }  
}
```

Setters & Getters

- The **get** syntax binds an object property to a function that will be called when that property is looked up.
- The **set** syntax binds an object property to a function to be called when there is an attempt to set that property.

getter

```
var obj = {
  log: ['a', 'b', 'c'],
  get latest() {
    if (this.log.length == 0) {
      return undefined;
    }
    return this.log[this.log.length - 1];
  }

console.log(obj.latest);
// expected output: "c"
```

setter

```
var language = {
  set current(name) {
    this.log.push(name);
  },
  log: []
}

language.current = 'EN';
language.current = 'FA';

console.log(language.log);
// expected output: Array ["EN", "FA"]
```

Static methods

- The `static` keyword defines a static method for a class.
- Static methods aren't called on instances of the class. Instead, they're called on the class itself.
- These are often utility functions, such as functions to create or clone objects.

Static method

```
class ClassWithStaticMethod {  
    static staticMethod() {  
        return 'static method has been called.';  
    }  
}  
  
console.log(ClassWithStaticMethod.staticMethod());  
// expected output: "static method has been called."
```