Introduction to

# JavaScript

the programming language of the Web

# About Me

Eman Fathi

## Agenda

- **Day 1** : JS Fundamentals
- **Day 2** : JS Fundamentals
- **Day 3** : JS Fundamentals
- **Day 4** : JS Fundamentals
- **Day 5** : JS Fundamentals
- **Day 6** : ES Next
- **Day7 :** ES Next
- **Day8:** ES Next
- **Day9:** Advanced JS
- **Day10:** Advanced JS
- **Day 11:** Advanced JS

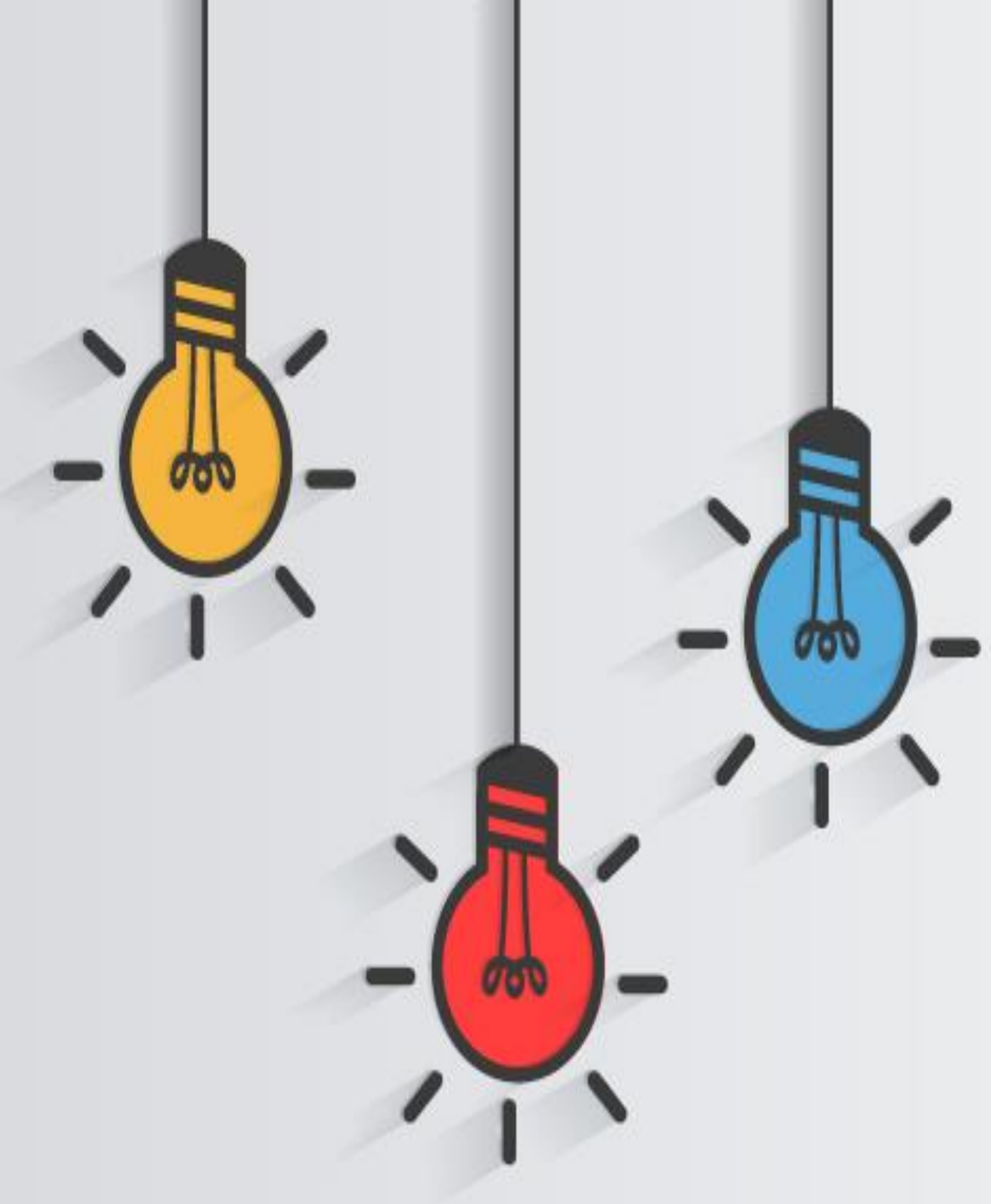
## Agenda for OS Track

- **Day 1** : Core JS
- **Day 2** : Core JS
- **Day 3** : Language Objects
- **Day 4** : BOM and DOM
- **Day 5** : Events and OOP
- **Day 6** : ESNext
- **Day7** : ESNext
- **Day8 :** ESNext

# Course Assessment

- Labs : 40%
- Project | Exam : 60%

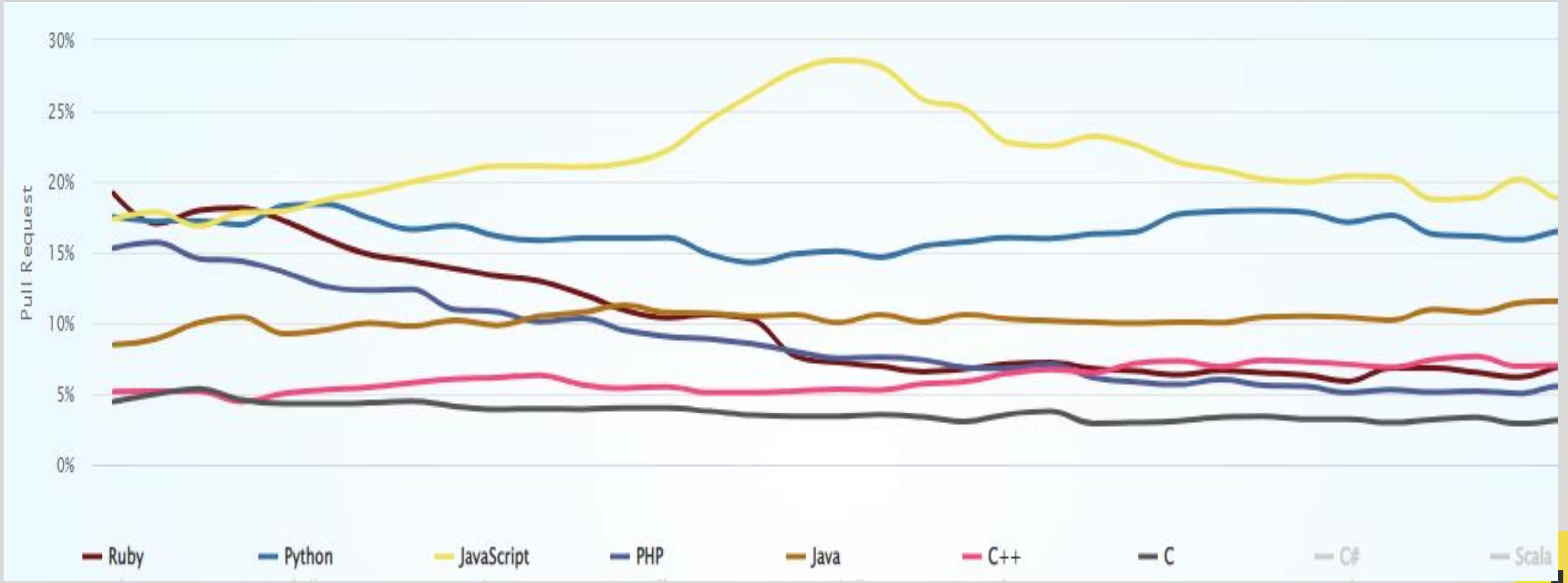
## Why should we learn Javascript?



# JavaScript is enormously popular

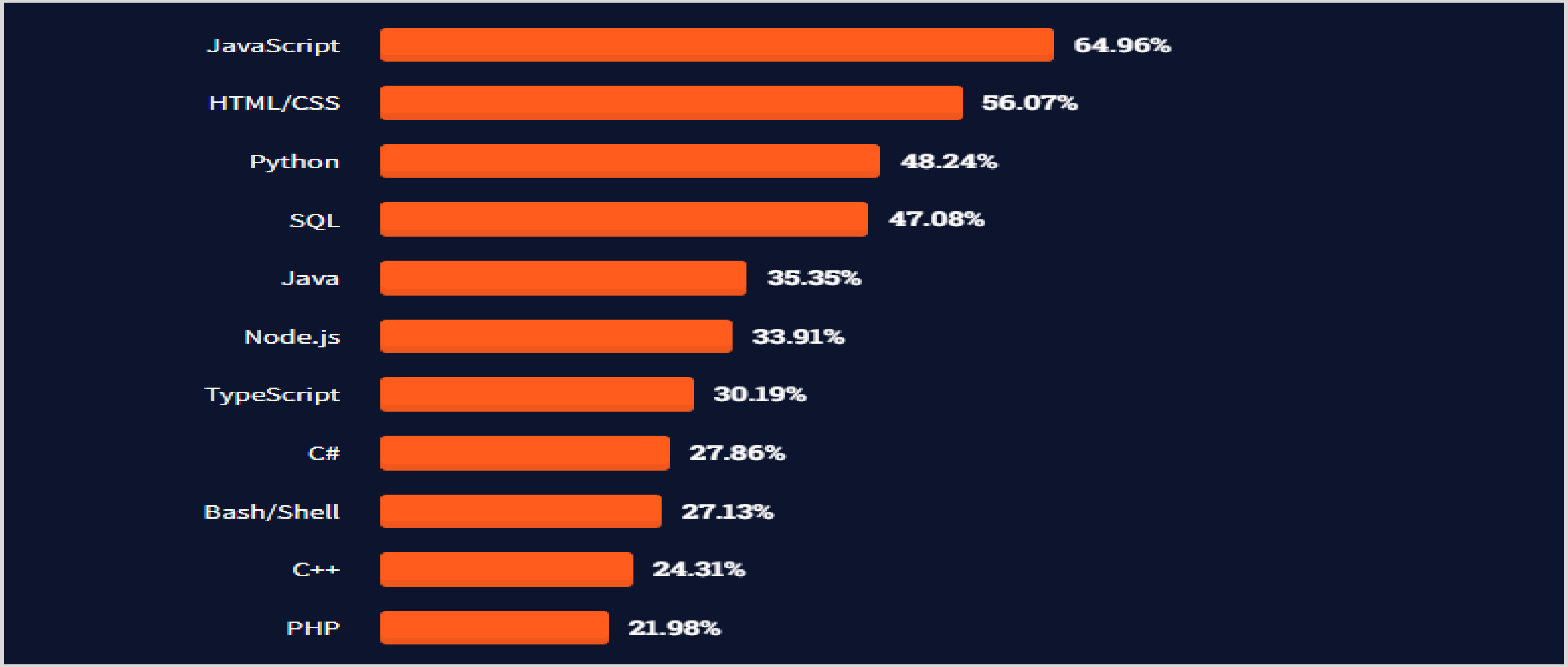
## Top 5 Languages on GitHub

By number of pull requests



# JavaScript is enormously popular

## Most popular language on Stack Overflow



# Static Web Page

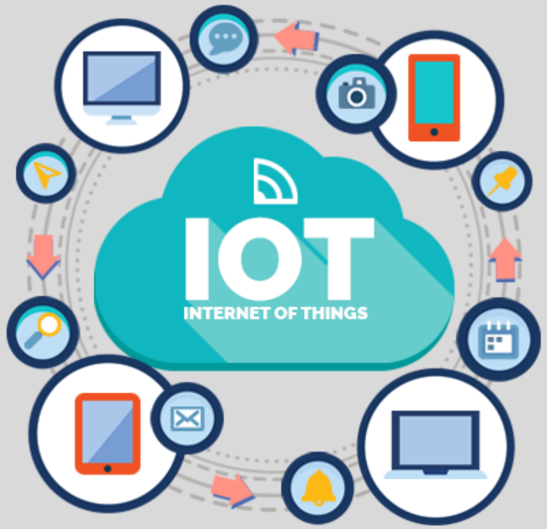
02/16/2008
Welcome to the The Map Store Great Service since 1971
The Map Store of Minnesota
Your Best Resource for Laminated and Framed Wall Maps
651-227-6277
info about The Map Store
Large Wall map of Minnesota
SIZE: 115'' x 76''
Bob's Birthday Gift Suggestions
Starship Earth I
WALL MAPS:
World
International
USA
States & Cities
Zip Code
Raven
Globes More than 100
World National Geographic 3 sheet Mural
Register Here
Mpls - St Paul Zip Code Map
Magnifiers
Geochron World Clock
ATLASES:
Delorme
Mpls-St Paul
World Map "upside down"

lightGallery: thumbnails
https://sachinchoolur.github.io/lightgallery.js/#lg=1&slide=13
14 / 20

# Dynamic Web Page

MACHINE
LEARNING

# EVERYWHERE


IOT
INTERNET OF THINGS

JS
node
METEOR
ember
polymer
BACKBONE.JS
Knockout.
aurelia
ext js
JS

# THERE IS NOTHING YOU CAN'T DO WITH JAVASCRIPT (WELL, ALMOST...)

Front-end apps

THIS COURSE

Dynamic effects and web applications in the browser 😍

100% based on JavaScript. They might go away, **but JavaScript won't!**

Back-end apps

Web applications on web servers

Native mobile applications

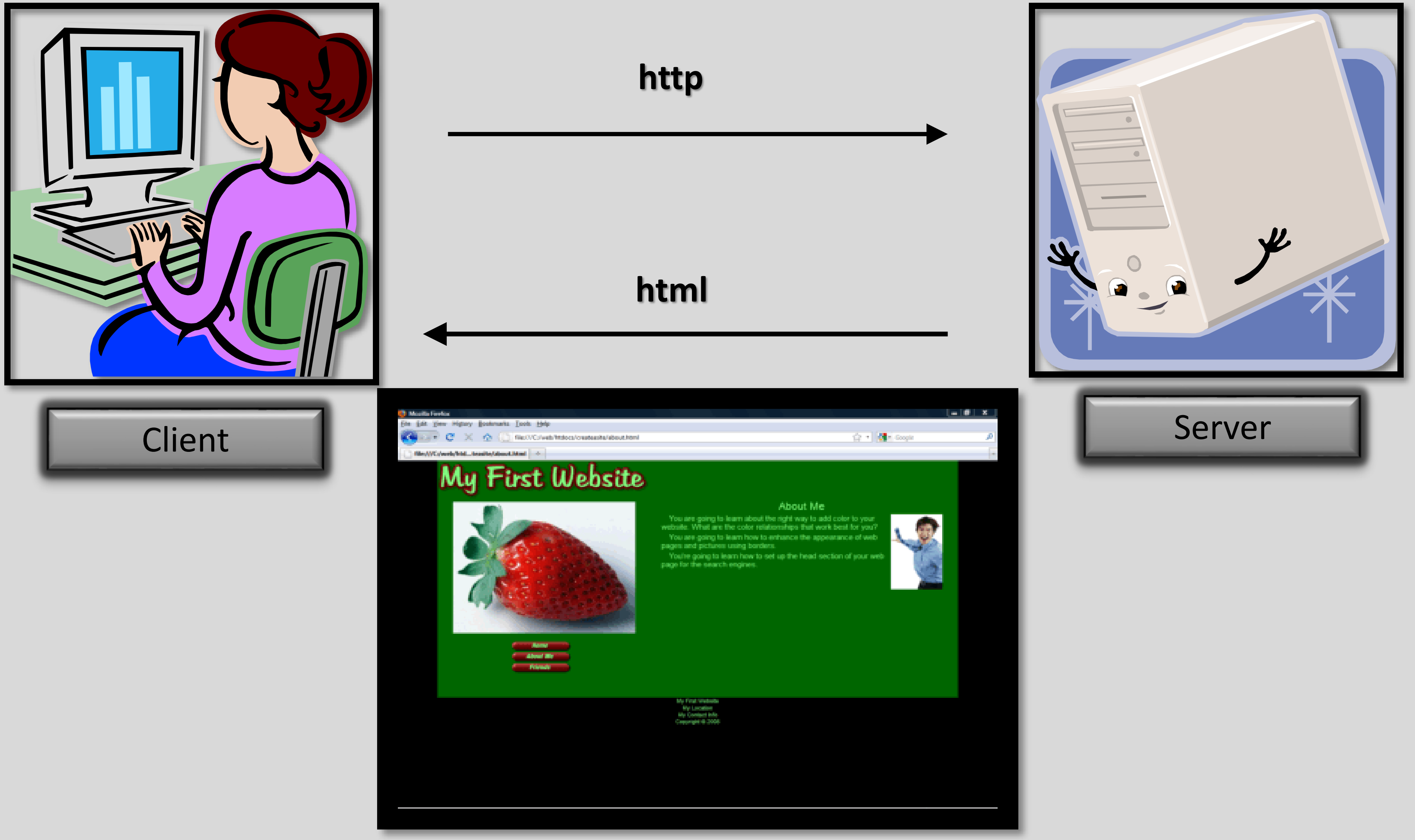
Native desktop applications

JS

## What you should already know

- A general understanding of the **Internet** and the **World Wide Web** (WWW).
- Good working knowledge of **HyperText Markup Language** (HTML).
- Some **programming experience**.

# How Does the Web Work?



My First Website

About Me

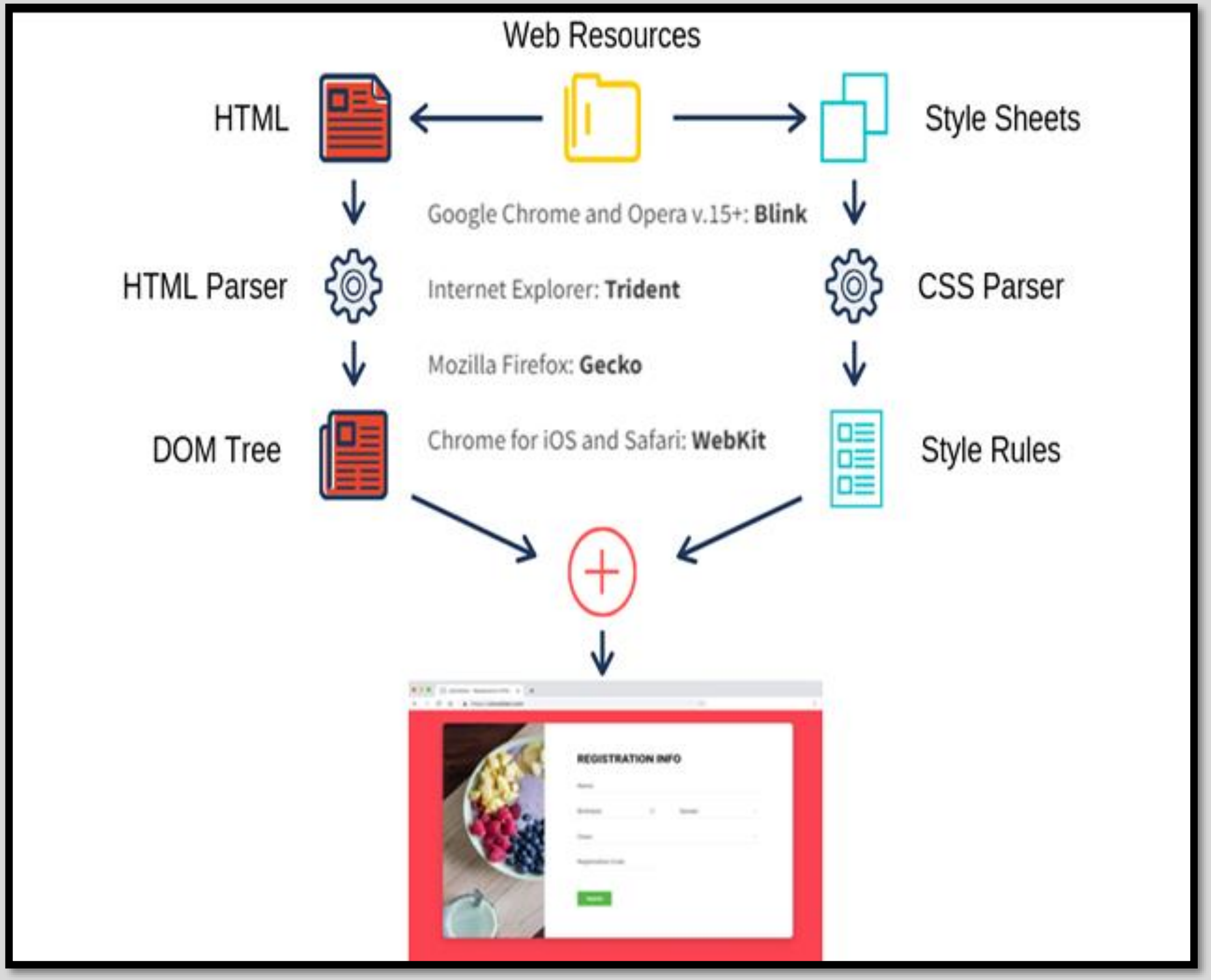
You are going to learn about the right way to add color to your website. What are the color relationships that work best for you?

You are going to learn how to enhance the appearance of web pages and pictures using borders.

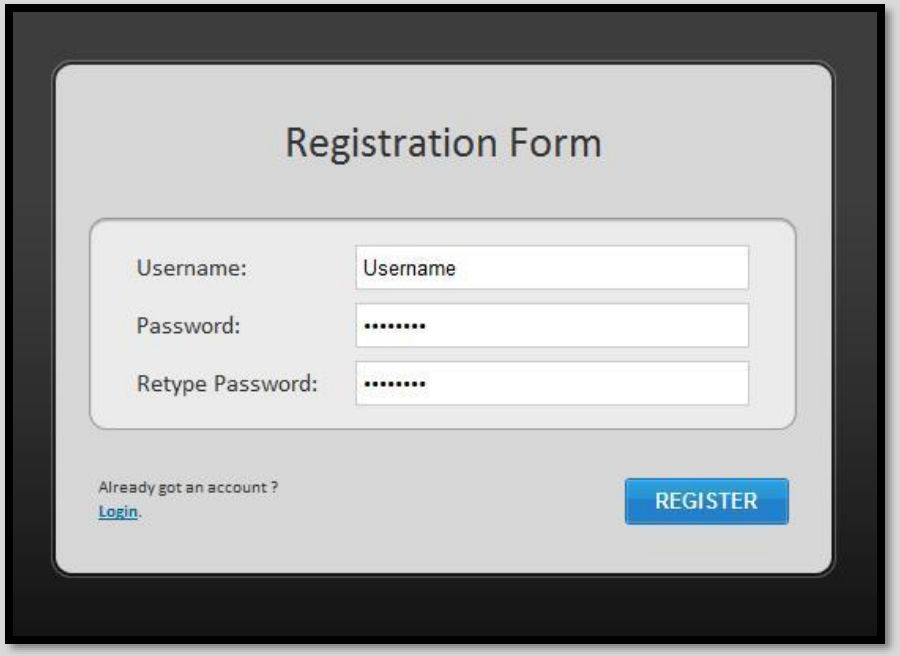
You're going to learn how to set up the head section of your web page for the search engines.

My First Website
My Location
My Contact Info
Copyright © 2008

# How Does the Web Work?


Web Resources
HTML
Style Sheets
Google Chrome and Opera v.15+: **Blink**
HTML Parser
Internet Explorer: **Trident**
CSS Parser
Mozilla Firefox: **Gecko**
DOM Tree
Chrome for iOS and Safari: **WebKit**
Style Rules
REGISTRATION INFO

# Traffic


Registration Form

| | |
|---|---|
| Username: | Username |
| Password: | ••••••••• |
| Retype Password: | ••••••••• |

Already got an account ?
Login.

*Send Data*

*Send Page*

If(userName=="")

If(Pass=="")

If(ConfPass=="")

If(Pass==CnfPass)

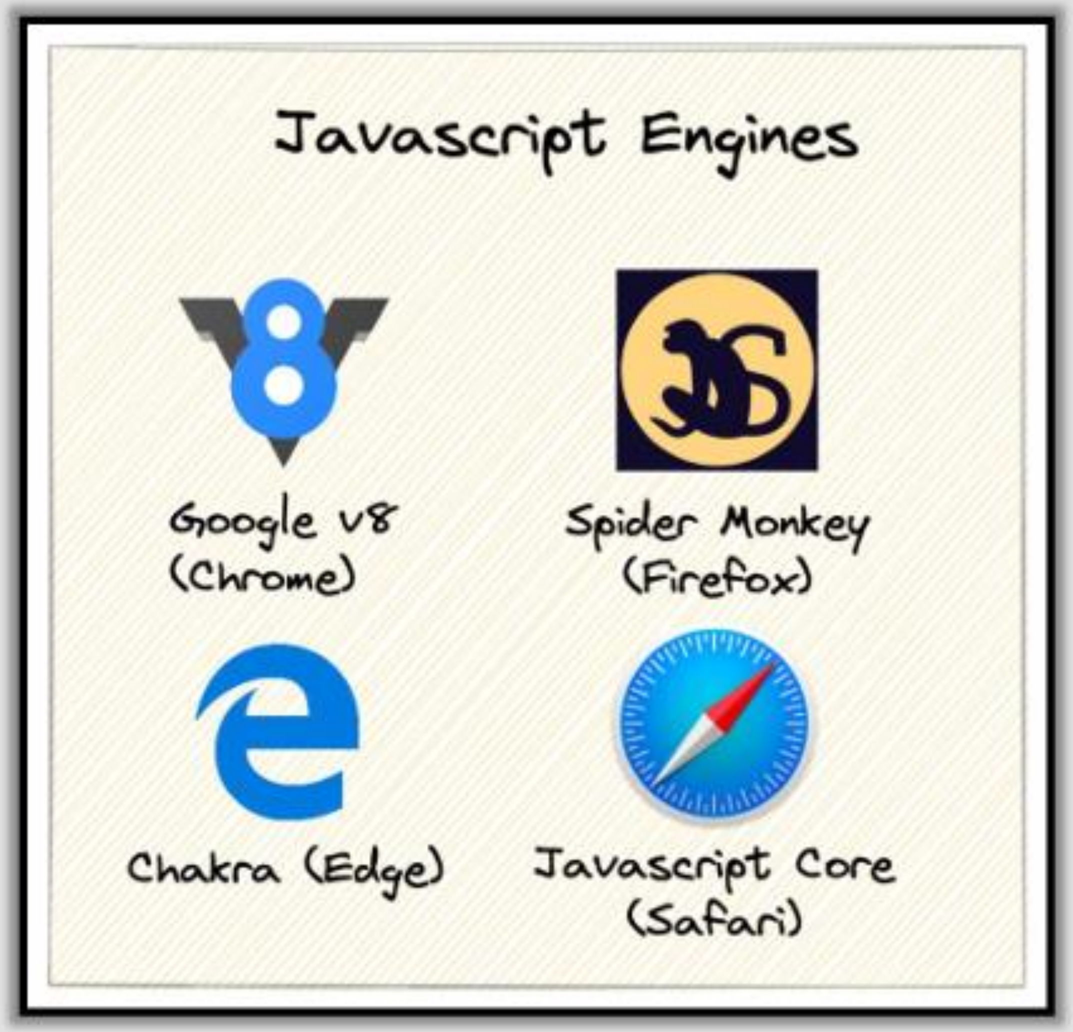
## JavaScript History

# JavaScript history

Brendan Eich convinced his boss at **Netscape** that the Navigator browser should have its own **scripting language**, and that only a new language would do, a new language designed and implemented in big hurry, and that no existing language should be considered for that role.

Brendan Eich

# JavaScript History



## Introduction to JavaScript

- JavaScript is *the* language of the Web. It was introduced in 1995 by **Netscape**, the company that created the first browser by the same name. Other browsers followed, and JavaScript quickly gained acceptance as the client-side scripting language on the internet. Today it is used by millions of websites to add **functionality** and **responsiveness**, **validate forms**, **communicate with the server**, and much more.
- Originally, JavaScript was named **LiveScript** but was renamed **JavaScript** as a marketing strategy to benefit from the exploding popularity of the **Java** programming language at that time. As it turned out, Java evolved into a server-side language and did not succeed on the browser, whereas JavaScript did. The change of name is unfortunate because it has caused a lot of confusion.

## Introduction to JavaScript

- Soon after its initial release the JavaScript language was submitted to **ECMA International** (European Computer Manufacture’s Association)-- an international non-profit standards organization -- for consideration as an industry standard. It was accepted and today we have a standardized version which is called **ECMAScript.** There are several implementations of the ECMAScript standard, including **JavaScript**, **Jscript** (Microsoft), and **ActionScript** (Adobe). The ECMAScript language is undergoing continuous improvements. The next standards release is ECMAScript **6th** edition and is code named "**Harmony**".
- Initially many developers felt it was an inferior language because of its perceived **simplicity**. Also, users would frequently disable JavaScript in their browsers because of security concerns. However, over the last few years, starting with the introduction of **AJAX** and the related Web 2.0 transition, it became clear that JavaScript allows developers to **build powerful and highly responsive web applications**. Today JavaScript is considered *the* Web language and an essential tool for building modern and responsive web apps.

## JavaScript History

- JavaScript language first became available in the web browser Netscape Navigator 2,it was called **LiveScript.**

- Since Java was the hot technology of the time, Netscape decided that **JavaScript** sounded more exciting.

- Microsoft decided to add their own brand of JavaScript to Internet Explorer, which they named **JScript**.

# ECMAScript Releases

1995 JavaScript is born as LiveScript

1997 ECMAScript standard is established

1999 ES3 comes out and IE5 is all the rage

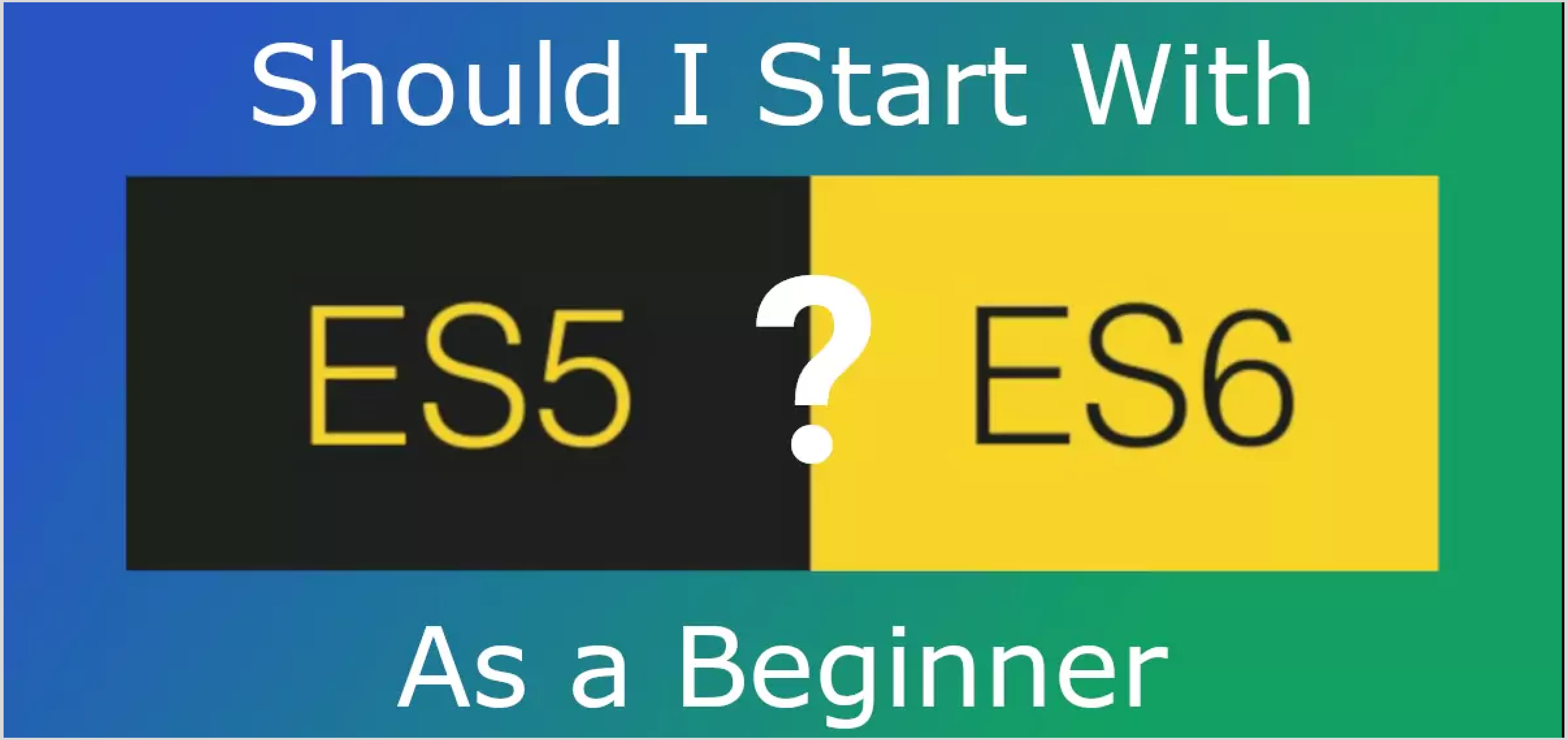
2000 XMLHttpRequest, a.k.a. AJAX, gains popularity

2009 ES5 comes out and standard JSON

2015 ES6/ECMAScript2015 comes out

2016 ES7/ECMAScript2016 comes out

2017 ES.Next

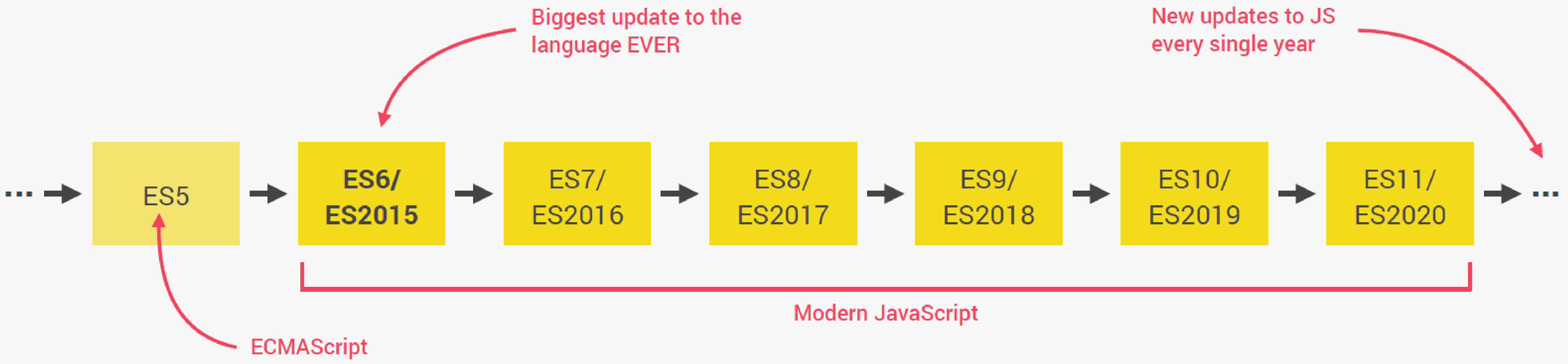


Should I Start With

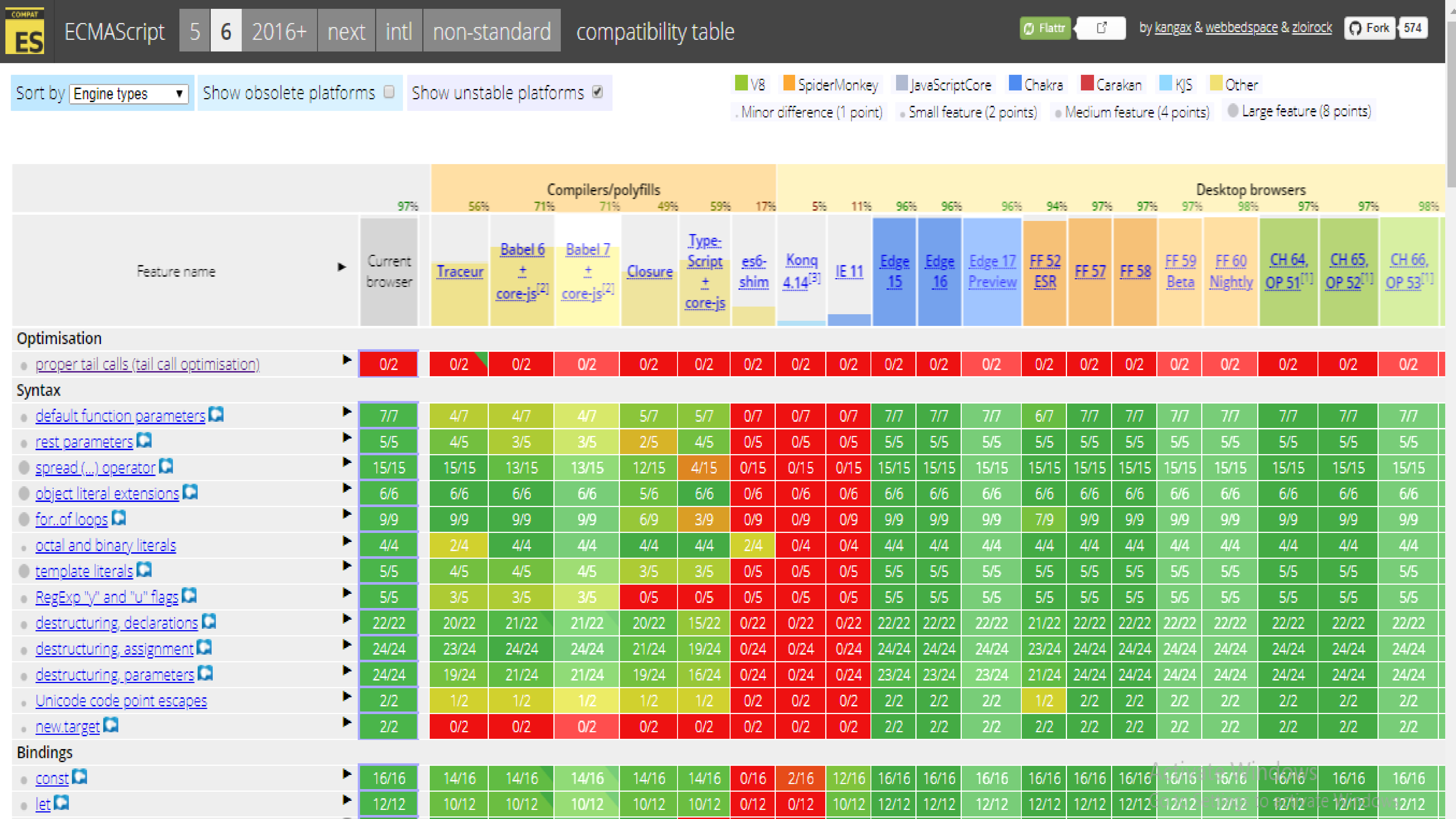
ES5 ? ES6

As a Beginner

# ECMAScript Releases



*Learn Modern Javascript but without forgetting the older parts*



ECMAScript | 5 | 6 | 2016+ | next | intl | non-standard | compatibility table

Flattr | by kangax & webbedspace & zloirock | Fork 574

Sort by: Engine types | Show obsolete platforms ☐ | Show unstable platforms ☑

🟩 V8 🟧 SpiderMonkey ⬜ JavaScriptCore 🟦 Chakra 🟥 Carakan 🟦 KJS 🟨 Other

Minor difference (1 point) · Small feature (2 points) · Medium feature (4 points) · Large feature (8 points)

| | | Compilers/polyfills | | | | | | Desktop browsers | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 97% | 56% | 71% | 71% | 49% | 59% | 17% | 5% | 11% | 96% | 96% | 96% | 94% | 97% | 97% | 97% | 98% | 97% | 97% | 98% |
| Feature name | Current browser | Traceur | Babel 6 + core-js[2] | Babel 7 + core-js[2] | Closure | Type-Script + core-js | es6-shim | Konq 4.14[3] | IE 11 | Edge 15 | Edge 16 | Edge 17 Preview | FF 52 ESR | FF 57 | FF 58 | FF 59 Beta | FF 60 Nightly | CH 64, OP 51[1] | CH 65, OP 52[1] | CH 66, OP 53[1] |
| **Optimisation** | | | | | | | | | | | | | | | | | | | | |
| proper tail calls (tail call optimisation) | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 |
| **Syntax** | | | | | | | | | | | | | | | | | | | | |
| default function parameters | 7/7 | 4/7 | 4/7 | 4/7 | 5/7 | 5/7 | 0/7 | 0/7 | 0/7 | 7/7 | 7/7 | 7/7 | 6/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 |
| rest parameters | 5/5 | 4/5 | 3/5 | 3/5 | 2/5 | 4/5 | 0/5 | 0/5 | 0/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 |
| spread (...) operator | 15/15 | 15/15 | 13/15 | 13/15 | 12/15 | 4/15 | 0/15 | 0/15 | 0/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 |
| object literal extensions | 6/6 | 6/6 | 6/6 | 6/6 | 5/6 | 6/6 | 0/6 | 0/6 | 0/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 |
| for..of loops | 9/9 | 9/9 | 9/9 | 9/9 | 6/9 | 3/9 | 0/9 | 0/9 | 0/9 | 9/9 | 9/9 | 9/9 | 7/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 |
| octal and binary literals | 4/4 | 2/4 | 4/4 | 4/4 | 4/4 | 4/4 | 2/4 | 0/4 | 0/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 |
| template literals | 5/5 | 4/5 | 4/5 | 4/5 | 3/5 | 3/5 | 0/5 | 0/5 | 0/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 |
| RegExp "y" and "u" flags | 5/5 | 3/5 | 3/5 | 3/5 | 0/5 | 0/5 | 0/5 | 0/5 | 0/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 |
| destructuring, declarations | 22/22 | 20/22 | 21/22 | 21/22 | 20/22 | 15/22 | 0/22 | 0/22 | 0/22 | 22/22 | 22/22 | 22/22 | 21/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 |
| destructuring, assignment | 24/24 | 23/24 | 24/24 | 24/24 | 21/24 | 19/24 | 0/24 | 0/24 | 0/24 | 24/24 | 24/24 | 24/24 | 23/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 |
| destructuring, parameters | 24/24 | 19/24 | 21/24 | 21/24 | 19/24 | 16/24 | 0/24 | 0/24 | 0/24 | 23/24 | 23/24 | 23/24 | 21/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 |
| Unicode code point escapes | 2/2 | 1/2 | 1/2 | 1/2 | 1/2 | 1/2 | 0/2 | 0/2 | 0/2 | 2/2 | 2/2 | 2/2 | 1/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 |
| new.target | 2/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 |
| **Bindings** | | | | | | | | | | | | | | | | | | | | |
| const | 16/16 | 14/16 | 14/16 | 14/16 | 14/16 | 14/16 | 0/16 | 2/16 | 12/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 |
| let | 12/12 | 10/12 | 10/12 | 10/12 | 10/12 | 10/12 | 0/12 | 0/12 | 10/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 |

# Embedding JavaScript in HTML

Client-side JavaScript code is embedded within HTML documents in four ways:

- Inline, between a pair of <script> and </script> tags
- From an external file specified by the src attribute of a <script> tag
- In an HTML event handler attribute, such as onclick or onmouseover

# The <script> Element

JavaScript code can appear inline within an HTML file between **<script>** and **</script>** tags.

```
<!DOCTYPE html> <!-- This is an HTML5 file -->
<html>  <!-- The root element -->
        <head>
                <title>Home Page</title>
                        </head>
        <body>
                <h1>Welcome to JS.</h1>
                <script>
                        //A script of js code
                </script>
        </body>
</html>
```

## Scripts in External Files

The <script> tag supports a **src** attribute that specifies the URL of a file containing JavaScript code.

```
<!DOCTYPE html> <!-- This is an HTML5 file -->
<html>
        <head>
                <title>Home Page</title>
                <script src="../../scripts/util.js"></script>
        </head>
        <body>
                <h1>Welcome to JS.</h1>
        </body>
</html>
```

## Event Handlers in HTML

JavaScript functions that are registered with the web browser and then invoked by the web browser in response to events (such as user input).

Event handler properties like onclick mirror HTML attributes with the same names, and it is also possible to define event handlers by placing JavaScript code in HTML attributes.

```
<input type="button" id="btn" value="click here"
     onclick="alert('click again'); this.disabled = true;">
```

## Case Sensitivity

JavaScript is a case-sensitive language.

This means that language keywords, variables, function names, and other *identifiers* must always be typed with a consistent capitalization of letters.

- Example while keyword, must be typed “while,” not “While” or “WHILE.”
- **online**, **Online**, **OnLine**, and **ONLINE** are four distinct variable names.

## Comments

JavaScript supports two styles of comments.

- Any text between a  and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is also treated as a comment; these comments may span multiple lines.

```
//This is a single-line comment.
/* This is also a comment */
   This is yet another comment.
    It has multiple lines.
*/
```

## Reserved Words

JavaScript reserves a number of identifiers as the keywords of the language itself.
You cannot use these words as identifiers in your programs:

break delete function return typeof case do if switch var catch else in this void continue
false instanceof throw while debugger finally new true with default for null try

# Core JavaScript

- Types, Values, and Variables
- Expressions and Operators
- Statements
- Arrays
- Objects
- Functions
- Classes
- Pattern Matching with **Regular Expressions (Report)**

ES5 classic

## Types, Values, and Variables

JS

## Identifiers and Reserved Words

An identifier is simply a name. In JavaScript, identifiers are used to **name variables** and **functions** in JavaScript code.

A JavaScript identifier must begin with a **letter**, an **underscore** (_), or a **dollar sign** ($). Subsequent characters can be letters, digits, underscores, or dollar signs.

(Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.)

These are all legal identifiers:

- i
- my_variable_name
- v13
- _dummy
- $str

# Types

JavaScript types can be divided into two categories:

- *primitive types.*
  - number.
  - string.
  - boolean.
  - undefined.
- *object types*.
  - Language objects
  - Browser objects
  - Document objects
  - User-defined objects
  - null

# Variable Declaration

Before you use a variable in a JavaScript program, you should *declare* it. Variables are declared with the **var** keyword, like this:

```
var i;
var sum;
```

You can also declare multiple variables with the same var keyword:

```
var i, sum;
```

And you can combine variable declaration with variable initialization:

```
var message = "hello";
var i = 0, j = 0, k = 0;
```

If you don’t specify an initial value for a variable with the var statement, the variable is declared, but its value is undefined until your code stores a value into it.

# number

JavaScript does not make a distinction between integer values and floating-point values.
All numbers in JavaScript are represented as floating-point values.
**Number** is a numeric data type in the double-precision 64-bit floating point format .

```
Integer
0
3500
10000000
Floating-Point
3.14
2345.789
.3333333333333333333
6.02e23                6.02 × 1023
1.4738223E-32   1.4738223 × 10−32
```

## string

Set of characters enclosed  within a matched pair of single or double quotes (' or ").

```
""  The empty string: it has zero characters
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"This string\n has two lines"
"π is the ratio of a circle's circumference to
its diameter"
```

## string

String literals must be written on a single line. In ECMAScript 5, however, you can break a string literal across multiple lines by ending each line except the last with a backslash (\).

```
"two\n lines"   A string representing 2 lines
written on one line

 A one-line string written on 3 lines.
ECMAScript 5 only.
"one\
long\
line";
```

# Long literal strings

Sometimes, your code will include strings which are very long.

- You can use the + operator to append multiple strings together, like this:

```
var longString = "This is a very long string which needs "
        + "to wrap across multiple lines because "
        + "otherwise my code is unreadable.“;
```

- Or you can use the backslash character ("\") at the end of each line to indicate that the string will continue on the next line.

```
var longString = "This is a very long string which needs \
        to wrap across multiple lines because \
        otherwise my code is unreadable.";
```

# Escape Sequences in String

The **backslash character** (\) has a special purpose in JavaScript strings.

Combined with the character that follows it, it represents a character that is not otherwise representable within the string. For example, \n is an *escape sequence* that represents a newline character.

the backslash allows you to escape from the usual interpretation of the single-quote character. Instead of using it to mark the end of the string, you use it as an apostrophe:

```
'You\'re right, it can\'t be a quote'
```

## boolean

- Boolean is a logical data type that can have only the values **true** or **false**.
- JavaScript Booleans can have one of two values: **true** or **false**.

```
var completed = false;
var isMember = true;
if(isMember)
    discount = 15;
```

## null

- The value null represents the intentional absence of any object value.

```
var element = null;
```

## undefined

- A primitive value automatically assigned to variables that have just been declared or to formal arguments for which there are no actual arguments.

```
var x;
console.log(x);

var temp;
typeof temp; //"undefined"
```

## Difference between null and undefined

```
typeof null         // "object" (not "null" for
legacy reasons)
typeof undefined    // "undefined"
null === undefined // false
null  == undefined // true
null === null //true
null == null //true
!null //true
```

# The + Operator

The binary + operator adds numeric operands or concatenates string operands

```
1 + 2  => 3
"hello" + " " + "there"  => "hello there"
"1" + "2"  => "12“


1 + 2  => 3: addition
"1" + "2"  => "12": concatenation
"1" + 2  => "12": concatenation after number-to-string
true + true  => 2: addition after boolean-to-number
2 + null  => 2: addition after null converts to 0
2 + undefined  => NaN: addition after undefined converts to NaN
```

# Template Literals & multiline String

## Template Literals

Template literals are ECMAScript 6's answer to the following features that JavaScript lacked in ECMAScript 5 and in earlier versions:

✓Multiline strings A formal concept of multiline strings

✓Basic string formatting The ability to substitute parts of the string for values contained in variables.

At their simplest, template literals act like regular strings delimited by **backticks (`)** instead of double or single quotes.

```
let message = `Hello world!`;
console.log(message); // "Hello world!"
console.log(typeof message); // "string"
console.log(message.length); // 12
```

There's no need to escape either double or single quotes inside template literals.

## Multiline Strings

JavaScript developers have wanted a way to create multiline strings since the first version of the language. But when you’re using double or single quotes, strings must be completely contained on a single line.

**Pre-ECMAScript 6** Workarounds Thanks to a long-standing syntax bug, JavaScript does have a workaround for creating multiline strings. You can create multiline strings by using a **backslash (\)** before a newline.

```
var message = "Multiline \

string";

console.log(message); // "Multiline string“
```

ECMAScript 6’s template literals make multiline strings easy because there’s no special syntax. Just include a newline where you want, and it appears in the result

```
let message = `Multiline
string`;
console.log(message); // "Multiline
                        // string"
console.log(message.length); // 16
```

All whitespace inside the backticks is part of the string, so be careful with indentation. For example:

```
let message = `Multiline
               string`;
console.log(message); // "Multiline
                                    // string"
console.log(message.length); // 31
```

# Making Substitutions

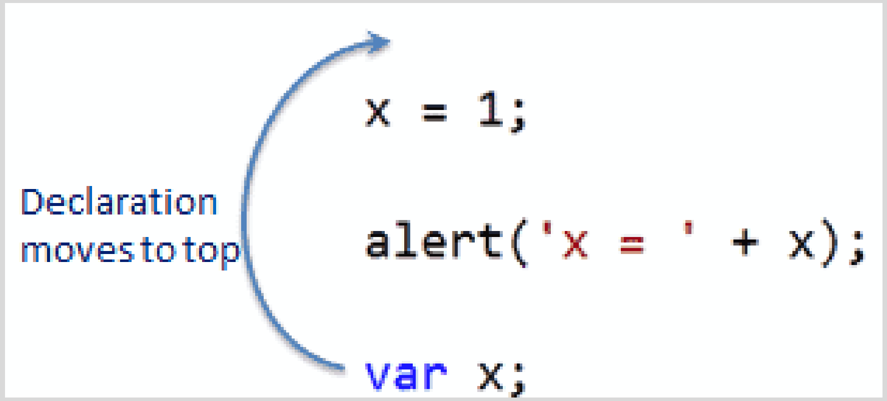
Substitutions are delimited by an opening ${ and a closing } that can have any JavaScript expression inside. The simplest substitutions let you embed local variables directly into a resulting string.

```
let name = "Nicholas",
message = `Hello, ${name}.`;
console.log(message); // "Hello, Nicholas.“
```

Because all substitutions are JavaScript expressions, you can substitute more than just simple variable names. You can easily embed calculations, function calls, and more.

```
let count = 10,
price = 0.25,
message = `${count} items cost $${(count * price).toFixed(2)}.`;
console.log(message); // "10 items cost $2.50."
```

## < Hoisting? />



```
x = 1;

alert('x = ' + x);

var x;
```

# Let *and* Const Block Scoping

## Block Level Declaration with let and const

Variable declarations using **var** are treated as if they're at the top of the function (or in the global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called **hoisting**.

Misunderstanding hoisting unique behaviour can end up causing bugs. For this reason, ECMAScript 6 introduces **block-level** scoping options to give developers more control over a variable's life cycle.

Block  Scopes are created in the following places:

1-Inside a function

2- Inside a block (indicated by the { and } characters)

# Let Declaration

The let declaration syntax is the same as the syntax for var. You can basically replace var with let to declare a variable but limit the variable's scope to only the current code block

```
 let studentId = 2;
console.log(studentId); // output  2
```

let declarations are not hoisted to the top of the enclosing block, it's best to place let declarations first in the block so they're available to the entire block.

```
console.log(studentId); //error: studentId is not defined
let studentId = 2;
```

By using let you **can not** define parameter twice

```
 let count = 2;
 var count=2; //error : Identifier 'count' has already been declared
 //or
 var count=2;
 let count = 2; //error : Identifier 'count' has already been declared
```

Because let will not redefine an identifier that already exists in the same scope, the let declaration will throw an error.

```
var count = 30;
if (true) {
           let count = 40; // doesn't throw an error
          }
```

## Const Declaration

constants, meaning their values **cannot be** changed once set. For this reason, every const variable must be initialized on declaration

```
// valid constant
const maxItems = 30;
maxItems=40; //error: Assignment to constant variable.
const name; // syntax error: Missing initializer in const declaration
```

**Const variables are not hoisted**

```
console.log(maxItems); //error : maxItems is not defined
const maxItems = 30;
```

✓Constants, like let declarations, are block-level declarations.

```
const maxItems = 30;
if (true) {
               const maxItems = 5;
               // more code
          }
```

✓In another similarity to let, a const declaration throws an error when made with an identifier for an already defined variable in the same scope.

```
var message = "Hello!";
let age = 25;
// each of these throws an error
const message = "Goodbye!";
const age = 30;
```

✓Even if we start defining variable with const

```
const age = 30;
let age=30;// error : Identifier 'age' has already been declared
```

## Operators

## Arithmetic Operators

Arithmetic operators are used to perform arithmetic between variables and/or values.
Given that y = 5, the table below explains the arithmetic operators:

| Operator | Description | Example | Result in y | Result in x |
|---|---|---|---|---|
| + | Addition | x = y + 2 | y = 5 | x = 7 |
| - | Subtraction | x = y - 2 | y = 5 | x = 3 |
| * | Multiplication | x = y * 2 | y = 5 | x = 10 |
| / | Division | x = y / 2 | y = 5 | x = 2.5 |
| % | Modulus (division remainder) | x = y % 2 | y = 5 | x = 1 |
| ++ | Increment | x = ++y | y = 6 | x = 6 |
| | | x = y++ | y = 6 | x = 5 |
| -- | Decrement | x = --y | y = 4 | x = 4 |
| | | x = y-- | y = 4 | x = 5 |

# Assignment operators

| Operator | Example | Equivalent |
|---|---|---|
| += | a += b | a = a + b |
| -= | a -= b | a = a – b |
| *= | a *= b | a = a * b |
| /= | a /= b | a = a / b |
| %= | a %= b | a = a % b |
| <<= | a <<= b | a = a << b |
| >>= | a >>= b | a = a >> b |
| >>>= | a >>>= b | a = a >>> b |
| &= | a & b | a = a & b |
| \|= | a \|= b | a = a \| b |
| ^= | a ^ b | a = a ^ b |

## Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or values.
Given that x = 5, the table below explains the comparison operators:

| Operator | Description | Comparing | Returns |
|---|---|---|---|
| == | equal to | x == 8 | false |
| | | x == 5 | true |
| === | equal value and equal type | x === "5" | false |
| | | x === 5 | true |
| != | not equal | x != 8 | true |
| !== | not equal value or not equal type | x !== "5" | true |
| | | x !== 5 | false |
| > | greater than | x > 8 | false |
| < | less than | x < 8 | true |
| >= | greater than or equal to | x >= 8 | false |
| <= | less than or equal to | x <= 8 | *true* |

## Logical Operators

Logical operators are used to determine the logic between variables or values.
Given that x = 6 and y = 3, the table below explains the logical operators:

| Operator | Description | Example |
|---|---|---|
| && | and | (x < 10 && y > 1) is true |
| \|\| | or | (x === 5 \|\| y === 5) is false |
| ! | not | !(x === y) is true |

# Bitwise Operators

Bit operators work on 32 bits numbers. Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

| Operator | Description | Example | Same as | Result | Decimal |
|---|---|---|---|---|---|
| & | AND | x = 5 & 1 | 0101 & 0001 | 0001 | 1 |
| \| | OR | x = 5 \| 1 | 0101 \| 0001 | 0101 | 5 |
| ~ | NOT | x = ~ 5 | ~0101 | 1010 | 10 |
| ^ | XOR | x = 5 ^ 1 | 0101 ^ 0001 | 0100 | 4 |
| << | Left shift | x = 5 << 1 | 0101 << 1 | 1010 | 10 |
| >> | Right shift | x = 5 >> 1 | 0101 >> 1 | 0010 | 2 |

## The Conditional Operator (?:)

The conditional operator is the only ternary operator (three operands) in JavaScript and is sometimes actually called the ternary operator.

**Syntax:** *condition ? expr1 : expr2*

```
var fee = (isMember ? "$2.00" : "$10.00")

==

if(isMember)

    fee = "$2.00";

else

    fee = "$10.00";
```

# Conditionals

## if

The fundamental control statement that allows JavaScript to make **decisions**, or, more precisely, to **execute statements conditionally**. This statement has two forms.

**if (*expression*)**
  ***statement***

```
if (username == null)     If username is null or undefined,
    username = "guest";   define it


if (!address) {
    address = "";
    message = "Please specify a mailing address.";
}
```

## if

The second form of the if statement introduces an else clause that is executed when *expression* is false.

**if (*expression*)**
    ***statement1***
**else**
    ***statement2***

```
if (hour < 18) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

## else if

```
if (n == 1) {
     Execute code block #1
}
else if (n == 2) {
     Execute code block #2
}
else if (n == 3) {
     Execute code block #3
}
else {
     If all else fails, execute block #4
}
```

## switch

An if statement causes a branch in the flow of a program's execution, and you can use the else if idiom to perform a **multiway branch**. The switch statement handles exactly this situation.

```
switch(n) {
    case 1:  Start here if n == 1
         Execute code block #1.
        break; Stop here
    case 2:  Start here if n == 2
         Execute code block #2.
        break;  Stop here
    default:  If all else fails...
         Execute code block #4.
        break;  stop here
}
```

# switch

Use the weekday number to get weekday name:

```
switch (new Date().getDay()) {
    case 0:
        day = "Sunday";
        break;
    case 1:
        day = "Monday";
        break;
    case 2:
        day = "Tuesday";
        break;
    case 3:
        day = "Wednesday";
        break;
    case 4:
        day = "Thursday";
        break;
    case 5:
        day = "Friday";
        break;
    case 6:
        day = "Saturday";
}
```

## Loops

## while

The **while statement** creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

```
var count = 0;
while (count < 10) {
    console.log(count);
    count++;
}
```

## do/while

The **do...while statement** creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

```
var i = 0;
do {
    i += 1;
    console.log(i);
} while (i < 5);
```

## for

The for statement provides a looping construct that is often more convenient than the while statement. The for statement simplifies loops that follow a common pattern. Most loops have a **counter variable** of some kind. This variable is **initialized** before the loop starts and is **tested before each iteration** of the loop. Finally, the counter variable is **incremented** or otherwise updated at the end of the loop body.

```
for(var count = 0; count < 10; count++)
        console.log(count);

var i,j;
for(i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

# for/in

A for...in loop only iterates over enumerable properties.

```
for(var i = 0; i < a.length; i++)          Assign array indexes to variable i
    console.log(a[i]);                     Print the value of each array element
for(i in a) console.log(i);
```

## break

The break statement, used alone, causes the innermost enclosing loop or switch statement to exit immediately.

```
for(var i = 0; i < a.length; i++) {
    if (a[i] == target) break;
}
```

# continue

The continue statement is similar to the break statement. Instead of exiting a loop, however, continue restarts a loop at the next iteration.

```
for(i = 0; i < data.length; i++) {
    if (!data[i]) continue;       Can't proceed with undefined data
    total += data[i];
}
```