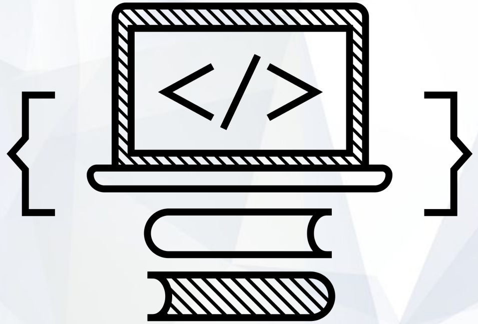


JS Fundamentals

Course: Java

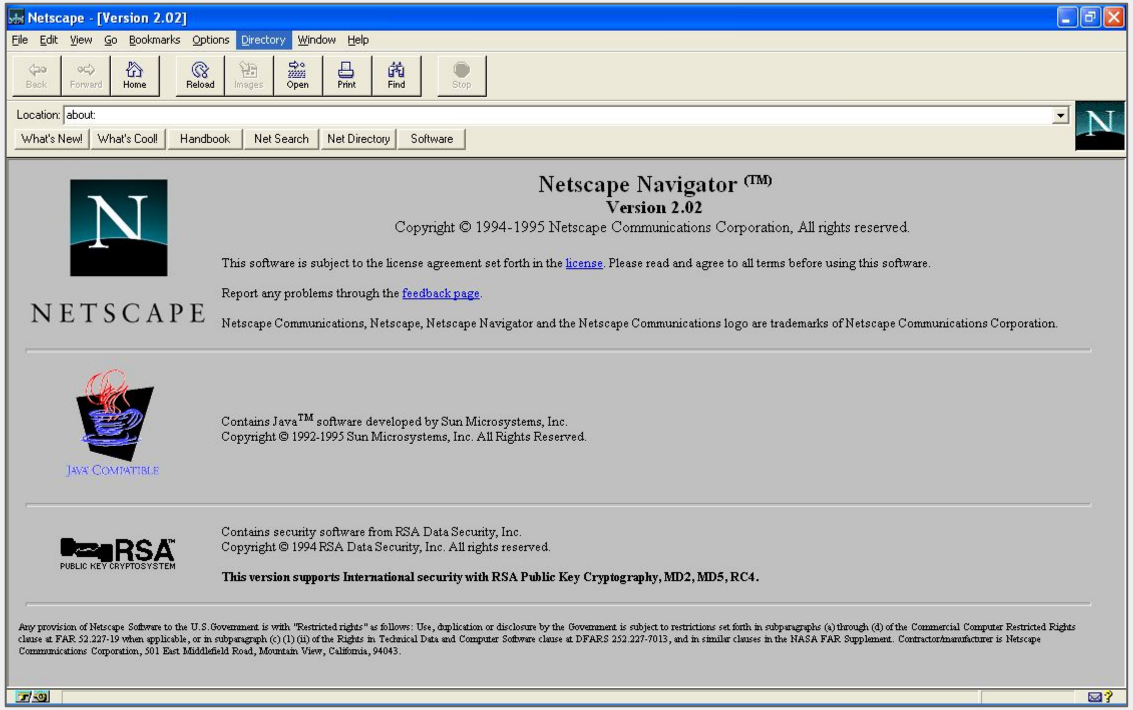
S1

The History of JavaScript

Before Brendan Eich created JavaScript (JS) in 1995, all websites were essentially just static pages—HTML/CSS with some links and images.

It was very difficult to create a dynamic application.

In order for anything dynamic to happen, there had to be a separate request to a server and a subsequent re-render of an entire site.


Netscape - [Version 2.02]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Location: about:
What's New! What's Cool! Handbook Net Search Net Directory Software
NETSCAPE
JAVA COMPATIBLE
RSA PUBLIC KEY CRYPTOSYSTEM

Netscape Navigator (TM)
Version 2.02
Copyright © 1994-1995 Netscape Communications Corporation, All rights reserved.

This software is subject to the license agreement set forth in the license. Please read and agree to all terms before using this software.

Report any problems through the feedback page.

Netscape Communications, Netscape, Netscape Navigator and the Netscape Communications logo are trademarks of Netscape Communications Corporation.

Contains Java TM software developed by Sun Microsystems, Inc.
Copyright © 1992-1995 Sun Microsystems, Inc. All Rights Reserved.

Contains security software from RSA Data Security, Inc.
Copyright © 1994 RSA Data Security, Inc. All rights reserved.

This version supports International security with RSA Public Key Cryptography, MD2, MD5, RC4.

Any provision of Netscape Software to the U.S.Government is with "Restricted rights" as follows: Use, duplication or disclosure by the Government is subject to restrictions set forth in subparagraphs (a) through (d) of the Commercial Computer Restricted Rights clause at FAR 52.227-19 when applicable, or in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, and in similar clauses in the NASA FAR Supplement. Contractor/manufacturer is Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, California, 94043.

The History of JavaScript

Then, along came JS (originally called LiveScript), and before long it was included in every browser. There was no need for extra add-ons, as was the case for Java applets and, later, Flash.

May 1995

Brendan Eich invents Mocha

September 1995

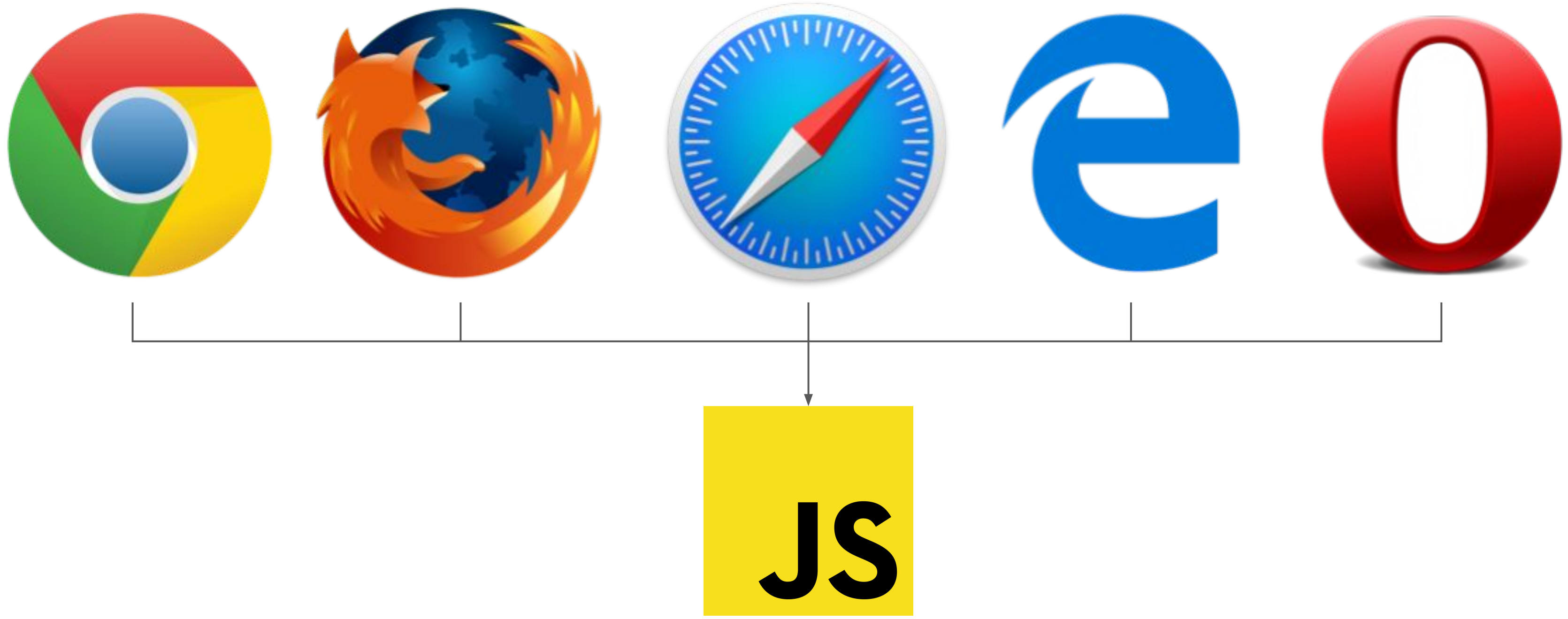
Mocha renamed LiveScript

December 1995

LiveScript renamed JavaScript

The History of JavaScript

JS was created specifically to make websites dynamic, and it now ships with all major browsers. JS rules the web (or at least, the front end).



Learning Outcomes

By the end of this lesson, you will be able to:

01 Use the Dev Tools Console (REPL).

02 Use dynamic data types—primitives (`typeof`).

03 Declare variables with `const` and `let`.

04 Use arrays and objects.

05 Use operators and operands.

06 Use conditional logic and ternary.

07 Use string concatenation and template literals.

08 Use `switch` - `case`.

09 Use `for` and `while`.

JavaScript

Modern JavaScript and Googling

Modern JavaScript and Googling

Version	Year	Official Name
ES1	1997	ECMAScript 1
ES2	1998	ECMAScript 2
ES3	1999	ECMAScript 3
ES4	never released	ECMAScript 4
ES5	2009	ECMAScript 5
ES6+	2015	ECMAScript 2015
	2016	ECMAScript 2016
	2017	ECMAScript 2017
	2018	ECMAScript 2018

- In 2015, there was a big overhaul of the language—the biggest of its history.
- That version is known both as ES2015 and ES6.
- **EcmaScript** is the more technical name for JavaScript, but the terms are used interchangeably.
- Since 2015, there have been a few annual updates (ES7, ES8, etc.). But, **ES6+** refers to modern JavaScript.

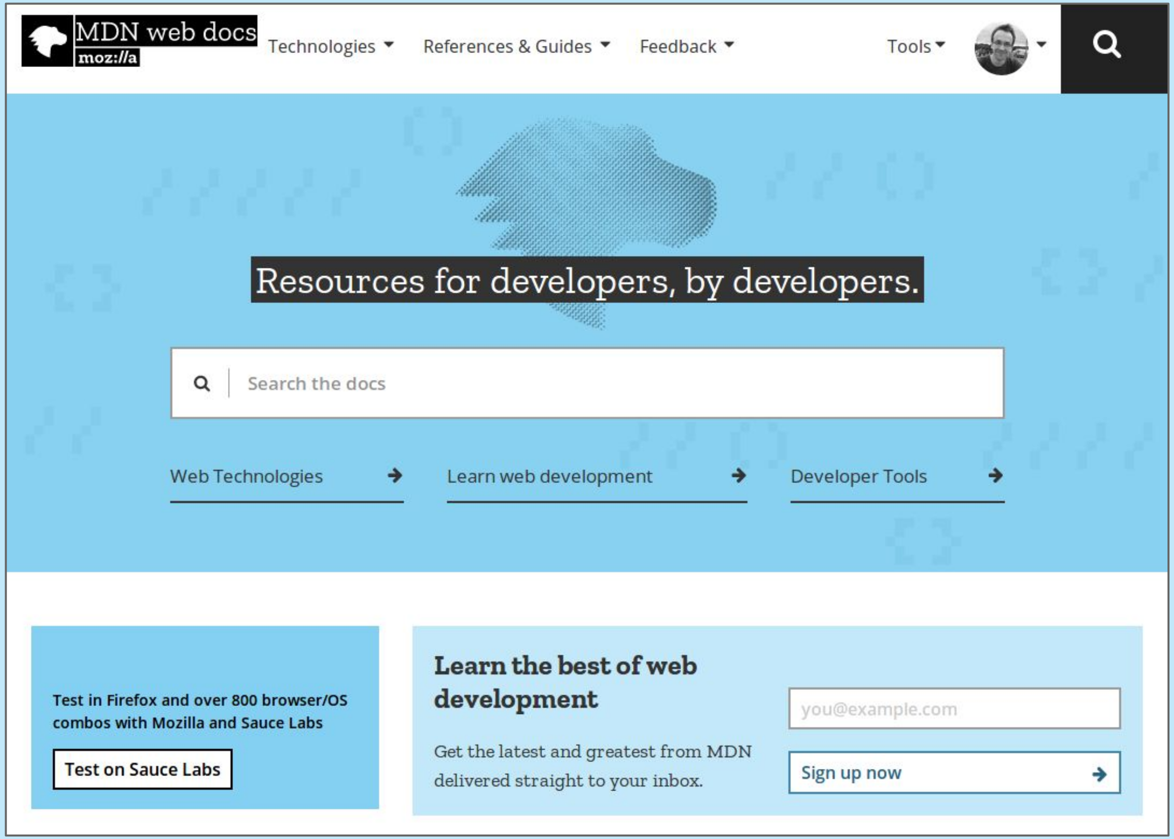
In this lesson, we'll lead with the fundamentals, but we'll quickly transition to introducing some of the most modern syntax.

Modern JavaScript and Googling

There is a lot of information out there about JS—some is good and some is bad.

Generally, we should check with MDN first.

In addition to reading the spec directly, this is usually a reliable source of the most modern syntax and best practices.



Resources for developers, by developers.

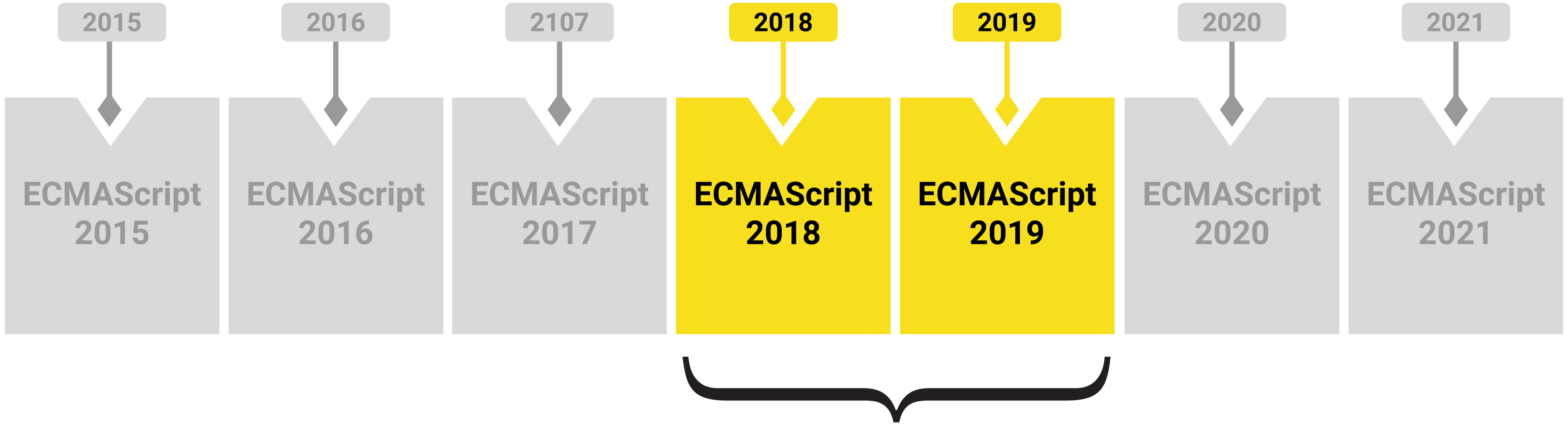
Test in Firefox and over 800 browser/OS combos with Mozilla and Sauce Labs

Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

Modern JavaScript and Googling

The most current version of JS is ES2021. However, we don’t typically use the most modern version, because it takes two to three years before a majority of browsers adopt all of the new syntax.



Therefore, ES8 or ES9 will suffice.

Modern JavaScript and Googling

There is nothing to install, set up, or configure. All major browsers have a JS engine that allows us to jump right in and use JS.

Blink	Gecko	Webkit	Trident	Blink
V8	SpiderMonkey	Nitro	Chakra	V8

Modern JavaScript and Googling

Use terms like "MDN" and "EcmaScript" for more reliable Google results.

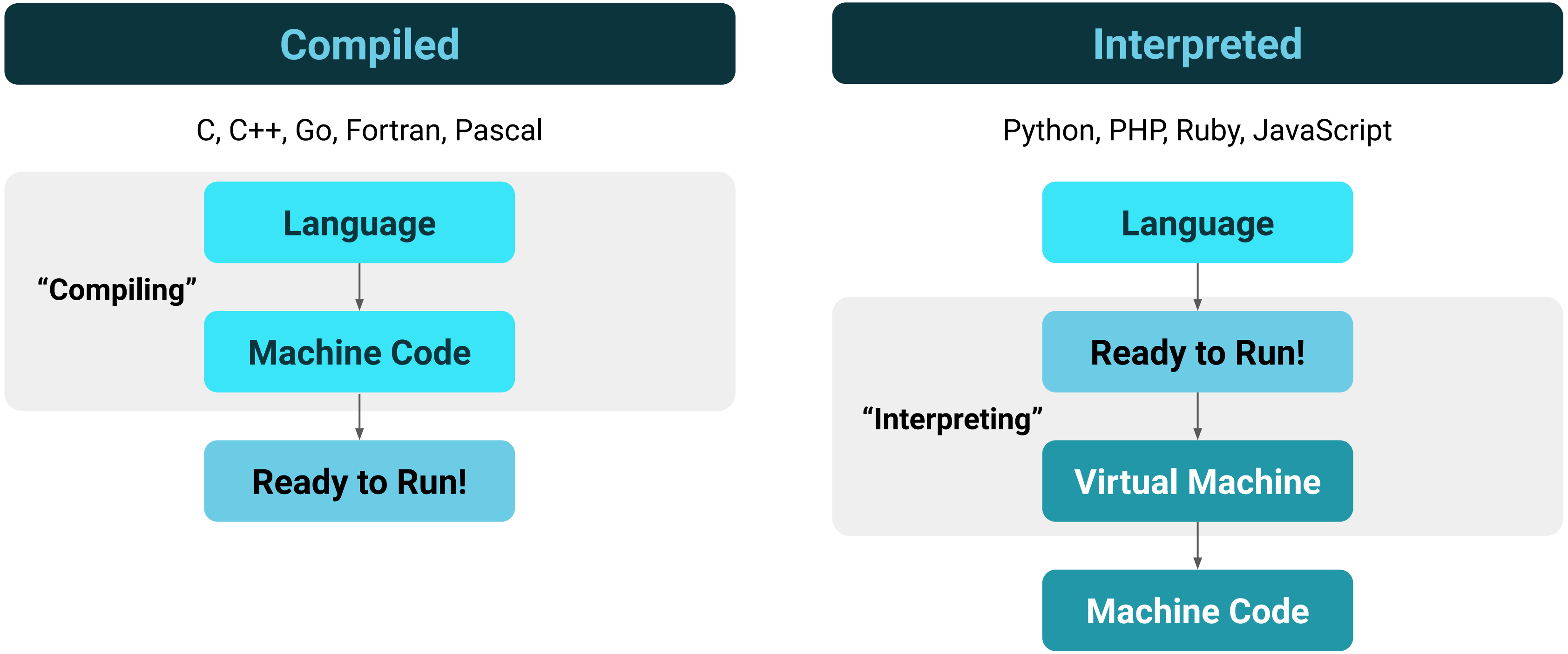
Be wary of any code examples that are more than three to four years old, and prefer resources that have been written more recently.

JS Fundamentals

What does it mean that JS is generally an interpreted language, as opposed to a compiled language like Java?

JS Fundamentals

In JS, we do not proactively declare the type of our data. We just create it, and JS interprets as it goes. Therefore, JS is an interpreted language (not compiled, per se).



What are JS’s primitive data types?

JS’s Primitive Data Types

`string`	`number`	`boolean`	`undefined`	`null`
Use single or double quotation marks, but be consistent. You can also use backticks, but usually just for **interpolation**. ● `"Your Name"` ● `'Your Name'` ● The following is **invalid**, because the quotation marks are not consistent: `'Your Name"`	Don't specify anything in particular (e.g., `float`, `int`, etc. as in Java). **No quotation marks.** ● `12` ● `8675309` ● `2.99` ● `-999`	Virtually identical to Java—but don't specify `boolean`. **No quotation marks.** ● `true` ● `false`	The absence of any value. This is synonymous with `null` in Java. We generally don't use this value deliberately. Instead, we allow the JS interpreter to `log` this for us so that we know a usable value is absent.	A deliberate nothing value. Deliberately assign this to a variable if you want to set it as nothing (as opposed to the absence of any value like `undefined`.) ● `let x = null;`

How do we bind data to variables?

JS Fundamentals: const

Use `const` by default. This will prevent unintended reassignment of a variable.

```
const bradyNumber = 12
```

```
const jordanNumber = 23
```

```
const greatestOfAllTime = "Tom Brady"
```

JS Fundamentals: let

Use `let` if you intend to re-assign a value at some point.

If you're not sure, use `const` by default.

If there is an error, `Assignment to constant variable`, you can decide whether this was intentional, in which case you can use `let`.

JS Fundamentals: let

`let` also introduces the **dynamically typed** nature of JS. You can re-assign values whimsically, unlike in Java, which is **statically typed**.

```
let name = "Ferdinand Lewis Alcindor Jr"
```

```
name = "Kareem Abdul-Jabbar"
```

It is a common mistake to use `let` for the same variable, instead of just re-assigning as shown above.

```
let name = "Cassius Marcellus Clay Jr"
// This is an error - "SyntaxError: Identifier 'name' has already been declared"
let name = "Muhammad Ali"
```

JS Fundamentals: var

`var` has some specific nuances that can lead to unintended behaviors. `const` and `let` were introduced in ES6 for this reason.

Is `undefined` related to the error message `not defined`?

JS Fundamentals: undefined

No! The primitive data type `undefined` will occur if we use `let` but don't assign a value. Referencing a variable that has not been declared will result in a `ReferenceError`:

```
/**
 * We have never mentioned `z` before.
 * JS will look in memory for anything labeled as `z`.
 * This will result in: `ReferenceError: z is not defined`
 */
console.log(z);
```

```
let z; // z is 'undefined' the absence of any deliberately assigned value;
```

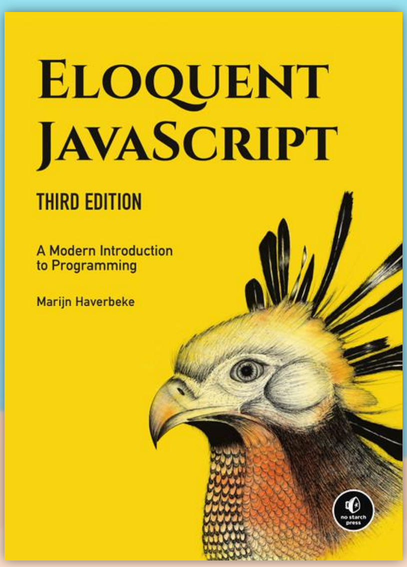
JS Fundamentals: null

And, remember the difference between `undefined` and `null`:

```
/**
 * We have chosen to deliberately assign nothing to 'z'.
 * This is especially worthless as `z` cannot be reassigned b/c of `const`.
 * It is nothing and will always be nothing.
 */
const z = null;
```

Eloquent JavaScript: Analogy

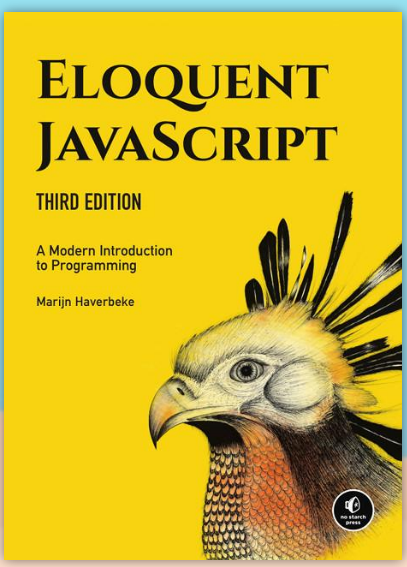
- The computer's memory is a vast ocean.
- Data just floats around in this ocean as we create it.
- There is a sea monster.
- When we use `const` or `let`, this sea monster grows a tentacle.
- Whatever name we use is tattooed onto the tentacle to give it a name/reference.
- This tentacle reaches out and grabs some of the floating data when we use the assignment operator `=`.



Eloquent JavaScript, 3rd Edition
by Marijn Haverbeke
(No Starch Press, 2018)

Eloquent JavaScript: Analogy (continued)

- If we assign `null`, the tentacle grabs only water but wraps closed—the deliberate assignment of nothing.
- If we say `let x`, the tentacle grows and is tattooed with `x` but doesn't grab anything at all. The tentacle is open.
- Any `let` tentacle can let go of the data that it is holding and grab something else.
- We cannot have two tentacles with the same tattoo (re-declaring `let` with the same variable).
- A `const` tentacle cannot release the data it is holding.



Eloquent JavaScript, 3rd Edition
by Marijn Haverbeke
(No Starch Press, 2018)

What are some of JS’s operators?

JS Fundamentals: Operators

Because of its dynamically typed nature, JS performs many implicit conversions in response to certain **operators**. We will convert some of these separately.

`+`	Concatenation Operator	Adds numbers or concatenates strings.
`-`	Minus Operator	Subtracts numbers. Using it with other data types results in `NaN` (still considered a `number` data type).
`*`	Multiplication Operator	Multiplies numbers.
`/`	Division Operator	Divides numbers.
`%`	Remainder Operator	Yields the remainder between two numbers.
`()`	Order of Operations	Similar to Java and most languages. When in doubt, use `()` to group things together.

JS Fundamentals: Incremental and Assignment Operators

Incremental Operators

These are the same as Java.

`++` `--`

Assignment Operators

These are mostly the same as Java.

`=` `*=`
`+=` `/=`
`-=`

There are additional assignment operators, such as bitwise, but they are rarely used.

JS Fundamentals: Relational Operators

Compare values to each other and evaluate to `boolean`s. These are commonly used with `if` statements.

`<`	Less than.
`>`	Greater than.
`<=`	Less than or equal to.
`>=`	Greater than or equal to.
`===`	Equal value **and** data type. (There is also `==`, but it has little benefit. To avoid confusion, stick with `===`.)
`!==`	Not equal value **or** not same data type. (Use this instead of `!=`.)

JS Fundamentals: Logical Operators

&&

||

!

Same as Java—
nothing new here!

What does the `typeof` unary operator do?

JS Fundamentals: typeof Unary Operator

It evaluates to the type of the **operand**. This always outputs a string.

```
typeof "Mark" // string
```

```
typeof 23 // number
```

```
typeof true // boolean
```

```
typeof typeof 23 // string
```

```
typeof NaN // number
```

How does JS’s implicit conversion of data work in relation to operators?

You don’t need to memorize this seemingly strange behavior in JS.

JS Fundamentals: Implicit Conversion

- If there is a string present with `+`, JS will coerce all operands into strings.
  - `23 + "Michael"` // "23Michael"`
  - `true + 'Jordan'` // 'trueJordan'`
  - If we use + with a boolean and a number, `true` is coerced to `1` and `false` is coerced to `0`
    - `true + 5 // 6`

- `-` will implicitly coerce to numbers or NaN if the coercion fails
  - `23 - "Michael"` // NaN`
  - `true - 5; // -4`
  - We can do `- 0` to convert strings to numbers when needed: `"23" - 0; // 23`

- All data types are **truthy** or **falsy**
  - `""` and `0` are both considered falsy: `true && ""; // false`
  - Any non-empty string or non-zero number are considered truthy: `false || 3; // true`

JS Fundamentals: Short-Circuiting

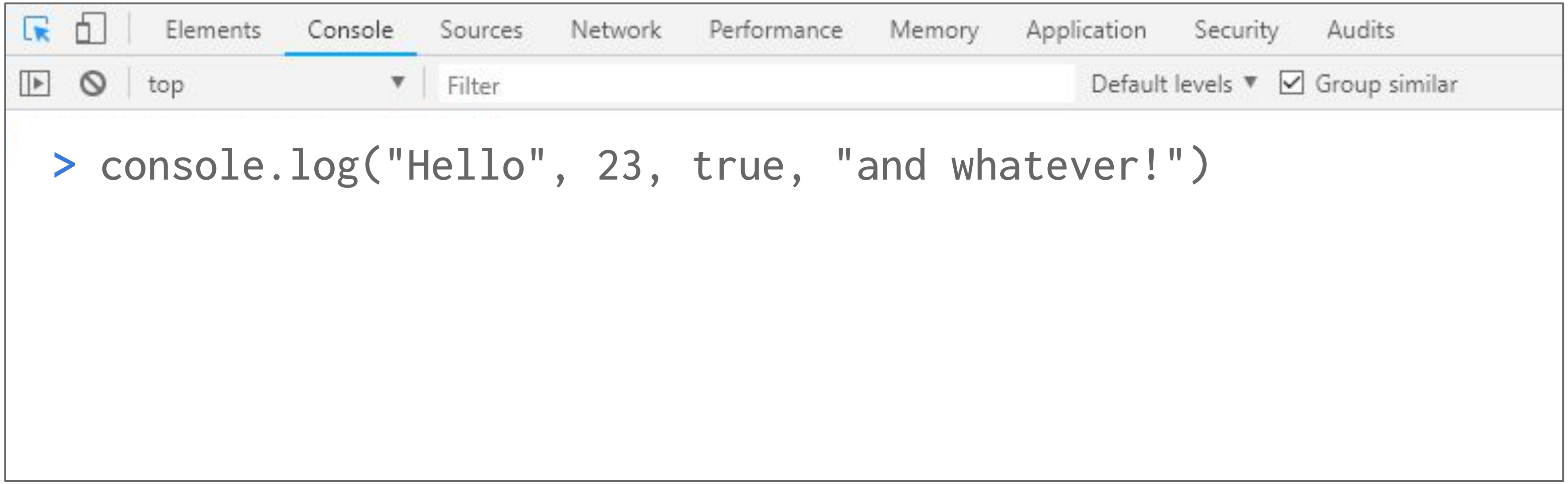
When working with `||`, if the left-hand side operand is `true`, JS will use it and not go to the right-hand side operand. We can use this in combination with falsy and truthy values:

```
const name = "Mark";

// If `name` is falsy, we will log our error
console.log(name || "Error! Name is blank!")
```

How do we print in JS?

JS Fundamentals: Printing

`console.log()` much data as we want separated by `,`:



```
> console.log("Hello", 23, true, "and whatever!")
```

How do we write `if` statements?

JS Fundamentals: if Statements

This is very similar to Java:

```
const x = 18;
if (x > 0) {
  console.log("X is greater than 0")
} else if (x < 0) {
  console.log("X is less than 0")
} else {
  console.log("X must be 0")
}
```

JS Fundamentals: Implicit Conversion

We can take advantage of implicit conversion instead of doing comparisons all the time. Whatever expression is kept inside of `()` will be implicitly coerced to `true` or `false`. This can save some typing:

```
const name = "Mark";

// Check if name is not empty, else `log` an error
if (name) {
  console.log("Name is ok. JS implicitly coerced it and found that it was truthy. There is no need to check a `length`.")
} else {
  // `console.error` and `console.log` are pretty much the same - some JS engines might display `error` in red.
  console.error("Name is empty. This will be seen if `name` is an empty string. It will be implicitly coerced to `false`. We say that an empty string is falsy.)
}
```

How do we write a ternary?

JS Fundamentals: Ternary

Again, this is very similar to Java:

```
const passingScore = 70;

const myScore = 68;

console.log(myScore >= passingScore ? "You passed!" : "You failed!");
```

What about `switch`?

JS Fundamentals: switch

Again, this is very similar to Java:

```
const action = "Get all data!"

switch(action) {
  case "Get all data!":
    console.log("Do some stuff to reach out to some database")
    break;
  case "Delete all data!":
    console.log("Um...are you sure?");
    break;
  default:
  console.log("Unrecognized action received!");
}
```

What about `for` and `while` loops?

JS Fundamentals: for and while

These are almost identical to Java. But we would use `let` for our iterator variable:

```
let i = 100;

while(i > 0) {
    console.log(i);
    i--
}
```

```
for (let i = 0; i < 100; i++) {
   console.log(i)
}
```

Break

JS Activities

Time to Code

Set Up Starter Code

Suggested Time:

10 Minutes

Activity: JS Playground

In this activity, you will use `console.log()` and observe the output in the Dev Tools “Console” tab.

Suggested Time:

20 Minutes

Activity: Fizz Buzz

In this activity, you will solve the classic Fizz Buzz problem that was used in coding interviews.

Suggested Time:

30 Minutes

Time's Up! Let's Review.

Questions?

Break

Activity: Simple Combo Menu

Suggested Time:

15 Minutes

Time's Up! Let's Review.

Questions?

Template Literals and Interpolation

As previously mentioned,
we can use backticks instead of
quotation marks for strings.

Template Literals and Interpolation

We'll start with an example that **does not** use template literals and backticks:

```
const name = "Mark";

// Notice how inconvenient it is to open and close the quotations and keep
the spacing correct with the `+`
const greeting = "Hello, my name is: " + name + "!";
```

Template Literals and Interpolation

Now, let’s do the same thing with modern syntax, using **backticks** and **interpolation**:

```
const name = "Mark";

/**
 * Whenever we want to interpolate the value of some expression or variable,
 * we just use `${}`.
 *
 * This will only work when used in conjunction with backticks.
 */
const greeting = `Hello, my name is ${name}!`
```

Activity: Update the switch-case Activity to Use Template Literals and Interpolation

Suggested Time:

15 Minutes

Time’s Up! Let’s Review.

Questions?

Collection/Composite Data Types

Arrays

Arrays

JS arrays, like other JS data types, are extremely flexible. Unlike Java, we don't need to specify what type of data will be in our array. We can mix and match as we wish.

We can access values via a **zero-based numerical index**, as in Java.

```
const exampleArray = ["Mix and match the data!", 23, true, ["even another array - but that's not too common"]]
console.log(exampleArray[1]); // 23
```

Arrays are zero-indexed. The last index of any array is at `length - 1`.

```
const animals = ["dog", "cat", "horse", "bird", "sheep", "goat"];
console.log(animals[animals.length - 1]);
```

Arrays

Accessing values beyond an array's `length` **does not** give an error (e.g., "Out of Bounds"). It's just `undefined`.

```
const animals = ["dog", "cat", "horse", "bird", "sheep", "goat"];

// This DOES NOT affect the length of the array.
console.log(animals[999]); // undefined

// Setting a new value means that the array's length is now 1000

animals[999] = "Lion";

// There will be many `undefined`s in here
console.log(animals);
```

Array Mutation

Array Mutation

Even if we use `const`, we can **mutate** our array.

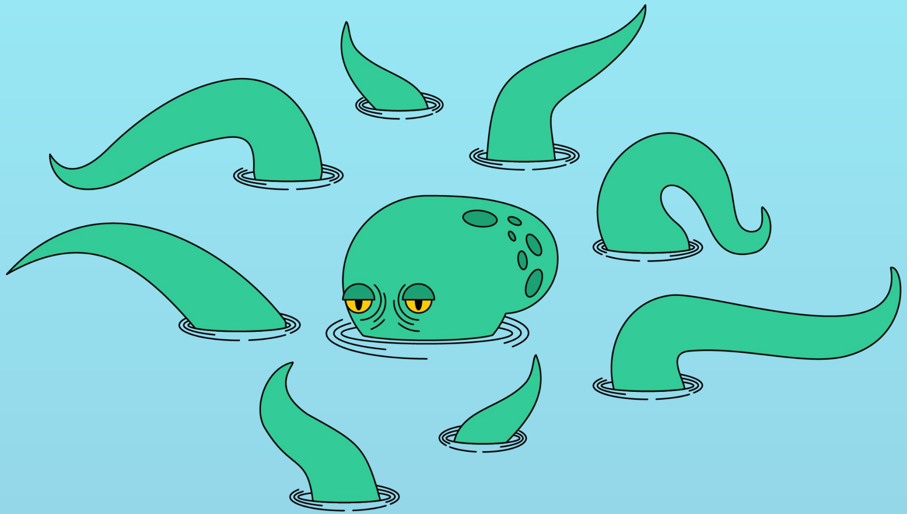
```
const greetings = ["Hello", "Goodbye", "Good Morning"]

// Replace "Good Morning" with "Good Night!"]

greetings[2] = "Good Night!"
```

Using `const` means that we **cannot re-assign a new value**. However, when working with collections such as arrays, we can still mutate them.

Array Mutation

Let's revisit our sea monster analogy from earlier in the lesson:

- Our sea monster is tired of growing so many tentacles.
- He starts using traps to gather up several pieces of data and keep them stored with just one tentacle.
- He cannot let go of the trap (assuming `const`).
- But, he can add and remove the data that is inside of the trap.



Break

Array Methods

Time to Code

Use a while Loop to Iterate Over an Array

Suggested Time:

5 Minutes

Array prototype Methods

In addition to the `length` property (as in Java), numerous methods are available for any given array, many of which work similarly to the Java equivalent.
A few of the simpler ones include:

`push`	Mutates an array by adding an element to the end of it. This will give back the new length of the array.
`pop`	Mutates an array by removing the last element. This gives back the element that was removed.
`shift`	Mutates an array by removing the first element and giving that back.
`unshift`	Like `push`, but adds the element to the beginning of the array.
`reverse`	Reverses the entire array.

`slice`	Takes in a starting index (inclusive) and ending index, and gives back just that portion of the array. The original array is **not mutated**.
`concat`	Similar to `push`, but we can avoid a mutation by re-assigning the result to a new array, thereby leaving the original intact.
`includes`	Checks for the inclusion of a specific element and gives back `true` or `false`.
`indexOf`	Gives back the “index of” where a specific element is found in an array, or `-1` if the element doesn't appear anywhere in the array.
`lastIndexOf`	Same as `indexOf`, but the search starts from the end of the array.

Time to Code

Try Some Array Methods

Suggested Time:

15 Minutes

Time to Code

Destructure Arrays

Suggested Time:

10 Minutes

Time’s Up! Let’s Review.

Questions?

Break

Time to Code

Spread Arrays

Suggested Time:

15 Minutes

Activity: Array Methods Playground

Suggested Time:

20 Minutes

Time to Code

Convert Arrays Into Strings and Vice Versa

Suggested Time:

10 Minutes

Time’s Up! Let’s Review.

Questions?

Break

Arrays and Objects

Time to Code

Convert Strings Into Arrays

Suggested Time:

10 Minutes

Activity: Extract Long Words from Some Text

Suggested Time:

10 Minutes

Time's Up! Let's Review.

Questions?

Object Literals

Object Literals

Everything in JS, except for primitive data, is an object.
(We will cover this more when we discuss OOP in JS.)

Today, we're discussing objects that we create ourselves, or, **object literals**.
(We use the terms interchangeably.)

Here's how to create an object in JS:

```
const someObject = {}
```

That's it! That's a JS object!

Time to Code

Object Examples

Suggested Time:

20 Minutes

Time’s Up! Let’s Review.

Questions?

Congratulations!

You've learned everything you need to know about data structures in JS.

Break

Nested Objects

Nested Objects

```
const employee = {
  name: "Horace Grant",
  company: {
    name: "Chicago Bulls"
  }
}

// Chain dot notation
console.log(employee.company.name);

/**
 * What if we use chaining on a property that doesn't exist? TYPE ERROR!
 * 'Cannot read property `name` of undefined'
 *
 * `employee.otherCompany` is `undefined` - that's OK.
 * `undefined.name` - NO! Cannot access a property on `undefined`
 * `undefined` cannot have properties.
 */
console.log(employee.otherCompany.name);
```

Activity: Objects Playground

Suggested Time:

30 Minutes

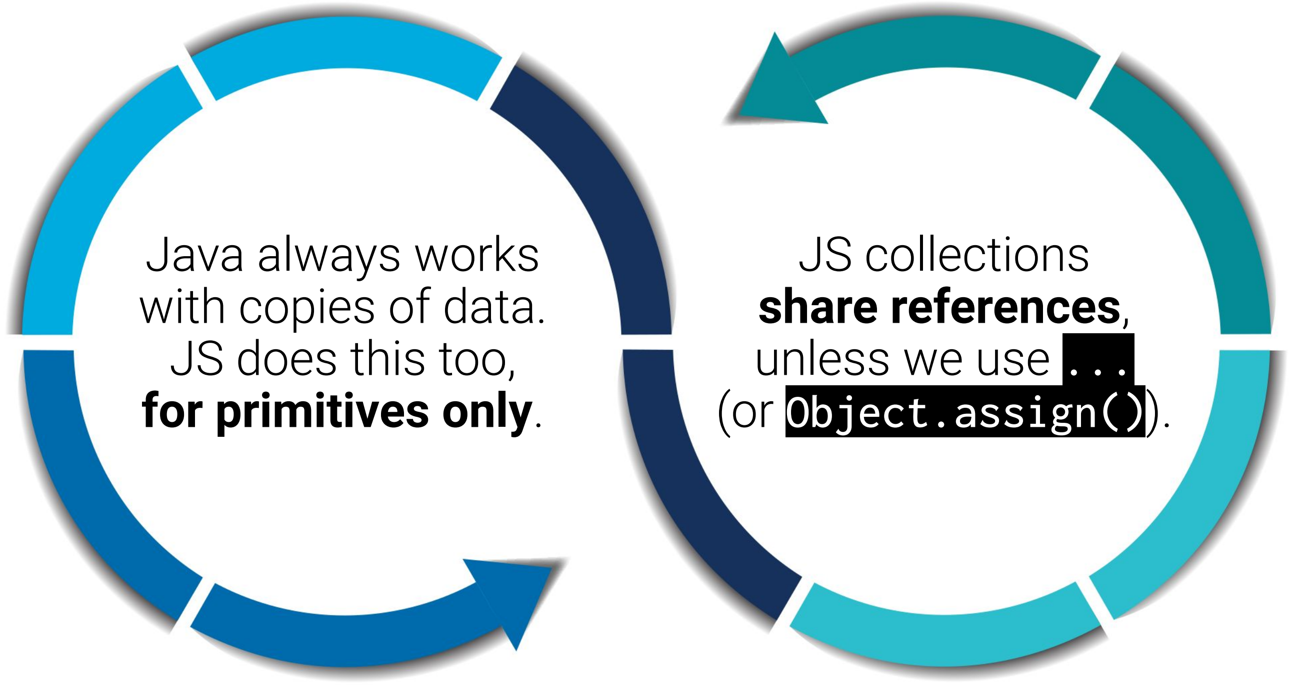
Time to Code

Copy/Values vs. Reference

Suggested Time:

10 Minutes

Copy/Values vs. Reference



Java always works with copies of data. JS does this too, **for primitives only**.

JS collections **share references**, unless we use `...` (or `Object.assign()`).

Time to Code

Spread Syntax … Doesn’t Prevent Mutations with Nested Collection

Suggested Time:

10 Minutes

Remember…
all collections, including arrays,
also share references.

Time’s Up! Let’s Review.

Questions?

RECAP

Learning Outcomes

By the end of this lesson, you will be able to:

01	Use the Dev Tools Console (REPL).
02	Use dynamic data types—primitives (`typeof`).
03	Declare variables with `const` and `let`.
04	Use arrays and objects.
05	Use operators and operands.
06	Use conditional logic and ternary.
07	Use string concatenation and template literals.
08	Use `switch` - `case`.
09	Use `for` and `while`.

The End