



# JavaScript Get Started

---

# What Is JavaScript?



# What Is JavaScript?

- JS is a text-based programming language used both on the client-side and server-side that allows you to make web pages interactive.
- JS is an implementation of the ECMAScript standard.
- JS was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.



# Name

- ECMAScript is the official name of the language
- Back in the early 2000s Microsoft alternative JScript
- From 2016 ECMAScript is named by year (ECMAScript 2016)
- ECMAScript is often abbreviated to ES
- JavaScript, JS, ECMAScript, or ES2019
- [https://www.w3schools.com/js/js\\_versions.asp](https://www.w3schools.com/js/js_versions.asp)



# JavaScript the multi-paradigm language

- Procedural
- Object-oriented (OO/classes)
- Functional (FP)



# Backwards & Forwards compatibility

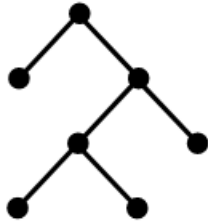
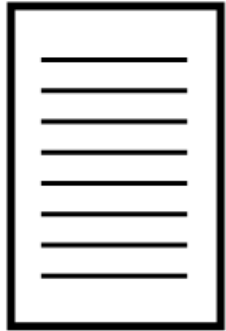
- JS is **backwards compatible** - there will not be a future change to the language that causes that code to become invalid JS
- JS is **not forwards compatible** - Being forwards-compatible means that including a new addition to the language in a program would not cause that program to break if it were run in an older JS engine
- For new and incompatible syntax, the solution is **transpiling**.



# JS is interpreted or compiled?

- JS is a compiled language, meaning the tools (including the JS engine) process and verify a program (reporting any errors!) before it executes.

# JS is interpreted or compiled?



```
0100110101001010  
1100010101011101  
0111010101010110  
1001101010101000  
1001010101011011  
0101010100101110  
1010101110000011  
1001010101011001  
0000100100001010  
1001010101010101  
...
```





# Surveying JS

syntax, patterns, and behaviors at a high level.





## Each File is a Program

- One file may fail and that will not necessarily prevent the next file from being processed.
- ES6, modules are also file-based.
- JS files act as a single program by sharing their state via the "**global scope.**"



# Values

## Primitives

- Strings
  - double-quote "
  - single-quotes '
  - back-tick ` (interpolation)
- numbers
- booleans
- null
- undefined
- symbol

## Objects

- **Objects** are more general: an unordered, keyed collection of any various values.
- **Arrays** are a special type of object that's comprised of an ordered and numerically indexed list of data
- **Functions**, like arrays, are a special kind (aka, sub-type) of object.



# Values

```
<script>
  var firstName = "Christian";
  var lastName = "Bale";
  var fullName = `Christian Bale`;
  var age = 46;
  var rating = 8.9;
  var isCool = true;
  var x = null;
  var y = undefined;
</script>
```

```
<script>
  var actor = {
    firstName: "Christian Bale",
    age: 46,
  };
  var numbers = [1, 2, 3, 4, 5];

  var sum = function (a, b) {
    return a + b;
  };
</script>
```



# Values vs. References

## Values

- primitive values are always assigned/passed as value copies.
- each variable holds its own copy of the value.

## References


- references are the idea that two or more variables are pointing at the same value, such that modifying this shared value would be reflected by an access via any of those references.
- only object values (arrays, objects, functions, etc.) are treated as references.



# Values vs. References


```
<script>
  // Value
  var myName = "Kyle";
  var yourName = myName;
  myName = "Frank";
  console.log(myName); // Frank
  console.log(yourName); // Kyle
</script>
```

```
<script>
  // Reference
  var myAddress = {
    street: "123 JS Blvd",
    city: "Austin",
    state: "TX",
  };
  var yourAddress = myAddress;
  // I've got to move to a new house!
  myAddress.street = "456 TS Ave";
  console.log(yourAddress.street); // 456 TS Ave
</script>
```



# Value Type Determination

```
<script>
  typeof 42; // "number"
  typeof "abc"; // "string"
  typeof true; // "boolean"
  typeof undefined; // "undefined"
  typeof null; // "object" -- oops, bug!
  typeof { a: 1 }; // "object"
  typeof [1, 2, 3]; // "object"
  typeof function hello() {}; // "function"
</script>
```



# Declaring and Using Variables

Values can either appear as literal values or they can be held in variables:

- **var**
- **let** - limited to “block scope”
- **const** - limited to “block scope” and cannot be re-assigned

```
<script>
  var adult = true;

  if (adult) {
    var myName = "Kyle";
    let age = 39;
    console.log("Shhh, this is a secret!");
  }

  console.log(myName); // Kyle
  console.log(age); // Error!
</script>
```





# Functions

- In JS, we should consider "function" to take the broader meaning of another related term: "procedure."
- **function declaration** - association between the identifier awesomeFunction and the function value happens during the **compile phase** of the code, before that code is executed.
- **function expression** - a function expression is not associated with its identifier until that statement during **runtime**
- **functions are values** that can be assigned and passed around
- can be defined to receive any number of parameters
- can return values using the return keyword



# Functions

```
<script>
  // Basic Syntax:
  function validFunctionName(parameter) {
    return statement;
  }

  // Function Expression:
  var fullName = function (firstName, lastName) {
    return `${firstName} ${lastName}`;
  };

  // Arrow Function:
  const double = (value) => {
    return value * 2;
  };
</script>
```



# Comparisons

===

- also described as the "**strict equality**" operator  
checking both the value and the type
- disallows any sort of type conversion
- all objects are compared by reference (identity) equality

==

- should be described as "**coercive equality**."
- If the value types being compared are different it allows coercion before the comparison
- <https://dorey.github.io/JavaScript-Equality-Table/>



# Comparisons

```
<script>
  3 === 3.0; // true
  "yes" === "yes"; // true
  null === null; // true
  false === false; // true

  42 === "42"; // false
  "hello" === "Hello"; // false
  true === 1; // false
  0 === null; // false
  "" === null; // false
  null === undefined; // false
</script>
```

```
<script>
  42 == "42"; // true
  1 == true; // true
</script>
```



# How We Organize in JS

Two major patterns for organizing code (data and behavior) are used broadly across the JS ecosystem: **classes** and **modules**.



# Classes

Classes define how such a data structure works, but classes are not themselves concrete values. To get a concrete value that you can use in the program, a class **must be instantiated** (with the new keyword) one or more times.

```
<script>
class Book {
  constructor(title, author, pubDate) {
    this.title = title;
    this.author = author;
    this.pubDate = pubDate;
  }
  print() {
    console.log(`
      Title: ${this.title}
      By: ${this.author}
      ${this.pubDate}
    `);
  }
}

var YDKJS = new Book({
  title: "You Don't Know JS",
  author: "Kyle Simpson",
  publishedOn: "June 2014",
});

YDKJS.print();
</script>
```



# Modules

- A module has the goal to group data and behavior together into logical units.
- **Classic Modules**
- **ES Modules**

# Digging to the Roots of JS

lower-level root characteristics of JS







# Iterables

- strings
- arrays
- maps
- sets
- all built-in iterables have three iterator forms available: keys-only (keys()), values-only (values()), and entries (entries()).

```
<script>
// an array is an iterable
var arr = [10, 20, 30];

for (let val of arr) {
  console.log(`Array value: ${val}`);
}
// Array value: 10
// Array value: 20
// Array value: 30
</script>
```



# Closure

- Closure is when a function remembers and continues to access variables from outside its scope, even when the function is executed in a different scope.
- Objects don't get closures.
- When an outer function finishes running, we would normally expect all of its variables to be garbage collected (removed from memory). If at least one inner function instance is still alive, then the outer function variables won't be garbage collected.



# Closure

```
<script>
function counter(step = 1) {
  var count = 0;
  return function increaseCount() {
    count = count + step;
    return count;
  };
}

var incBy1 = counter(1);
var incBy3 = counter(3);

incBy1(); // 1
incBy1(); // 2

incBy3(); // 3
incBy3(); // 6
incBy3(); // 9
</script>
```



# this Keyword

- **execution context** - another characteristic besides their scope that influences what they can access and it's exposed to the function via its this keyword
- entirely dependent **on how it is called** (regardless of where it is defined or even called from)

```
<script>
  var assignment = function () {
    console.log(`Kyle says to study ${this.topic}`);
  };

  var homework1 = {
    topic: "Git",
    assignment: assignment,
  };

  var homework2 = {
    topic: "JS",
    assignment: assignment,
  };

  console.log(homework1.assignment());
  console.log(homework2.assignment());
</script>
```



# Prototypes

- a **linkage** between two objects
- A series of objects linked together via prototypes is called the "**prototype chain**."
- The purpose of this prototype linkage (i.e., from an object B to another object A) is so that accesses against B for properties/methods that B does not have, are delegated to A to handle.

```
<script>
  var homework = {
    topic: "JS",
  };

  var otherHomework = Object.create(homework);

  otherHomework.topic; // "JS"
</script>
```

# The Bigger Picture

---



## Pillar 1: Scope and Closure

- **Lexical scope** - only variables at that level of scope nesting, or in higher/outer scopes, are accessible
- **Closure** - When a function makes reference to variables from an outer scope, and that function is passed around as a value and executed in other scopes, it maintains access to its original scope variables



## Pillar 2: Prototypes

- JS is one of very few languages where you have the option to **create objects directly** and explicitly, without first defining their structure in a class.
- behavior delegation - let objects cooperate through the **prototype chain**
- classes **aren't the only way** to use objects





## Pillar 3: Types and Coercion

- Types and Coercion is by far the most overlooked part of JS's nature.
- We don't have to follow the "static typing" way to be smart and solid with types in our programs.