Javascript in 7 Minutes: A Guide for 2024

thegoldenmule

July 1, 2024

Contents

1 Introductory Laments

This shouldn't need to exist yet here I am, writing it. Please lament with me. This guide is purely pragmatic. It is not entirely accurate, and in the future, much like learning anything, you may need to unlearn a bit. It is intended for students that have an assignment due fifteen minutes from now but forgot to learn Javascript. Perhaps you, dear reader, are in class *right now* and you're trying to catch up on something you saw on the board.

Have no fear, you'll be productive in seven minutes. Assignment done in fourteen.

2 Quick Facts

Quickly now, Javascript:

- ...runs on a *javascript runtime*, which may compile to bytecode, machine code, or run as a simple interpreter. The runtime usually runs in a browser or on a server, but plenty of runtimes are put other places.
- ...is *weakly* typed, meaning the language tries to automatically coerce types into other types instead of just failing (eg - it tries to make strings into ints for you).
- ...is also *dynamically* typed, meaning type information is determined while it's running, and heck, you can even change types midway through.
- ...has *historical issues* to watch out for, but is generally safe and fun to use in modern form.
- ...has great documentation on MDN (Mozilla Developer Network). There is no better reference.

3 Variables

Variables in Javascript are written differently depending on whether or not they are **mutable**, i.e. whether or not they can be changed. Get in the habit of always starting with variables that are **immutable**, or don't change.

```
// define an immutable (unchangeable) variable
const myVariable = 5;

// define a mutable (changeable) variable
let myMut = "hello";
```

Never, God help you, under any circumstances define variables in these ways:

```
foo = 5;

var bar = 6;
```

These are syntactically correct, but can easily create global variables (which, as we all know, are almost-categorically bad).

4 Types

As stated before, Javascript is weakly typed. This means that the runtime will try to *coerce* types for you automatically. In many languages, these examples will fail to compile, but in Javascript they may give completely nonsensical results:

```
const sum = 3 + "4"; // "34"
const wat = [] + 1; // "1"
```

In addition, the equality operator, ==, is very hard to use.

```
true == "true"; // false
3 == "3"; // true
undefined == null; // true
```

In general, just don't use == at all! Instead, use "strict equality", ===, which usually matches expectations much better.

```
true === "true"; // false
3 ===  "3"; // false
undefined === null; // false
```

Types fall in two categories: **primitives** or **objects**. There are seven primitives, but here are the ones you should know how to use right now:

```
// number (there are no integers or fixed points, only floats)
const num = 5.4;

// strings (there are no chars -- strings only)
const str = "I am a string";

// boolean
const bool = true; // or false, obviously
```

```
// undefined
let bar = undefined;

// null
let fizz = null;
```

Note that the **null** and **undefined** data types are not the same. Variables that have been declared but not initialized are undefined.

```
let a;

// a === undefined

a = 4;

// a === 4
```

Everything that is not a primitive is an **Object**, including **Function** and **Array**.

5 Objects

A Javascript **Object** is very similar to an associative array, map, or dictionary in other languages. Any type can act as a key.

```
const myObject = {
        counter: 0,
        foo: () => console.log("foo"),
        5: 6,
        "fizz": "buzz",
};
```

Note that, while the *myObject* reference is const, the object itself can be manipulated.

```
// perfectly legal
myObject.bar = true;

// throws error
myObject = {};
```

Furthermore, we can access things in an object in multiple ways:

```
myObject.counter; // 0

myObject["counter"]; // 0

const key = "counter";
myObject[key]; // 0
```

6 Functions

A mathematical function usually looks something like this:

$$\mathbf{f}(x) : \mathbb{R} \to \mathbb{R} \tag{1}$$

This defines the function, called **f**, that takes a real number ($\mathbb{R}$), called x, and returns a real number. (1) is a *different function* than this one:

$$\mathbf{g}(x) : \mathbb{R} \to \mathbb{N} \tag{2}$$

Not just because the name is different, but because this one returns a natural number ($\mathbb{N}$).

In Javascript, functions are written like this:

```
const sum = (a, b) => {
        return a + b;
};
```

This defines the function **sum** that takes two parameters, **a** and **b**, and returns their sum. The block {} and **return** statement can be removed if the function is a single line long.

```
const sum = (a, b) => a + b;
```

Since Functions are just objects, they too act like associative arrays, and can be passed around just like anything else.

```
const foo = () => 3;
foo.wat = "bar";

const bar = () => 4;
```

```
// this is absurd, but valid
bar[foo] = "no";

// functions can be passed as arguments
const c = (a, b) => a() + b();
c(foo, bar);
```

7 Arrays

In Javascript, arrays are implemented like **doubly linked lists**. Remember those? You can insert in front, push onto the end, pop things off, pull things out of the middle, and change the length dynamically. This means that Javascript Arrays can also be queues, stacks, and more.

Arrays are usually created using an "array literal", which looks like [ ], with zero or more items inside.

```
// create an array
const arr = [];

console.log(arr.length); // 0

arr.push(1); // [1]

arr.splice(0, 0, "start"); // ["start", 1]

arr.splice(0, 1); // [1]

// this creates a new array
const newArr = arr.concat([2, 3]); // [1, 2, 3]
```

Arrays have many useful, powerful methods on them that can be used to do things quickly.

```
// iterate
[1, 2, 3].forEach(i => console.log(i));

// transform
[1, 2, 3].map(i => i + 1); // [2, 3, 4]

// conditions
[1, 2, 3].every(i => i > 2); // false
[1, 2, 3].some(i => i > 2); // true
```

```
// flatten
[1, [2, 3, [4, 5]]].flat(Infinity); // [1, 2, 3, 4, 5]
```

8 Classes

Javascript is **not** a classical language. This does not mean that javascript cannot play violin, it means that it doesn't have "classical inheritance". C++, C#, Java – these languages encourage you to define "classes" and then create "instances" of classes. Classes are the means to share functionality or shape.

```
// C# Example
class MyClass {
        public int Foo;
}

...

// now create an instance
MyClass instance = new MyClass();
```

Javascript does not work the same way. While the **new** operator does exist (as does **class**), it doesn't work the same way. Javascript uses "prototypal inheritance", which is fundamentally different and generally considered "trickier". This guide does not describe Javascript classes, other than to avoid their use entirely.

9 Conclusion

The seven minute hourglass is almost empty, and here we are. You should now know Javascript as well as 90% of professional Javascript developers. GLHF.