Introduction to JavaScript

Despite its name, JavaScript is only related to Java in that the two share a few syntactic similarities. JavaScript's syntax in its initial development was loosely inspired by Java's syntax and it was called "LiveScript" when it first shipped in a beta version of Netscape Navigator in 1995, both to align with some of Netscape's other named offerings and as a nod to the fact that it runs "live" in the browser. Microsoft released their own implementation of JavaScript, "JScript," shortly afterwards with Internet Explorer 3.0.

Netscape submitted this early work to Ecma International, an organization that develops and publishes technical standards, to formalize and detail how this scripting language should be understood by other browsers. In 1997, Ecma International released ECMA-262, standardizing the first version of a scripting language called ECMAScript. ECMAScript is the standard that informs the creation of more specific scripting languages, for example, Microsoft's later work on the now-defunct JScript, Adobe's ActionScript, and JavaScript itself.

This distinction is important when discussing specific aspects and features of JavaScript. "ES5" refers to the first major "versioned" release of the ECMAScript standard in 2009, following years of more piecemeal development. "ES6" (or "ES2015") is shorthand for the standards set by the sixth edition of ECMAScript, released in 2015. After ES6, new editions of the ECMAScript standard have been released yearly, with each edition's changes and additions referred to by year as in "ES2016" or "ES2017".
The basic rules

Unlike compiled languages, JavaScript isn't translated from code a person writes into a form the browser can understand. A script is sent to the browser alongside assets like markup, images, and stylesheets, the browser interprets it the same way it was written: as a human-readable sequence of Unicode characters, parsed from left to right and top to bottom.

When a JavaScript interpreter receives a script, it first performs lexical analysis, parsing the long string of characters that makes up a script and converting it into the following discrete input elements:

- Tokens
- Format control characters
- Line terminators
- Comments
- Whitespace (almost always meaning tabs and spaces).

The results of a script won't persist after reloading or navigating away from the current page, unless you include explicit instructions to do otherwise in the script.

At a high level, JavaScript applications are made up of statements and expressions.

Statements:

A statement is a unit of instruction made up of one or more lines of code that represent an action. For example, you can use the following statement to assign a value to a variable named myVariable:

```
let myVariable = 4;

myVariable;
> 4
```

To be interpreted correctly, statements must end in a semicolon. However, these semicolons aren't always required when writing JavaScript. A feature called automatic semicolon insertion lets a line break following a complete statement be treated as a semicolon if a missing semicolon would cause an error.

ASI is error correction, not a permissive aspect of JavaScript itself. Because relying too much on this error correction can lead to ambiguity that breaks your code, you should still manually end every statement with a semicolon.
Block statements

A block statement groups any number of statements and declarations inside a pair of braces ({}). It lets you combine statements in places where JavaScript expects only one.

You'll most frequently see block statements alongside control flow statements, such as if:

```
if ( x === 2 ) {
  //some behavior;
}
```

Expressions:

An expression is a unit of code that results in a value, and can therefore be used wherever a value is expected. 2 + 2 is an expression that results in the value 4:

```
2 + 2;
> 4
```

The "grouping operator", a matched pair of enclosing parentheses, is used to group parts of an expression to ensure that a portion of the expression is evaluated as a single unit. For example, you might use a grouping operator to override the mathematical order of operations, or to improve the readability of code:

```
2 + 2 * 4;
> 10

( 2 + 2 ) * 4;
```

```
> 16

let myVariable = ( 2 + 2 );

myVariable;
> 4
```

Weak typing:

JavaScript is a weakly typed language, which means a data value doesn't need to be explicitly marked as a specific data type. Unlike a strongly typed language, JavaScript can infer the intended type from a value's context and convert the value to that type. This process is called type coercion.

For example, if you add a number to a string value in a strongly typed language, such as Python, the result is an error:

```
>>> "1" + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Instead of returning an error, JavaScript coerces the number value to a string and concatenates the two values, the most likely intended behavior when adding any value to a string:

```
"1" + 1;
> "11"
```

Data types can also be explicitly coerced. The following example coerces the numeric value 100 to a string value of "100" using JavaScript's built-in toString method:

```
let myVariable = 100;

typeof myVariable;
> "number"

myVariable = myVariable.toString();
> "100"

typeof myVariable;
> "string"
```

Case sensitivity:

Unlike HTML and the majority of CSS, JavaScript itself is fully case sensitive. This means you must always capitalize everything consistently, from the properties and methods built into the language to the identifiers you define yourself.

```
console.log( "Log this." );
> Log this.

console.Log( "Log this too." );
> Uncaught TypeError: console.Log is not a function

const myVariable = 2;

myvariable;
> Uncaught ReferenceError: myvariable is not defined

myVariable;
> 2
```

Whitespace:

JavaScript is whitespace insensitive. This means the interpreter ignores the amount and type (tabs or spaces) of whitespace used.

```
                    console.log(       "Log this"  );console.log("Log this too");
> "Log this."
> "Log this too."
```

However, the presence of whitespace can be significant as a separator between lexical tokens:

```
let x;

[tokens: [let] [x] ]

letx;
> Uncaught ReferenceError: letx is not defined

[tokens: [letx] ]
```

Where whitespace is used to separate meaningful lexical tokens, the parser ignores the amount and type of whitespace:

```
let          x                            =                          2;

[tokens: [let] [x] [=] [2] ]
```

The same is true of line breaks, though there are cases where line breaks can cause issues by prematurely ending a statement):

```
let x
=
2;
```

[tokens: [let] [x] [=] [2]]

Stylistically speaking, some types of statement frequently occupy a single line:

```
let x = 1;
let y = 2;
```

While some statements commonly use multiple lines:

```
if ( x == 2 ) {
  //some behavior;
}
```

These conventions are strictly for the sake of readability, however. JavaScript interprets the previous examples in the same way as the following:

```
let x=1;let y=2;

if(x==2){}
```

Because of this, an automated process that strips nonessential whitespace from script files to reduce transfer size is a common step in preparing JavaScript for a production environment, alongside a number of other optimizations.

Use of whitespace characters in JavaScript is largely a matter of author and maintainer preferences. JavaScript projects with multiple developers contributing code often suggest or enforce certain whitespace conventions to ensure consistent code formatting—for example, use of tabs or spaces to indent nested statements:

```
let myVariable = 10;

if ( typeof myVariable === "number" ) {
    console.log( "This variable is a number." );
    if( myVariable > 5 ) {
     console.log( "This variable is greater than five." );
    }
}

> "This variable is a number."
> "This variable is greater than five."
```