Grammar and types

This chapter discusses JavaScript's basic grammar, variable declarations, data types and literals.

Basics

JavaScript borrows most of its syntax from Java, C, and C++, but it has also been influenced by Awk, Perl, and Python.

JavaScript is **case-sensitive** and uses the **Unicode** character set. For example, the word Früh (which means "early" in German) could be used as a variable name.

```
let Früh = "foobar"
```

But, the variable `früh` is not the same as `Früh` because JavaScript is case sensitive.

In JavaScript, instructions are called statements and are separated by semicolons (;).

A semicolon is not necessary after a statement if it is written on its own line. But if more than one statement on a line is desired, then they *must* be separated by semicolons.

> **Note:** ECMAScript also has rules for automatic insertion of semicolons (ASI) to end statements. (For more information, see the detailed reference about JavaScript's lexical grammar.)

It is considered best practice, however, to always write a semicolon after a statement, even when it is not strictly needed. This practice reduces the chances of bugs getting into the code.

The source text of JavaScript script gets scanned from left to right, and is converted into a sequence of input elements which are *tokens*, *control characters*, *line terminators*, *comments*, or whitespace. (Spaces, tabs, and newline characters are considered whitespace.)

Comments

The syntax of **comments** is the same as in C++ and in many other languages:

```
// a one line comment

/* this is a longer,
 * multi-line comment
 */

/* You can't, however, /* nest comments */ SyntaxError */
```

Comments behave like whitespace, and are discarded during script execution.

> **Note:** You might also see a third type of comment syntax at the start of some JavaScript files, which looks something like this: `#!/usr/bin/env node`.
>
> This is called **hashbang comment** syntax, and is a special comment used to specify the path to a particular JavaScript engine that should execute the script. See Hashbang comments for more details.

Declarations

JavaScript has three kinds of variable declarations.

`var`
: Declares a variable, optionally initializing it to a value.

`let`
: Declares a block-scoped, local variable, optionally initializing it to a value.

`const`
: Declares a block-scoped, read-only named constant.

Variables

You use variables as symbolic names for values in your application. The names of variables, called identifiers, conform to certain rules.

A JavaScript identifier must start with a letter, underscore (`_`), or dollar sign (`$`). Subsequent characters can also be digits (`0`–`9`).

Because JavaScript is case sensitive, letters include the characters "`A`" through "`Z`" (uppercase) as well as "`a`" through "`z`" (lowercase).

You can use most of ISO 8859-1 or Unicode letters such as `å` and `ü` in identifiers. (For more details, see this blog post .) You can also use the Unicode escape sequences as characters in identifiers.

Some examples of legal names are `Number_hits`, `temp99`, `$credit`, and `_name`.

Declaring variables

You can declare a variable in two ways:

- With the keyword `var`. For example, `var x = 42`. This syntax can be used to declare both **local** and **global** variables, depending on the *execution context*.
- With the keyword `const` or `let`. For example, `let y = 13`. This syntax can be used to declare a block-scope local variable. (See Variable scope below.)

You can declare variables to unpack values from Object Literals using the Destructuring Assignment syntax. For example, `let { bar } = foo`. This will create a variable named `bar` and assign to it the value corresponding to the key of the same name from our object `foo`.

You can also assign a value to a variable For example, `x = 42`. This form creates an **undeclared global** variable. It also generates a strict JavaScript warning. Undeclared global variables can often lead to unexpected behavior. Thus, it is discouraged to use undeclared global variables.

Evaluating variables

A variable declared using the `var` or `let` statement with no assigned value specified has the value of `undefined`.

An attempt to access an undeclared variable results in a `ReferenceError` exception being thrown:

```
var a;
console.log('The value of a is ' + a); // The value of a is undefined
```

```
console.log('The value of b is ' + b); // The value of b is undefined
var b;
// This one may puzzle you until you read 'Variable hoisting' below

console.log('The value of c is ' + c); // Uncaught ReferenceError: c is not

let x;
console.log('The value of x is ' + x); // The value of x is undefined

console.log('The value of y is ' + y); // Uncaught ReferenceError: y is not
let y;
```

You can use `undefined` to determine whether a variable has a value. In the following code, the variable `input` is not assigned a value, and the `if` statement evaluates to `true`.

```
var input;
if (input === undefined) {
  doThis();
} else {
  doThat();
}
```

The `undefined` value behaves as `false` when used in a boolean context. For example, the following code executes the function `myFunction` because the `myArray` element is `undefined`:

```
var myArray = [];
if (!myArray[0]) myFunction();
```

The `undefined` value converts to `NaN` when used in numeric context.

```
var a;
a + 2;  // Evaluates to NaN
```

When you evaluate a `null` variable, the null value behaves as `0` in numeric contexts and as `false` in boolean contexts. For example:

```
var n = null;
console.log(n * 32); // Will log 0 to the console
```

Variable scope

When you declare a variable outside of any function, it is called a *global* variable, because it is available to any other code in the current document. When you declare a variable within a function, it is called a *local* variable, because it is available only within that function.

JavaScript before ECMAScript 2015 does not have block statement scope. Rather, a variable declared within a block is local to the *function (or global scope)* that the block resides within.

For example, the following code will log `5`, because the scope of `x` is the global context (or the function context if the code is part of a function). The scope of `x` is not limited to the immediate `if` statement block.

```
if (true) {
  var x = 5;
}
console.log(x);  // x is 5
```

This behavior changes when using the `let` declaration (introduced in ECMAScript 2015).

```
if (true) {
  let y = 5;
}
console.log(y);  // ReferenceError: y is not defined
```

Variable hoisting

Another unusual thing about variables in JavaScript is that you can refer to a variable declared later, without getting an exception.

This concept is known as **hoisting.** Variables in JavaScript are, in a sense, "hoisted" (or "lifted") to the top of the function or statement. However, variables that are hoisted return a value of `undefined`. So even if you declare and initialize after you use or refer to this variable, it still returns `undefined`.

```
/**
 * Example 1
 */
console.log(x === undefined); // true
var x = 3;

/**
 * Example 2
 */
```

```
// will return a value of undefined
var myvar = 'my value';

(function() {
  console.log(myvar); // undefined
  var myvar = 'local value';
})();
```

The above examples will be interpreted the same as:

```
/**
 * Example 1
 */
var x;
console.log(x === undefined); // true
x = 3;

/**
 * Example 2
 */
var myvar = 'my value';

(function() {
  var myvar;
  console.log(myvar); // undefined
  myvar = 'local value';
})();
```

Because of hoisting, all `var` statements in a function should be placed as near to the top of the function as possible. This best practice increases the clarity of the code.

In ECMAScript 2015, `let` and `const` **are hoisted but not initialized**. Referencing the variable in the block before the variable declaration results in a `ReferenceError`, because the variable is in a "temporal dead zone" from the start of the block until the declaration is processed.

```
console.log(x); // ReferenceError
let x = 3;
```

Function hoisting

In the case of functions, only function *declarations* are hoisted—but *not* the function *expressions*.

```
/* Function declaration */

foo(); // "bar"


function foo() {
  console.log('bar');
}

/* Function expression */

baz(); // TypeError: baz is not a function

var baz = function() {
  console.log('bar2');
};
```

Global variables

Global variables are in fact properties of the *global object*.

In web pages, the global object is `window`, so you can set and access global variables using the `window.variable` syntax.

Consequently, you can access global variables declared in one window or frame from another window or frame by specifying the `window` or `frame` name. For example, if a variable called `phoneNumber` is declared in a document, you can refer to this variable from an `iframe` as `parent.phoneNumber`.

Constants

You can create a read-only, named constant with the `const` keyword.

The syntax of a constant identifier is the same as any variable identifier: it must start with a letter, underscore, or dollar sign (`$`), and can contain alphabetic, numeric, or underscore characters.

```
const PI = 3.14;
```

A constant cannot change value through assignment or be re-declared while the script is running. It must be initialized to a value.

The scope rules for constants are the same as those for `let` block-scope variables. If the

The scope rules for constants are the same as those for `let` block-scope variables. If the `const` keyword is omitted, the identifier is assumed to represent a variable.

You cannot declare a constant with the same name as a function or variable in the same scope. For example:

```
// THIS WILL CAUSE AN ERROR
function f() {};
const f = 5;

// THIS WILL CAUSE AN ERROR TOO
function f() {
  const g = 5;
  var g;

  //statements
}
```

However, the properties of objects assigned to constants are not protected, so the following statement is executed without problems.

```
const MY_OBJECT = {'key': 'value'};
MY_OBJECT.key = 'otherValue';
```

Also, the contents of an array are not protected, so the following statement is executed without problems.

```
const MY_ARRAY = ['HTML','CSS'];
MY_ARRAY.push('JAVASCRIPT');
console.log(MY_ARRAY); //logs ['HTML','CSS','JAVASCRIPT'];
```

Data structures and types

Data types

The latest ECMAScript standard defines eight data types:

- Seven data types that are primitives:
  1. Boolean. `true` and `false`.
  2. null. A special keyword denoting a null value. (Because JavaScript is case-sensitive, `null` is not the same as `Null`, `NULL`, or any other variant.)

3. undefined. A top-level property whose value is not defined.
4. Number. An integer or floating point number. For example: `42` or `3.14159`.
5. BigInt. An integer with arbitrary precision. For example: `9007199254740992n`.
6. String. A sequence of characters that represent a text value. For example: "Howdy"
7. Symbol (new in ECMAScript 2015). A data type whose instances are unique and immutable.

- and Object

Although these data types are relatively few, they enable you to perform useful functions with your applications. Objects and functions are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your script can perform.

Data type conversion

JavaScript is a *dynamically typed* language. This means you don't have to specify the data type of a variable when you declare it. It also means that data types are automatically converted as-needed during script execution.

So, for example, you could define a variable as follows:

```
var answer = 42;
```

And later, you could assign the same variable a string value, for example:

```
answer = 'Thanks for all the fish...';
```

Because JavaScript is dynamically typed, this assignment does not cause an error message.

Numbers and the '+' operator

In expressions involving numeric and string values with the `+` operator, JavaScript converts numeric values to strings. For example, consider the following statements:

```
x = 'The answer is ' + 42 // "The answer is 42"
y = 42 + ' is the answer' // "42 is the answer"
```

With all other operators, JavaScript does *not* convert numeric values to strings. For example:

```
'37' - 7 // 30
'37' + 7 // "377"
```

Converting strings to numbers

In the case that a value representing a number is in memory as a string, there are methods for conversion.

- `parseInt()`
- `parseFloat()`

`parseInt` only returns whole numbers, so its use is diminished for decimals.

> **Note:** Additionally, a best practice for `parseInt` is to always include the *radix* parameter. The radix parameter is used to specify which numerical system is to be used.

```
parseInt('101', 2) // 5
```

An alternative method of retrieving a number from a string is with the `+` (unary plus) operator:

```
'1.1' + '1.1' // '1.11.1'
(+'1.1') + (+'1.1') // 2.2
// Note: the parentheses are added for clarity, not required.
```

Literals

Literals represent values in JavaScript. These are fixed values—not variables—that you *literally* provide in your script. This section describes the following types of literals:

- Array literals
- Boolean literals
- Floating-point literals
- Numeric literals
- Object literals
- RegExp literals
- String literals

Array literals

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets (`[]`). When you create an array using an array literal, it

is initialized with the specified values as its elements, and its `length` is set to the number of arguments specified.

The following example creates the `coffees` array with three elements and a `length` of three:

```
let coffees = ['French Roast', 'Colombian', 'Kona'];
```

> **Note:** An array literal is a type of *object initializer*. See Using Object Initializers.

If an array is created using a literal in a top-level script, JavaScript interprets the array each time it evaluates the expression containing the array literal. In addition, a literal used in a function is created each time the function is called.

> **Note:** Array literals are also `Array` objects. See `Array` and Indexed collections for details on `Array` objects.

Extra commas in array literals

You do not have to specify all elements in an array literal. If you put two commas in a row, the array fills in the value `undefined` for the unspecified elements. The following example creates the `fish` array:

```
let fish = ['Lion', , 'Angel'];
```

This array has two elements with values and one empty element:

- `fish[0]` is "Lion"
- `fish[1]` is `undefined`
- `fish[2]` is "Angel"

If you include a trailing comma at the end of the list of elements, the comma is ignored.

In the following example, the `length` of the array is three. There is no `myList[3]`. All other commas in the list indicate a new element.

> **Note:** Trailing commas can create errors in older browser versions and it is a best practice to remove them.

```
let myList = ['home', , 'school', ];
```

In the following example, the `length` of the array is four, and `myList[0]` and `myList[2]` are missing.

```
let myList = [ ,'home', , 'school'];
```

In the following example, the `length` of the array is four, and `myList[1]` and `myList[3]` are missing. **Only the last comma is ignored.**

```
let myList = ['home', , 'school', , ];
```

Understanding the behavior of extra commas is important to understanding JavaScript as a language.

However, when writing your own code, you should explicitly declare the missing elements as `undefined`. Doing this increases your code's clarity and maintainability.

Boolean literals

The Boolean type has two literal values: `true` and `false`.

> **Note:** Do not confuse the primitive Boolean values `true` and `false` with the true and false values of the `Boolean` object.
>
> The Boolean object is a wrapper around the primitive Boolean data type. See `Boolean` for more information.

Numeric literals

JavaScript numeric literals include integer literals in different bases as well as floating-point literals in base-10.

Note that the language specification requires numeric literals to be unsigned. Nevertheless, code fragments like `-123.4` are fine, being interpreted as a unary `-` operator applied to the numeric literal `123.4`.

numeric literal `123.44`

Integer literals

Integer and `BigInt` literals can be written in decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2).

- A *decimal* integer literal is a sequence of digits without a leading `0` (zero).
- A leading `0` (zero) on an integer literal, or a leading `0o` (or `0O`) indicates it is in *octal*. Octal integer literals can include only the digits `0–7`.
- A leading `0x` (or `0X`) indicates a *hexadecimal* integer literal. Hexadecimal integers can include digits (`0–9`) and the letters `a–f` and `A–F`. (The case of a character does not change its value. Therefore: `0xa = 0xA = 10` and `0xf = 0xF = 15`.)
- A leading `0b` (or `0B`) indicates a *binary* integer literal. Binary integer literals can only include the digits `0` and `1`.
- A trailing `n` suffix on an integer literal indicates a `BigInt` literal. The integer literal can use any of the above bases. Note that leading-zero octal syntax like `0123n` is not allowed, but `0o123n` is fine.

Some examples of integer literals are:

```
0, 117, 123456789123456789n             (decimal, base 10)
015, 0001, 0o777777777777n              (octal, base 8)
0x1123, 0x00111, 0x123456789ABCDEFn     (hexadecimal, "hex" or base 16)
0b11, 0b0011, 0b1110100101010101010n    (binary, base 2)
```

For more information, see Numeric literals in the Lexical grammar reference.

Floating-point literals

A floating-point literal can have the following parts:

- An unsigned decimal integer,
- A decimal point ("`.`"),
- A fraction (another decimal number),
- An exponent.

The exponent part is an "`e`" or "`E`" followed by an integer, which can be signed (preceded by "`+`" or "`-`"). A floating-point literal must have at least one digit, and either a decimal point or "`e`" (or "`E`").

More succinctly, the syntax is:

More succinctly, the syntax is:

```
[digits].[digits][(E|e)[(+|-)]digits]
```

For example:

```
3.1415926
.123456789
3.1E+12
.1e-23
```

Object literals

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`).

> **Warning:** Do not use an object literal at the beginning of a statement! This will lead to an error (or not behave as you expect), because the `{` will be interpreted as the beginning of a block.

The following is an example of an object literal. The first element of the `car` object defines a property, `myCar` , and assigns to it a new string, "`Saturn`"; the second element, the `getCar` property, is immediately assigned the result of invoking the function `(carTypes("Honda"))` ; the third element, the `special` property, uses an existing variable (`sales`).

```
var sales = 'Toyota';

function carTypes(name) {
  if (name === 'Honda') {
    return name;
  } else {
    return "Sorry, we don't sell " + name + ".";
  }
}

var car = { myCar: 'Saturn', getCar: carTypes('Honda'), special: sales };

console.log(car.myCar);   // Saturn
console.log(car.getCar);  // Honda
console.log(car.special); // Toyota
```

Additionally, you can use a numeric or string literal for the name of a property or nest an object inside another. The following example uses these options.

```
var car = { manyCars: {a: 'Saab', b: 'Jeep'}, 7: 'Mazda' };

console.log(car.manyCars.b); // Jeep
console.log(car[7]); // Mazda
```

Object property names can be any string, including the empty string. If the property name would not be a valid JavaScript identifier or number, it must be enclosed in quotes.

Property names that are not valid identifiers cannot be accessed as a dot (`.`) property, but *can* be accessed and set with the array-like notation(" `[]` ").

```
var unusualPropertyNames = {
  '': 'An empty string',
  '!': 'Bang!'
}
console.log(unusualPropertyNames.'');   // SyntaxError: Unexpected string
console.log(unusualPropertyNames['']);  // An empty string
console.log(unusualPropertyNames.!);    // SyntaxError: Unexpected token !
console.log(unusualPropertyNames['!']); // Bang!
```

Enhanced Object literals

In ES2015, object literals are extended to support setting the prototype at construction, shorthand for `foo: foo` assignments, defining methods, making `super` calls, and computing property names with expressions.

Together, these also bring object literals and class declarations closer together, and allow object-based design to benefit from some of the same conveniences.

```
var obj = {
    // __proto__
    __proto__: theProtoObj,
    // Shorthand for ‘handler: handler’
    handler,
    // Methods
    toString() {
     // Super calls
     return 'd ' + super.toString();
    },
    // Computed (dynamic) property names
    [ 'prop_' + (() => 42)() ]: 42
};
```

RegExp literals

A regex literal (which is defined in detail later) is a pattern enclosed between slashes. The following is an example of a regex literal.

```
var re = /ab+c/;
```

String literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type (that is, either both single quotation marks, or both double quotation marks).

The following are examples of string literals:

```
'foo'
"bar"
'1234'
'one line \n another line'
"John's cat"
```

You should use string literals unless you specifically need to use a `String` object. See `String` for details on `String` objects.

You can call any of the `String` object's methods on a string literal value. JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the `String.length` property with a string literal:

```
// Will print the number of symbols in the string including whitespace.
console.log("John's cat".length)  // In this case, 10.
```

Template literals are also available. Template literals are enclosed by the back-tick (`) (grave accent) character instead of double or single quotes.

Template literals provide syntactic sugar for constructing strings. (This is similar to string interpolation features in Perl, Python, and more.)

```
// Basic literal string creation
`In JavaScript '\n' is a line-feed.`

// Multiline strings
`In JavaScript, template strings can run
 over multiple lines, but double and single
```

```
 quoted strings cannot.`

// String interpolation
var name = 'Bob', time = 'today';
`Hello ${name}, how are you ${time}?`
```

Tagged templates are a compact syntax for specifying a template literal along with a call to a "tag" function for parsing it; the name of the template tag function precedes the template literal — as in the following example, where the template tag function is named "`myTag`":

```
let myTag = (str, name, age) => `${str[0]}${name}${str[1]}${age}${str[2]}`;
let [name, age] = ['Mika', 28];
myTag`Participant "${ name }" is ${ age } years old.`;
// Participant "Mika" is 28 years old.
```

Using special characters in strings

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.

```
'one line \n another line'
```

The following table lists the special characters that you can use in JavaScript strings.

Table: JavaScript special characters

Character	Meaning
`\0`	Null Byte
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\'`	Apostrophe or single quote

\" **Character**	Double quote **Meaning**
`\\`	Backslash character

`\XXX`	The character with the Latin-1 encoding specified by up to three octal digits *XXX* between `0` and `377`. For example, `\251` is the octal sequence for the copyright symbol.
`\xXX`	The character with the Latin-1 encoding specified by the two hexadecimal digits *XX* between `00` and `FF`. For example, `\xA9` is the hexadecimal sequence for the copyright symbol.
`\uXXXX`	The Unicode character specified by the four hexadecimal digits *XXXX*. For example, `\u00A9` is the Unicode sequence for the copyright symbol. See Unicode escape sequences.
`\u{XXXXX}`	Unicode code point escapes. For example, `\u{2F804}` is the same as the simple Unicode escapes `\uD87E\uDC04`.

Escaping characters

For characters not listed in the table, a preceding backslash is ignored, but this usage is deprecated and should be avoided.

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark. For example:

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service.";
console.log(quote);
```

The result of this would be:

```
He read "The Cremation of Sam McGee" by R.W. Service.
```

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path `c:\temp` to a string, use the following:

```
var home = 'c:\\temp';
```

You can also escape line breaks by preceding them with backslash. The backslash and line break are both removed from the value of the string.

```
var str = 'this string \
is broken \
across multiple \
lines.'
console.log(str);   // this string is broken across multiple lines.
```

Although JavaScript does not have "heredoc" syntax, you can get close by adding a line break escape and an escaped line break at the end of each line:

```
var poem =
'Roses are red,\n\
Violets are blue.\n\
Sugar is sweet,\n\
and so is foo.'
```

ECMAScript 2015 introduces a new type of literal, namely **template literals**. This allows for many new features, including multiline strings!

```
var poem =
`Roses are red,
Violets are blue.
Sugar is sweet,
and so is foo.`
```

More information

This chapter focuses on basic syntax for declarations and types. To learn more about JavaScript's language constructs, see also the following chapters in this guide:

- Control flow and error handling
- Loops and iteration
- Functions
- Expressions and operators

In the next chapter, we will have a look at control flow constructs and error handling.

Last modified: Jul 20, 2021, by MDN contributors