**Values, Variables, and Literals**

This chapter discusses values that JavaScript recognizes and describes the fundamental building blocks of JavaScript expressions: variables and literals.

This chapter contains the following sections:

* [Values](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1008306)

* [Variables](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1008330)

* [Literals](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1008348)

* [Unicode](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1009568)

**Values**

JavaScript recognizes the following types of values:

* Numbers, such as 42 or 3.14159.
* Logical (Boolean) values, either true or false.
* Strings, such as "Howdy!".
* null, a special keyword denoting a null value; null is also a primitive value. Because JavaScript is case sensitive, null is not the same as Null, NULL, or any other variant.
* undefined, a top-level property whose value is undefined; undefined is also a primitive value.

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 application can perform.

**Data Type Conversion**

JavaScript is a dynamically typed language. That means you do not have to specify the data type of a variable when you declare it, and data types are converted automatically 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.

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 // returns "The answer is 42"  
y = 42 + " is the answer" // returns "42 is the answer"

In statements involving other operators, JavaScript does not convert numeric values to strings. For example:

"37" - 7 // returns 30  
"37" + 7 // returns 377

**Variables**

A JavaScript identifier, or *name,* must start with a letter or underscore ("\_"); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Some examples of legal names are Number\_hits, temp99, and \_name.

**Declaring Variables**

You can declare a variable in two ways:

* By simply assigning it a value. For example, x = 42
* With the keyword var. For example, var x = 42

**Evaluating Variables**

A variable or array element that has not been assigned a value has the value undefined. The result of evaluating an unassigned variable depends on how it was declared:

* If the unassigned variable was declared without var, the evaluation results in a runtime error.
* If the unassigned variable was declared with var, the evaluation results in the undefined value, or NaN in numeric contexts.

The following code demonstrates evaluating unassigned variables.

function f1() {  
   return y - 2;  
}  
f1() //Causes runtime error

function f2() {  
   return var y - 2;  
}  
f2() //returns NaN

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 as a Boolean value. For example, the following code executes the function myFunction because the array element is not defined:

myArray=new Array()  
if (!myArray[0])  
   myFunction()

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  
n \* 32 //returns 0

**Variable Scope**

Using var to declare a global variable is optional. However, you must use var to declare a variable inside a function.

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 FRAMESET document, you can refer to this variable from a child frame as parent.phoneNumber.

**Literals**

You use literals to 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](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1011655)

* [Boolean Literals](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1012095)

* [Floating-Point Literals](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1009427)

* [Integers](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1008351)

* [Object Literals](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1009450)

* [String Literals](https://docs.oracle.com/cd/E19957-01/816-6409-10/ident.htm" \l "1008368)

**Array Literals**

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

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

coffees = ["French Roast", "Columbian", "Kona"].

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.

**Array literals are also 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 is created with spaces for the unspecified elements. The following example creates the fish array:

fish = ["Lion", , "Angel"]

### Array Object

JavaScript does not have an explicit array data type. However, you can use the predefined Array object and its methods to work with arrays in your applications.

* The Array object has **methods** for manipulating arrays in various ways, such as joining, reversing, and sorting them.
* It has a **property** for determining the array length and other properties for use with regular expressions.

#### Creating an Array

To create an Array object:

1. arrayObjectName = new Array(element0, element1, ..., element*N*)  
2. arrayObjectName = new Array(arrayLength)

arrayLength is the initial length of the array. The following code creates an array of five elements:

billingMethod = new Array(5)

Array literals are also Array objects; for example, the following literal is an Array object.

coffees = ["French Roast", "Columbian", "Kona"]

**Array Methods**

The Array object has the following methods:

* **Concat**  joins two arrays and returns a new array.
* **Join**  joins all elements of an array into a string.
* **Pop**  removes the last element from an array and returns that element.
* **push**  adds one or more elements to the end of an array and returns that last element added.
* **reverse** transposes the elements of an array: the first array element becomes the last and the last becomes the first.
* **shift**  removes the first element from an array and returns that element
* **slice**  extracts a section of an array and returns a new array.
* **splice**  adds and/or removes elements from an array.
* **sort**  sorts the elements of an array.
* **unshift** adds one or more elements to the front of an array and returns the new length of the array.

For example, suppose you define the following array:

myArray = new Array("Wind","Rain","Fire")

myArray.join() returns "Wind,Rain,Fire";

myArray.reverse transposes the array so that myArray[0] is "Fire", myArray[1] is "Rain", and myArray[2] is "Wind".

myArray.sort sorts the array so that myArray[0] is "Fire",myArray[1] is "Rain", and myArray[2] is "Wind".

#### Two-Dimensional Arrays

The following code creates a two-dimensional array.

a = new Array(4)  
for (i=0; i < 4; i++) {  
   a[i] = new Array(4)  
   for (j=0; j < 4; j++) {  
      a[i][j] = "["+i+","+j+"]"  
   }  
}

# [**How to create multidimensional array**](https://stackoverflow.com/questions/7545641/how-to-create-multidimensional-array)

Declared without value assignment.

2 dimensions...

var arrayName = new Array(new Array());

3 dimensions...

var arrayName = new Array(new Array(new Array()));

**Boolean Literals**

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

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.

**Floating-Point Literals**

A floating-point literal can have the following parts:

1. A decimal integer
2. A decimal point (".")
3. A fraction (another decimal number)
4. 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").

Some examples of floating-point literals are 3.1415, -3.1E12, .1e12, and 2E-12

**Integers**

Integers can be expressed in decimal (base 10), hexadecimal (base 16), and octal (base 8). A decimal integer literal consists of a sequence of digits without a leading 0 (zero).

A leading 0 (zero) on an integer literal indicates it is in octal. Octal integers can include only the digits 0-7.

A leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F.

Some examples of integer literals are: 42, 0xFFF, and -345.

**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 ({}). You should not use an object literal at the beginning of a statement. This will lead to an error.

The following is an example of an object literal. The first element of the car object defines a property, myCar; the second element, the getCar property, invokes a function (Cars("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 + ".";  
}

car = {myCar: "Saturn", getCar: CarTypes("Honda"), special: Sales}

document.write(car.myCar); // Saturn  
document.write(car.getCar); // Honda  
document.write(car.special); // Toyota

Additionally, you can use an index for the object, the index property (for example, 7), or nest an object inside another. The following example uses these options. These features, however, may not be supported by other ECMA-compliant browsers.

car = {manyCars: {a: "Saab", b: "Jeep"}, 7: "Mazda"}

document.write(car.manyCars.b); // Jeep  
document.write(car[7]); // Mazda

## Creating New Objects

JavaScript has a number of predefined objects. In addition, you can create your own objects. In JavaScript 1.2, you can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object using that function and the new operator.

### #1 Using Object Initializers

In addition to creating objects using a constructor function, you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation. "Object initializer" is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

objectName = {property1:value1, property2:value2,..., property*N*:value*N*}

**#2 Using a Constructor Function**

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function.
2. Create an instance of the object with new.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called car, and you want it to have properties for make, model, year, and color. To do this, you would write the following function:

function car(make, model, year) {  
   this.make = make  
   this.model = model  
   this.year = year  
}

Notice the use of this to assign values to the object's properties based on the values passed to the function.

Now you can create an object called mycar as follows:

mycar = new car("Eagle", "Talon TSi", 1993)

**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:

* "blah"
* 'blah'
* "1234"
* "one line \n another line"

You can call any of the methods of the String object 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.

You should use string literals unless you specifically need to use a String object.

To create a Date object:

dateObjectName = new Date([parameters])

where dateObjectName is the name of the Date object being created; it can be a new object or a property of an existing object.

The parameters in the preceding syntax can be any of the following:

* Nothing: creates today's date and time. For example, today = new Date().
* A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, Xmas95 = new Date("December 25, 1995 13:30:00"). If you omit hours, minutes, or seconds, the value will be set to zero.
* A set of integer values for year, month, and day. For example, Xmas95 = new Date(1995,11,25). A set of values for year, month, day, hour, minute, and seconds. For example, Xmas95 = new Date(1995,11,25,9,30,0).

**Methods of the Date Object**

The Date object methods for handling dates and times fall into these broad categories:

1. "set" methods, for setting date and time values in Date objects.
2. "get" methods, for getting date and time values from Date objects.
3. "to" methods, for returning string values from Date objects.

### Date Object

JavaScript does not have a date data type. However, you can use the Date object and its methods to work with dates and times in your applications.

### Function Object

The predefined Function object specifies a string of JavaScript code to be compiled as a function.

To create a Function object:

functionObjectName = new Function ([arg1, arg2, ... argn], functionBody)

functionObjectName is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as window.onerror.

arg1, arg2, ... argn are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theForm".

functionBody is a string specifying the JavaScript code to be compiled as the function body.

Function objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the function statement.

## do...while statement

The [do...while](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/do...while) statement repeats until a specified condition evaluates to false. A do...while statement looks as follows:

do

statement

while (condition);

statement is always executed once before the condition is checked (and then again until the while condition returns false). To execute multiple statements, use a block statement ({ ... }) to group those statements. If condition is true, the statement executes again. At the end of every execution, the condition is checked. When the condition is false, execution stops and control passes to the statement following do...while.

In the following example, the do loop iterates at least once and reiterates until i is no longer less than 5.

var i = 0;

do {

i += 1;

console.log(i);

} while (i < 5);

## while statement

A [while](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/while) statement executes its statements as long as a specified condition evaluates to true. A while statement looks as follows:

while (condition)

statement

If the condition becomes false, statement within the loop stops executing and control passes to the statement following the loop.

The condition test occurs before statement in the loop is executed. If the condition returns true, statement is executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following while.

## break statement

Use the [break](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/break) statement to terminate a loop, switch, or in conjunction with a labeled statement.

* When you use break without a label, it terminates the innermost enclosing while, do-while, for, or switch immediately and transfers control to the following statement.
* When you use break with a label, it terminates the specified labeled statement.

The syntax of the break statement looks like this:

The following example iterates through the elements in an array until it finds the index of an element whose value is theValue:

for (var i = 0; i < a.length; i++) {

if (a[i] == theValue) {

break;

}

}

Breaking to a label:

var x = 0;

var z = 0;

labelCancelLoops: while (true) {

console.log('Outer loops: ' + x);

x += 1;

z = 1;

while (true) {

console.log('Inner loops: ' + z);

z += 1;

if (z === 10 && x === 10) {

break labelCancelLoops;

} else if (z === 10) {

break;

}

}

}

## continue statement

The [continue](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/continue) statement can be used to restart a while, do-while, for, or label statement.

* When you use continue without a label, it terminates the current iteration of the innermost enclosing while, do-while, or for statement and continues execution of the loop with the next iteration. In contrast to the break statement, continue does not terminate the execution of the loop entirely. In a while loop, it jumps back to the condition. In a for loop, it jumps to the increment-expression.
* When you use continue with a label, it applies to the looping statement identified with that label.

## for...of statement

The [for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/for...of) statement creates a loop Iterating over [iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable) (including [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/functions/arguments) object and so on),

## for...in statement

## For each distinct property, JavaScript executes the specified statements.

The following example shows the difference between a for...of loop and a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/statements/for...in) loop. While for...in iterates over property names, for...of iterates over property values:

var arr = [3, 5, 7];

arr.foo = 'hello';

arr.bar = 'kitty';

for (var i in arr) {

console.log(i); // logs "0", "1", "2", "foo", "bar"

}

for (var i of arr) {

console.log(i); // logs 3, 5, 7

}

## Pre-defined functions

JavaScript has several top-level, built-in functions:

[**eval()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval)

The **eval()** method evaluates JavaScript code represented as a string.

[**uneval()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/uneval)

The **uneval()** method creates a string representation of the source code of an [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object).

[**isFinite()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isFinite)

The global **isFinite()** function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.

[**isNaN()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN)

The **isNaN()** function determines whether a value is [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) or not. Note: coercion inside the isNaN function has [interesting](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN#Description) rules; you may alternatively want to use [Number.isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN), as defined in ECMAScript 2015, or you can use [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) to determine if the value is Not-A-Number.

[**parseFloat()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat)

The **parseFloat()** function parses a string argument and returns a floating point number.

[**parseInt()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt)

The **parseInt()** function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).

[**decodeURI()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/decodeURI)

The **decodeURI()** function decodes a Uniform Resource Identifier (URI) previously created by [encodeURI](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURI) or by a similar routine.

[**decodeURIComponent()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/decodeURIComponent)

The **decodeURIComponent()** method decodes a Uniform Resource Identifier (URI) component previously created by [encodeURIComponent](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent) or by a similar routine.

[**encodeURI()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURI)

The **encodeURI()** method encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

[**encodeURIComponent()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent)

The **encodeURIComponent()** method encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

## Exporting module features

The first thing you need to do to get access to module features is export them. This is done using the [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) statement.

The easiest way to use it is to place it in front of any items you want exported out of the module, for example:

export const name = 'square';

You can export functions, var, let, const, and — as we'll see later — classes. They need to be top-level items; you can't use export inside a function.

## Importing features into your script

Once you've exported some features out of your module, you need to import them into your script to be able to use them. The simplest way to do this is as follows:

import { name, draw, reportArea, reportPerimeter } from './modules/square.js';

we are using the dot (.) syntax to mean "the current location", followed by the path beyond that to the file we are trying to find. This is much better than writing out the entire relative path each time, as it is shorter, and it makes the URL portable.

**Note**: In some module systems, you can omit the file extension and the dot (e.g. '/modules/square'). This doesn't work in native JavaScript modules.

## Default exports versus named exports

The functionality we've exported so far has been comprised of **named exports** — each item (be it a function, const, etc.) has been referred to by its name upon export, and that name has been used to refer to it on import as well.

There is also a type of export called the **default export** — this is designed to make it easy to have a default function provided by a module

## Creating a module object

The above method works OK, but it's a little messy and longwinded. An even better solution is to import each module's features inside a module object. The following syntax form does that:

import \* as Module from './modules/module.js';

This grabs all the exports available inside module.js, and makes them available as members of an object Module, effectively giving it its own namespace. So for example:

Module.function1()

Module.function2()

etc.

Again, let's look at a real example. If you go to our [module-objects](https://github.com/mdn/js-examples/tree/master/modules/module-objects) directory, you'll see the same example again, but rewritten to take advantage of this new syntax. In the modules, the exports are all in the following simple form:

export { name, draw, reportArea, reportPerimeter };

The imports on the other hand look like this:

import \* as Canvas from './modules/canvas.js';

import \* as Square from './modules/square.js';

import \* as Circle from './modules/circle.js';

import \* as Triangle from './modules/triangle.js';

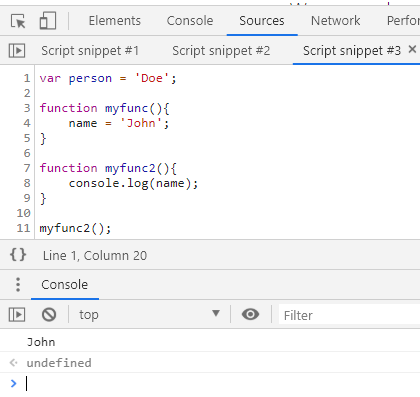
In each case, you can now access the module's imports underneath the specified object name, for example:

let square1 = Square.draw(myCanvas.ctx, 50, 50, 100, 'blue');

Square.reportArea(square1.length, reportList);

Square.reportPerimeter(square1.length, reportList);

So you can now write the code just the same as before (as long as you include the object names where needed), and the imports are much neater.

1. Which of the following is an example of anonymous function in JavaScript?
   1. var myFunc = function(){ };
2. A variable declared without var keyword inside a function will become **global** variable.
   1. 
3. What will be the output of the following JavaScript code?

x = 1;

console.log('x = ' + x);

var x;

**Correct Answer:** x = 1