JavaScript typeof

[❮ Previous](https://www.w3schools.com/js/js_maps.asp)[Next ❯](https://www.w3schools.com/js/js_type_conversion.asp)

In JavaScript there are 5 different data types that can contain values:

* string
* number
* boolean
* object
* function

There are 6 types of objects:

* Object
* Date
* Array
* String
* Number
* Boolean

And 2 data types that cannot contain values:

* null
* undefined

The typeof Operator

You can use the typeof operator to find the data type of a JavaScript variable.

Example

typeof "John"                 // Returns "string"  
typeof 3.14                   // Returns "number"  
typeof NaN                    // Returns "number"  
typeof false                  // Returns "boolean"  
typeof [1,2,3,4]              // Returns "object"  
typeof {name:'John', age:34}  // Returns "object"  
typeof new Date()             // Returns "object"  
typeof function () {}         // Returns "function"  
typeof myCar                  // Returns "undefined" \*  
typeof null                   // Returns "object"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_typeof_all)

Please observe:

* The data type of NaN is number
* The data type of an array is object
* The data type of a date is object
* The data type of null is object
* The data type of an undefined variable is **undefined** \*
* The data type of a variable that has not been assigned a value is also **undefined** \*

You cannot use typeof to determine if a JavaScript object is an array (or a date).

Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The typeof operator can return one of these primitive types:

* string
* number
* boolean
* undefined

Example

typeof "John"              // Returns "string"  
typeof 3.14                // Returns "number"  
typeof true                // Returns "boolean"  
typeof false               // Returns "boolean"  
typeof x                   // Returns "undefined" (if x has no value)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_typeof_primitive)

Complex Data

The typeof operator can return one of two complex types:

* function
* object

The typeof operator returns "object" for objects, arrays, and null.

The typeof operator does not return "object" for functions.

Example

typeof {name:'John', age:34} // Returns "object"  
typeof [1,2,3,4]             // Returns "object" (not "array", see note below)  
typeof null                  // Returns "object"  
typeof function myFunc(){}   // Returns "function"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_typeof_complex)

The typeof operator returns "object" for arrays because in JavaScript arrays are objects.

The Data Type of typeof

The typeofoperator is not a variable. It is an operator. Operators ( + - \* / ) do not have any data type.

But, the typeof operator always **returns a string** (containing the type of the operand).

The constructor Property

The constructor property returns the constructor function for all JavaScript variables.

Example

"John".constructor                // Returns function String()  {[native code]}  
(3.14).constructor                // Returns function Number()  {[native code]}  
false.constructor                 // Returns function Boolean() {[native code]}  
[1,2,3,4].constructor             // Returns function Array()   {[native code]}  
{name:'John',age:34}.constructor  // Returns function Object()  {[native code]}  
new Date().constructor            // Returns function Date()    {[native code]}  
function () {}.constructor        // Returns function Function(){[native code]}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_constructor_all)

You can check the constructor property to find out if an object is an Array (contains the word "Array"):

Example

function isArray(myArray) {  
  return myArray.constructor.toString().indexOf("Array") > -1;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_array_isarray)

Or even simpler, you can check if the object is an **Array function**:

Example

function isArray(myArray) {  
  return myArray.constructor === Array;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_isarray)

You can check the constructor property to find out if an object is a Date (contains the word "Date"):

Example

function isDate(myDate) {  
  return myDate.constructor.toString().indexOf("Date") > -1;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_date_isdate)

Or even simpler, you can check if the object is a **Date function**:

Example

function isDate(myDate) {  
  return myDate.constructor === Date;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_isdate)

Undefined

In JavaScript, a variable without a value, has the value undefined. The type is also undefined.

Example

let car;    // Value is undefined, type is undefined

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_undefined)

Any variable can be emptied, by setting the value to undefined. The type will also be undefined.

Example

car = undefined;    // Value is undefined, type is undefined

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_undefined_2)

Empty Values

An empty value has nothing to do with undefined.

An empty string has both a legal value and a type.

Example

let car = "";    // The value is "", the typeof is "string"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_empty)

Null

In JavaScript null is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of null is an object.

You can consider it a bug in JavaScript that typeof null is an object. It should be null.

You can empty an object by setting it to null:

Example

let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};  
person = null;    // Now value is null, but type is still an object

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_null)

You can also empty an object by setting it to undefined:

Example

let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};  
person = undefined;   // Now both value and type is undefined

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_undefined_1)

Difference Between Undefined and Null

undefined and null are equal in value but different in type:

typeof undefined           // undefined  
typeof null                // object  
  
null === undefined         // false  
null == undefined          // true

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_datatypes_undefined_3)

JavaScript Type Conversion

[❮ Previous](https://www.w3schools.com/js/js_typeof.asp)[Next ❯](https://www.w3schools.com/js/js_bitwise.asp)

* Converting Strings to Numbers
* Converting Numbers to Strings
* Converting Dates to Numbers
* Converting Numbers to Dates
* Converting Booleans to Numbers
* Converting Numbers to Booleans

JavaScript Type Conversion

JavaScript variables can be converted to a new variable and another data type:

* By the use of a JavaScript function
* **Automatically** by JavaScript itself

Converting Strings to Numbers

The global method Number() can convert strings to numbers.

Strings containing numbers (like "3.14") convert to numbers (like 3.14).

Empty strings convert to 0.

Anything else converts to NaN (Not a Number).

Number("3.14")    // returns 3.14  
Number(" ")       // returns 0  
Number("")        // returns 0  
Number("99 88")   // returns NaN

Number Methods

In the chapter [Number Methods](https://www.w3schools.com/js/js_number_methods.asp), you will find more methods that can be used to convert strings to numbers:

|  |  |
| --- | --- |
| **Method** | **Description** |
| Number() | Returns a number, converted from its argument |
|  |  |
| parseFloat() | Parses a string and returns a floating point number |
| parseInt() | Parses a string and returns an integer |

The Unary + Operator

The **unary + operator** can be used to convert a variable to a number:

Example

let y = "5";      // y is a string  
let x = + y;      // x is a number

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_typeof3)

If the variable cannot be converted, it will still become a number, but with the value NaN (Not a Number):

Example

let y = "John";   // y is a string  
let x = + y;      // x is a number (NaN)

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_typeof4)

ADVERTISEMENT

Converting Numbers to Strings

The global method String() can convert numbers to strings.

It can be used on any type of numbers, literals, variables, or expressions:

Example

String(x)         // returns a string from a number variable x  
String(123)       // returns a string from a number literal 123  
String(100 + 23)  // returns a string from a number from an expression

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_number_string)

The Number method toString() does the same.

Example

x.toString()  
(123).toString()  
(100 + 23).toString()

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_number_tostring)

More Methods

In the chapter [Number Methods](https://www.w3schools.com/js/js_number_methods.asp), you will find more methods that can be used to convert numbers to strings:

|  |  |
| --- | --- |
| **Method** | **Description** |
| toExponential() | Returns a string, with a number rounded and written using exponential notation. |
| toFixed() | Returns a string, with a number rounded and written with a specified number of decimals. |
| toPrecision() | Returns a string, with a number written with a specified length |

Converting Dates to Numbers

The global method Number() can be used to convert dates to numbers.

d = new Date();  
Number(d)          // returns 1404568027739

The date method getTime() does the same.

d = new Date();  
d.getTime()        // returns 1404568027739

Converting Dates to Strings

The global method String() can convert dates to strings.

String(Date())  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)"

The Date method toString() does the same.

Example

Date().toString()  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)"

In the chapter [Date Methods](https://www.w3schools.com/js/js_date_methods.asp), you will find more methods that can be used to convert dates to strings:

|  |  |
| --- | --- |
| **Method** | **Description** |
| getDate() | Get the day as a number (1-31) |
| getDay() | Get the weekday a number (0-6) |
| getFullYear() | Get the four digit year (yyyy) |
| getHours() | Get the hour (0-23) |
| getMilliseconds() | Get the milliseconds (0-999) |
| getMinutes() | Get the minutes (0-59) |
| getMonth() | Get the month (0-11) |
| getSeconds() | Get the seconds (0-59) |
| getTime() | Get the time (milliseconds since January 1, 1970) |

Converting Booleans to Numbers

The global method Number() can also convert booleans to numbers.

Number(false)     // returns 0  
Number(true)      // returns 1

Converting Booleans to Strings

The global method String() can convert booleans to strings.

String(false)      // returns "false"  
String(true)       // returns "true"

The Boolean method toString() does the same.

false.toString()   // returns "false"  
true.toString()    // returns "true"

Automatic Type Conversion

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

The result is not always what you expect:

5 + null    // returns 5         because null is converted to 0  
"5" + null  // returns "5null"   because null is converted to "null"  
"5" + 2     // returns "52"      because 2 is converted to "2"  
"5" - 2     // returns 3         because "5" is converted to 5  
"5" \* "2"   // returns 10        because "5" and "2" are converted to 5 and 2

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_auto)

Automatic String Conversion

JavaScript automatically calls the variable's toString() function when you try to "output" an object or a variable:

document.getElementById("demo").innerHTML = myVar;  
  
// if myVar = {name:"Fjohn"}  // toString converts to "[object Object]"  
// if myVar = [1,2,3,4]       // toString converts to "1,2,3,4"  
// if myVar = new Date()      // toString converts to "Fri Jul 18 2014 09:08:55 GMT+0200"

Numbers and booleans are also converted, but this is not very visible:

// if myVar = 123             // toString converts to "123"  
// if myVar = true            // toString converts to "true"  
// if myVar = false           // toString converts to "false"

JavaScript Type Conversion Table

This table shows the result of converting different JavaScript values to Number, String, and Boolean:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Original Value** | **Converted to Number** | **Converted to String** | **Converted to Boolean** | **Try it** |
| false | 0 | "false" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_false) |
| true | 1 | "true" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_true) |
| 0 | 0 | "0" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_number_0) |
| 1 | 1 | "1" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_number_1) |
| "0" | 0 | "0" | **true** | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_string_0) |
| "000" | 0 | "000" | **true** | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_string_000) |
| "1" | 1 | "1" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_string_1) |
| NaN | NaN | "NaN" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_nan) |
| Infinity | Infinity | "Infinity" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_infinity) |
| -Infinity | -Infinity | "-Infinity" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_infinity_minus) |
| "" | **0** | "" | **false** | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_string_empty) |
| "20" | 20 | "20" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_string_number) |
| "twenty" | NaN | "twenty" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_string_text) |
| [ ] | **0** | "" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_array_empty) |
| [20] | **20** | "20" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_array_one_number) |
| [10,20] | NaN | "10,20" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_array_two_numbers) |
| ["twenty"] | NaN | "twenty" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_array_one_string) |
| ["ten","twenty"] | NaN | "ten,twenty" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_array_two_strings) |
| function(){} | NaN | "function(){}" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_function) |
| { } | NaN | "[object Object]" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_object) |
| null | **0** | "null" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_type_convert_null) |
| undefined | NaN | "undefined" | false |  |

# JavaScript Bitwise Operations

[❮ Previous](https://www.w3schools.com/js/js_type_conversion.asp)[Next ❯](https://www.w3schools.com/js/js_regexp.asp)

## JavaScript Bitwise Operators

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Description** |
| & | AND | Sets each bit to 1 if both bits are 1 |
| | | OR | Sets each bit to 1 if one of two bits is 1 |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 |
| ~ | NOT | Inverts all the bits |
| << | Zero fill left shift | Shifts left by pushing zeros in from the right and let the leftmost bits fall off |
| >> | Signed right shift | Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off |
| >>> | Zero fill right shift | Shifts right by pushing zeros in from the left, and let the rightmost bits fall off |

## Examples

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Result** | **Same as** | **Result** |
| 5 & 1 | 1 | 0101 & 0001 | 0001 |
| 5 | 1 | 5 | 0101 | 0001 | 0101 |
| ~ 5 | 10 | ~0101 | 1010 |
| 5 << 1 | 10 | 0101 << 1 | 1010 |
| 5 ^ 1 | 4 | 0101 ^ 0001 | 0100 |
| 5 >> 1 | 2 | 0101 >> 1 | 0010 |
| 5 >>> 1 | 2 | 0101 >>> 1 | 0010 |

## JavaScript Uses 32 bits Bitwise Operands

JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers.

Before a bitwise operation is performed, JavaScript converts numbers to 32 bits signed integers.

After the bitwise operation is performed, the result is converted back to 64 bits JavaScript numbers.

The examples above uses 4 bits unsigned binary numbers. Because of this ~ 5 returns 10.

Since JavaScript uses 32 bits signed integers, it will not return 10. It will return -6.

00000000000000000000000000000101 (5)

11111111111111111111111111111010 (~5 = -6)

A signed integer uses the leftmost bit as the minus sign.

ADVERTISEMENT

## Bitwise AND

When a bitwise AND is performed on a pair of bits, it returns 1 if both bits are 1.

One bit example:

|  |  |
| --- | --- |
| **Operation** | **Result** |
| 0 & 0 | 0 |
| 0 & 1 | 0 |
| 1 & 0 | 0 |
| 1 & 1 | 1 |

4 bits example:

|  |  |
| --- | --- |
| **Operation** | **Result** |
| 1111 & 0000 | 0000 |
| 1111 & 0001 | 0001 |
| 1111 & 0010 | 0010 |
| 1111 & 0100 | 0100 |

## Bitwise OR

When a bitwise OR is performed on a pair of bits, it returns 1 if one of the bits are 1:

One bit example:

|  |  |
| --- | --- |
| **Operation** | **Result** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

4 bits example:

|  |  |
| --- | --- |
| **Operation** | **Result** |
| 1111 | 0000 | 1111 |
| 1111 | 0001 | 1111 |
| 1111 | 0010 | 1111 |
| 1111 | 0100 | 1111 |

## Bitwise XOR

When a bitwise XOR is performed on a pair of bits, it returns 1 if the bits are different:

One bit example:

|  |  |
| --- | --- |
| **Operation** | **Result** |
| 0 ^ 0 | 0 |
| 0 ^ 1 | 1 |
| 1 ^ 0 | 1 |
| 1 ^ 1 | 0 |

4 bits example:

|  |  |
| --- | --- |
| **Operation** | **Result** |
| 1111 ^ 0000 | 1111 |
| 1111 ^ 0001 | 1110 |
| 1111 ^ 0010 | 1101 |
| 1111 ^ 0100 | 1011 |

## JavaScript Bitwise AND (&)

Bitwise AND returns 1 only if both bits are 1:

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
| 5 | 00000000000000000000000000000101 |
| 1 | 00000000000000000000000000000001 |
| 5 & 1 | 00000000000000000000000000000001 (1) |

### Example

let x = 5 & 1;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_and)

## JavaScript Bitwise OR (|)

Bitwise OR returns 1 if one of the bits are 1:

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
| 5 | 00000000000000000000000000000101 |
| 1 | 00000000000000000000000000000001 |
| 5 | 1 | 00000000000000000000000000000101 (5) |

### Example

let x = 5 | 1;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_or)

## JavaScript Bitwise XOR (^)

Bitwise XOR returns 1 if the bits are different:

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
| 5 | 00000000000000000000000000000101 |
| 1 | 00000000000000000000000000000001 |
| 5 ^ 1 | 00000000000000000000000000000100 (4) |

### Example

let x = 5 ^ 1;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_xor)

## JavaScript Bitwise NOT (~)

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
| 5 | 00000000000000000000000000000101 |
| ~5 | 11111111111111111111111111111010 (-6) |

### Example

let x = ~5;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_not)

## JavaScript (Zero Fill) Bitwise Left Shift (<<)

This is a zero fill left shift. One or more zero bits are pushed in from the right, and the leftmost bits fall off:

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
| 5 | 00000000000000000000000000000101 |
| 5 << 1 | 00000000000000000000000000001010 (10) |

### Example

let x = 5 << 1;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_left)

## JavaScript (Sign Preserving) Bitwise Right Shift (>>)

This is a sign preserving right shift. Copies of the leftmost bit are pushed in from the left, and the rightmost bits fall off:

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
| -5 | 11111111111111111111111111111011 |
| -5 >> 1 | 11111111111111111111111111111101 (-3) |

### Example

let x = -5 >> 1;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_right_sign)

## JavaScript (Zero Fill) Right Shift (>>>)

This is a zero fill right shift. One or more zero bits are pushed in from the left, and the rightmost bits fall off:

|  |  |
| --- | --- |
| **Decimal** | **Binary** |
| 5 | 00000000000000000000000000000101 |
| 5 >>> 1 | 00000000000000000000000000000010 (2) |

### Example

let x = 5 >>> 1;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_right)

## Binary Numbers

Binary numbers with only one bit set is easy to understand:

|  |  |
| --- | --- |
| **Binary Representation** | **Decimal value** |
| 00000000000000000000000000000001 | 1 |
| 00000000000000000000000000000010 | 2 |
| 00000000000000000000000000000100 | 4 |
| 00000000000000000000000000001000 | 8 |
| 00000000000000000000000000010000 | 16 |
| 00000000000000000000000000100000 | 32 |
| 00000000000000000000000001000000 | 64 |

Setting a few more bits reveals the binary pattern:

|  |  |
| --- | --- |
| **Binary Representation** | **Decimal value** |
| 00000000000000000000000000000101 | 5 (4 + 1) |
| 00000000000000000000000000001101 | 13 (8 + 4 + 1) |
| 00000000000000000000000000101101 | 45 (32 + 8 + 4 + 1) |

JavaScript binary numbers are stored in two's complement format.

This means that a negative number is the bitwise NOT of the number plus 1:

|  |  |
| --- | --- |
| **Binary Representation** | **Decimal value** |
| 00000000000000000000000000000101 | 5 |
| 11111111111111111111111111111011 | -5 |
| 00000000000000000000000000000110 | 6 |
| 11111111111111111111111111111010 | -6 |
| 00000000000000000000000000101000 | 40 |
| 11111111111111111111111111011000 | -40 |

## Converting Decimal to Binary

### Example

function dec2bin(dec){  
  return (dec >>> 0).toString(2);  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_convert)

## Converting Binary to Decimal

### Example

function bin2dec(bin){  
  return parseInt(bin, 2).toString(10);  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_bitwise_convert_binary)

# JavaScript Regular Expressions

[❮ Previous](https://www.w3schools.com/js/js_bitwise.asp)[Next ❯](https://www.w3schools.com/js/js_errors.asp)

A regular expression is a sequence of characters that forms a search pattern.

The search pattern can be used for text search and text replace operations.

## What Is a Regular Expression?

A regular expression is a sequence of characters that forms a **search pattern**.

When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of **text search** and **text replace** operations.

## Syntax

/pattern/modifiers;

### Example

/w3schools/i;

Example explained:

**/w3schools/i**  is a regular expression.

**w3schools**  is a pattern (to be used in a search).

**i**  is a modifier (modifies the search to be case-insensitive).

## Using String Methods

In JavaScript, regular expressions are often used with the two **string methods**: search() and replace().

The search() method uses an expression to search for a match, and returns the position of the match.

The replace() method returns a modified string where the pattern is replaced.

## Using String search() With a String

The search() method searches a string for a specified value and returns the position of the match:

### Example

Use a string to do a search for "W3schools" in a string:

let text = "Visit W3Schools!";  
let n = text.search("W3Schools");

The result in n will be:

6

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_string_search)

## Using String search() With a Regular Expression

### Example

Use a regular expression to do a case-insensitive search for "w3schools" in a string:

let text = "Visit W3Schools";  
let n = text.search(/w3schools/i);

The result in n will be:

6

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_string_search)

ADVERTISEMENT

## Using String replace() With a String

The replace() method replaces a specified value with another value in a string:

let text = "Visit Microsoft!";  
let result = text.replace("Microsoft", "W3Schools");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_string_replace)

## Use String replace() With a Regular Expression

### Example

Use a case insensitive regular expression to replace Microsoft with W3Schools in a string:

let text = "Visit Microsoft!";  
let result = text.replace(/microsoft/i, "W3Schools");

The result in res will be:

Visit W3Schools!

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_string_replace)

## Did You Notice?

Regular expression arguments (instead of string arguments) can be used in the methods above.  
Regular expressions can make your search much more powerful (case insensitive for example).

## Regular Expression Modifiers

**Modifiers** can be used to perform case-insensitive more global searches:

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** | **Try it** |
| i | Perform case-insensitive matching | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_i) |
| g | Perform a global match (find all matches rather than stopping after the first match) | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_g) |
| m | Perform multiline matching | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_m) |

## Regular Expression Patterns

**Brackets** are used to find a range of characters:

|  |  |  |
| --- | --- | --- |
| **Expression** | **Description** | **Try it** |
| [abc] | Find any of the characters between the brackets | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_abc) |
| [0-9] | Find any of the digits between the brackets | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_0-9) |
| (x|y) | Find any of the alternatives separated with | | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_xy) |

**Metacharacters** are characters with a special meaning:

|  |  |  |
| --- | --- | --- |
| **Metacharacter** | **Description** | **Try it** |
| \d | Find a digit | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_d) |
| \s | Find a whitespace character | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_s) |
| \b | Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_b) [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_b2) |
| \uxxxx | Find the Unicode character specified by the hexadecimal number xxxx | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_ux) |

**Quantifiers** define quantities:

|  |  |  |
| --- | --- | --- |
| **Quantifier** | **Description** | **Try it** |
| n+ | Matches any string that contains at least one n | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n1) |
| n\* | Matches any string that contains zero or more occurrences of n | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n2) |
| n? | Matches any string that contains zero or one occurrences of n | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_n3) |

## Using the RegExp Object

In JavaScript, the RegExp object is a regular expression object with predefined properties and methods.

## Using test()

The test() method is a RegExp expression method.

It searches a string for a pattern, and returns true or false, depending on the result.

The following example searches a string for the character "e":

### Example

const pattern = /e/;  
pattern.test("The best things in life are free!");

Since there is an "e" in the string, the output of the code above will be:

true

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_regexp_test)

You don't have to put the regular expression in a variable first. The two lines above can be shortened to one:

/e/.test("The best things in life are free!");

## Using exec()

The exec() method is a RegExp expression method.

It searches a string for a specified pattern, and returns the found text as an object.

If no match is found, it returns an empty *(null)* object.

The following example searches a string for the character "e":

### Example

/e/.exec("The best things in life are free!");

# JavaScript Errors

[❮ Previous](https://www.w3schools.com/js/js_regexp.asp)[Next ❯](https://www.w3schools.com/js/js_scope.asp)

## Throw, and Try...Catch...Finally

The try statement defines a code block to run (to try).

The catch statement defines a code block to handle any error.

The finally statement defines a code block to run regardless of the result.

The throw statement defines a custom error.

## Errors Will Happen!

When executing JavaScript code, different errors can occur.

Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

### Example

In this example we misspelled "alert" as "adddlert" to deliberately produce an error:

<p id="demo"></p>  
  
<script>  
try {  
  adddlert("Welcome guest!");  
}  
catch(err) {  
  document.getElementById("demo").innerHTML = err.message;  
}  
</script>

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_try_catch)

JavaScript catches **adddlert** as an error, and executes the catch code to handle it.

## JavaScript try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The JavaScript statements try and catch come in pairs:

try {  
  Block of code to try}  
catch(err) {  
  Block of code to handle errors}

ADVERTISEMENT

## JavaScript Throws Errors

When an error occurs, JavaScript will normally stop and generate an error message.

The technical term for this is: JavaScript will **throw an exception (throw an error)**.

JavaScript will actually create an **Error object** with two properties: **name** and **message**.

## The throw Statement

The throw statement allows you to create a custom error.

Technically you can **throw an exception (throw an error)**.

The exception can be a JavaScript String, a Number, a Boolean or an Object:

throw "Too big";    // throw a text  
throw 500;          // throw a number

If you use throw together with try and catch, you can control program flow and generate custom error messages.

## Input Validation Example

This example examines input. If the value is wrong, an exception (err) is thrown.

The exception (err) is caught by the catch statement and a custom error message is displayed:

<!DOCTYPE html>  
<html>  
<body>  
  
<p>Please input a number between 5 and 10:</p>  
  
<input id="demo" type="text">  
<button type="button" onclick="myFunction()">Test Input</button>  
<p id="p01"></p>  
  
<script>  
function myFunction() {  
  const message = document.getElementById("p01");  
  message.innerHTML = "";  
  let x = document.getElementById("demo").value;  
  try {  
    if(x == "") throw "empty";  
    if(isNaN(x)) throw "not a number";  
    x = Number(x);  
    if(x < 5) throw "too low";  
    if(x > 10) throw "too high";  
  }  
  catch(err) {  
    message.innerHTML = "Input is " + err;  
  }  
}  
</script>  
  
</body>  
</html>

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_throw_error)

## HTML Validation

The code above is just an example.

Modern browsers will often use a combination of JavaScript and built-in HTML validation, using predefined validation rules defined in HTML attributes:

<input id="demo" type="number" min="5" max="10" step="1">

You can read more about forms validation in a later chapter of this tutorial.

## The finally Statement

The finally statement lets you execute code, after try and catch, regardless of the result:

### Syntax

try {  
  Block of code to *try*}  
catch(err) {  
  Block of code to handle errors}  
finally {  
  Block of code to be executed regardless of the *try* / *catch* result}

### Example

function myFunction() {  
  const message = document.getElementById("p01");  
  message.innerHTML = "";  
  let x = document.getElementById("demo").value;  
  try {  
    if(x == "") throw "is empty";  
    if(isNaN(x)) throw "is not a number";  
    x = Number(x);  
    if(x > 10) throw "is too high";  
    if(x < 5) throw "is too low";  
  }  
  catch(err) {  
    message.innerHTML = "Error: " + err + ".";  
  }  
  finally {  
    document.getElementById("demo").value = "";  
  }  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_finally_error)

## The Error Object

JavaScript has a built in error object that provides error information when an error occurs.

The error object provides two useful properties: name and message.

## Error Object Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| name | Sets or returns an error name |
| message | Sets or returns an error message (a string) |

## Error Name Values

Six different values can be returned by the error name property:

|  |  |
| --- | --- |
| **Error Name** | **Description** |
| EvalError | An error has occurred in the eval() function |
| RangeError | A number "out of range" has occurred |
| ReferenceError | An illegal reference has occurred |
| SyntaxError | A syntax error has occurred |
| TypeError | A type error has occurred |
| URIError | An error in encodeURI() has occurred |

The six different values are described below.

## Eval Error

An EvalError indicates an error in the eval() function.

Newer versions of JavaScript do not throw EvalError. Use SyntaxError instead.

## Range Error

A RangeError is thrown if you use a number that is outside the range of legal values.

For example: You cannot set the number of significant digits of a number to 500.

### Example

let num = 1;  
try {  
  num.toPrecision(500);   // A number cannot have 500 significant digits  
}  
catch(err) {  
  document.getElementById("demo").innerHTML = err.name;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_rangeerror)

## Reference Error

A ReferenceError is thrown if you use (reference) a variable that has not been declared:

### Example

let x = 5;  
try {  
  x = y + 1;   // y cannot be used (referenced)  
}  
catch(err) {  
  document.getElementById("demo").innerHTML = err.name;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_referenceerror)

## Syntax Error

A SyntaxError is thrown if you try to evaluate code with a syntax error.

### Example

try {  
  eval("alert('Hello)");   // Missing ' will produce an error  
}  
catch(err) {  
  document.getElementById("demo").innerHTML = err.name;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_syntaxerror)

## Type Error

A TypeError is thrown if you use a value that is outside the range of expected types:

### Example

let num = 1;  
try {  
  num.toUpperCase();   // You cannot convert a number to upper case  
}  
catch(err) {  
  document.getElementById("demo").innerHTML = err.name;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_error_typeerror)

## URI (Uniform Resource Identifier) Error

A URIError is thrown if you use illegal characters in a URI function:

### Example

try {  
  decodeURI("%%%");   // You cannot URI decode percent signs  
}  
catch(err) {  
  document.getElementById("demo").innerHTML = err.name;  
}

JavaScript Scope

[❮ Previous](https://www.w3schools.com/js/js_errors.asp)[Next ❯](https://www.w3schools.com/js/js_hoisting.asp)

Scope determines the accessibility (visibility) of variables.

JavaScript has 3 types of scope:

* Block scope
* Function scope
* Global scope

Block Scope

Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope**.

ES6 introduced two important new JavaScript keywords: let and const.

These two keywords provide **Block Scope** in JavaScript.

Variables declared inside a { } block cannot be accessed from outside the block:

Example

{  
  let x = 2;  
}  
// x can NOT be used here

Variables declared with the var keyword can NOT have block scope.

Variables declared inside a { } block can be accessed from outside the block.

Example

{  
  var x = 2;  
}  
// x CAN be used here

Local Scope

Variables declared within a JavaScript function, become **LOCAL** to the function.

Example

// code here can NOT use carName  
  
function myFunction() {  
  let carName = "Volvo";  
  // code here CAN use carName  
}  
  
// code here can NOT use carName

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_scope_local)

Local variables have **Function Scope**:

They can only be accessed from within the function.

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

Function Scope

JavaScript has function scope: Each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Variables declared with var, let and const are quite similar when declared inside a function.

They all have **Function Scope**:

function myFunction() {  
  var carName = "Volvo";   // Function Scope  
}

function myFunction() {  
  let carName = "Volvo";   // Function Scope  
}

function myFunction() {  
  const carName = "Volvo";   // Function Scope  
}

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

Example

let carName = "Volvo";  
// code here can use carName  
  
function myFunction() {  
// code here can also use carName  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_scope_global)

A global variable has **Global Scope**:

All scripts and functions on a web page can access it.

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

**Global** variables can be accessed from anywhere in a JavaScript program.

Variables declared with var, let and const are quite similar when declared outside a block.

They all have **Global Scope**:

var x = 2;       // Global scope

let x = 2;       // Global scope

const x = 2;       // Global scope

JavaScript Variables

In JavaScript, objects and functions are also variables.

Scope determines the accessibility of variables, objects, and functions from different parts of the code.

ADVERTISEMENT

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable carName, even if the value is assigned inside a function.

Example

myFunction();  
  
// code here can use carName  
  
function myFunction() {  
  carName = "Volvo";  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_local_global)

Strict Mode

All modern browsers support running JavaScript in "Strict Mode".

You will learn more about how to use strict mode in a later chapter of this tutorial.

In "Strict Mode", undeclared variables are not automatically global.

Global Variables in HTML

With JavaScript, the global scope is the JavaScript environment.

In HTML, the global scope is the window object.

Global variables defined with the var keyword belong to the window object:

Example

var carName = "Volvo";  
// code here can use window.carName

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_scope_window)

Global variables defined with the let keyword do not belong to the window object:

Example

let carName = "Volvo";  
// code here can not use window.carName

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_let_global)

Warning

Do NOT create global variables unless you intend to.

Your global variables (or functions) can overwrite window variables (or functions).  
Any function, including the window object, can overwrite your global variables and functions.

The Lifetime of JavaScript Variables

The lifetime of a JavaScript variable starts when it is declared.

Function (local) variables are deleted when the function is completed.

In a web browser, global variables are deleted when you close the browser window (or tab).

JavaScript Use Strict

[❮ Previous](https://www.w3schools.com/js/js_hoisting.asp)[Next ❯](https://www.w3schools.com/js/js_this.asp)

"use strict"; Defines that JavaScript code should be executed in "strict mode".

The "use strict" Directive

The "use strict" directive was new in ECMAScript version 5.

It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.

The purpose of "use strict" is to indicate that the code should be executed in "strict mode".

With strict mode, you can not, for example, use undeclared variables.

All modern browsers support "use strict" except Internet Explorer 9 and lower:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Directive |  |  |  |  |  |
| "use strict" | 13.0 | 10.0 | 4.0 | 6.0 | 12.1 |

The numbers in the table specify the first browser version that fully supports the directive.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string, so IE 9 will not throw an error even if it does not understand it.

Declaring Strict Mode

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

Example

"use strict";  
x = 3.14;       // This will cause an error because x is not declared

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_variable)

Example

"use strict";  
myFunction();  
  
function myFunction() {  
  y = 3.14;   // This will also cause an error because y is not declared  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_global)

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

x = 3.14;       // This will not cause an error.  
myFunction();  
  
function myFunction() {  
  "use strict";  
  y = 3.14;   // This will cause an error  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_local)

The "use strict"; Syntax

The syntax, for declaring strict mode, was designed to be compatible with older versions of JavaScript.

Compiling a numeric literal (4 + 5;) or a string literal ("John Doe";) in a JavaScript program has no side effects. It simply compiles to a non existing variable and dies.

So "use strict"; only matters to new compilers that "understand" the meaning of it.

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.

In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

Not Allowed in Strict Mode

Using a variable, without declaring it, is not allowed:

"use strict";  
x = 3.14;                // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_variable)

Objects are variables too.

Using an object, without declaring it, is not allowed:

"use strict";  
x = {p1:10, p2:20};      // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_object)

Deleting a variable (or object) is not allowed.

"use strict";  
let x = 3.14;  
delete x;                // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_delete)

Deleting a function is not allowed.

"use strict";  
function x(p1, p2) {};  
delete x;                // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_delete_function)

Duplicating a parameter name is not allowed:

"use strict";  
function x(p1, p1) {};   // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_duplicate)

Octal numeric literals are not allowed:

"use strict";  
let x = 010;             // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_octal)

Octal escape characters are not allowed:

"use strict";  
let x = "\010";            // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_escape)

Writing to a read-only property is not allowed:

"use strict";  
const obj = {};  
Object.defineProperty(obj, "x", {value:0, writable:false});  
  
obj.x = 3.14;            // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_readonly)

Writing to a get-only property is not allowed:

"use strict";  
const obj = {get x() {return 0} };  
  
obj.x = 3.14;            // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_getonly)

Deleting an undeletable property is not allowed:

"use strict";  
delete Object.prototype; // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_deleteprop)

The word eval cannot be used as a variable:

"use strict";  
let eval = 3.14;         // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_eval)

The word arguments cannot be used as a variable:

"use strict";  
let arguments = 3.14;    // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_arguments)

The with statement is not allowed:

"use strict";  
with (Math){x = cos(2)}; // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_width)

For security reasons, eval() is not allowed to create variables in the scope from which it was called:

"use strict";  
eval ("let x = 2");  
alert (x);             // This will cause an error

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_eval2)

The this keyword in functions behaves differently in strict mode.

The this keyword refers to the object that called the function.

If the object is not specified, functions in strict mode will return undefined and functions in normal mode will return the global object (window):

"use strict";  
function myFunction() {  
  alert(this); // will alert "undefined"  
}  
myFunction();

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_strict_this)

Future Proof!

Keywords reserved for future JavaScript versions can NOT be used as variable names in strict mode.

These are:

* implements
* interface
* let
* package
* private
* protected
* public
* static
* yield

"use strict";  
let public = 1500;      // This will cause an error

# The JavaScript this Keyword

[❮ Previous](https://www.w3schools.com/js/js_strict.asp)[Next ❯](https://www.w3schools.com/js/js_arrow_function.asp)

### Example

const **person** = {  
  firstName: "John",  
  lastName : "Doe",  
  id       : 5566,  
  fullName : function() {  
    return **this**.firstName + " " + **this**.lastName;  
  }  
};

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_method)

## What is this?

In JavaScript, the this keyword refers to an **object**.

**Which** object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

|  |
| --- |
| In an object method, this refers to the **object**. |
| Alone, this refers to the **global object**. |
| In a function, this refers to the **global object**. |
| In a function, in strict mode, this is undefined. |
| In an event, this refers to the **element** that received the event. |
| Methods like call(), apply(), and bind() can refer this to **any object**. |

## Note

this is not a variable. It is a keyword. You cannot change the value of this.

## this in a Method

When used in an object method, this refers to the **object**.

In the example on top of this page, this refers to the **person** object.

Because the **fullName** method is a method of the **person** object.

fullName : function() {  
  return **this**.firstName + " " + **this**.lastName;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_method)

## this Alone

When used alone, this refers to the **global object**.

Because this is running in the global scope.

In a browser window the global object is [object Window]:

### Example

let x = this;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this)

 In **strict mode**, when used alone, this also refers to the **global object**:

### Example

"use strict";  
let x = this;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_alone)

## this in a Function (Default)

In a function, the **global object** is the default binding for this.

In a browser window the global object is [object Window]:

### Example

function myFunction() {  
  return this;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_function)

ADVERTISEMENT

## this in a Function (Strict)

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, this is undefined.

### Example

"use strict";  
function myFunction() {  
  return this;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_strict)

## this in Event Handlers

In HTML event handlers, this refers to the HTML element that received the event:

### Example

<button onclick="this.style.display='none'">  
  Click to Remove Me!  
</button>

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_event)

## Object Method Binding

In these examples, this is the **person object**:

### Example

const **person** = {  
  firstName  : "John",  
  lastName   : "Doe",  
  id         : 5566,  
  myFunction : function() {  
    return **this**;  
  }  
};

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_object)

### Example

const **person** = {  
  firstName: "John",  
  lastName : "Doe",  
  id       : 5566,  
  fullName : function() {  
    return **this**.firstName + " " + **this**.lastName;  
  }  
};

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_method)

i.e. **this.firstName** is the **firstName** property of **this** (the person object).

## Explicit Function Binding

The call() and apply() methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument.

## See Also:

[The Function call() Method](https://www.w3schools.com/js/js_function_call.asp)

[The Function apply() Method](https://www.w3schools.com/js/js_function_apply.asp)

[The Function bind() Method](https://www.w3schools.com/js/js_function_bind.asp)

The example below calls person1.fullName with person2 as an argument, **this** refers to person2, even if fullName is a method of person1:

### Example

const person1 = {  
  fullName: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
  
const person2 = {  
  firstName:"John",  
  lastName: "Doe",  
}  
  
// Return "John Doe":  
person1.fullName.call(person2);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_this_call)

## Function Borrowing

With the bind() method, an object can borrow a method from another object.

This example creates 2 objects (person and member).

The member object borrows the fullname method from the person object:

### Example

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  }  
}  
  
const member = {  
  firstName:"Hege",  
  lastName: "Nilsen",  
}  
  
let fullName = person.fullName.bind(member);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_bind_borrow)

## This Precedence

To determine which object this refers to; Use the following precedence of order.

|  |
| --- |
|  |
| Precedence | Object |
| 1 | bind() |
| 2 | apply() and call() |
| 3 | Object method |
| 4 | Global scope |