Resumen de Javascript

Semana 4

String.prototype.indexOf()

El **`indexOf()`** método devuelve el índice, dentro del objeto `String` que realiza la llamada, de la primera ocurrencia del valor especificado, comenzando la búsqueda desde`indiceBusqueda`; o -1 si no se encuentra dicho valor.

Sintaxis

```
cadena.indexOf(valorBusqueda[, indiceDesde]);
```

Parámetros

`valorBusqueda`

Una cadena que representa el valor de búsqueda.

`indiceDesde`

La localización dentro de la cadena llamada desde la que empezará la búsqueda. Puede ser un entero entre 0 y la longitud de la cadena. El valor predeterminado es 0.

DESCRIPCIÓN

Los caracteres de una cadena se indexan de izquierda a derecha. El índice del primer carácter es 0, y el índice del último carácter de una cadena llamada `nombreCadena` es`nombreCadena.length - 1`.

```
"Blue Whale".indexOf("Blue")    // returns 0
"Blue Whale".indexOf("Blute")   // returns -1
"Blue Whale".indexOf("Whale",0) // returns 5
"Blue Whale".indexOf("Whale",5) // returns 5
"Blue Whale".indexOf("",9)      // returns 9
"Blue Whale".indexOf("",10)     // returns 10
"Blue Whale".indexOf("",11)     // returns 10
```

El método `indexOf` es sensible a mayúsculas. Por ejemplo, la siguiente expresión devuelve -1:

```
"Ballena Azul".indexOf("azul");
```

EJEMPLOS

```
var cualquierCadena="Brave new world"


document.write("<P>The index of the first w from the beginning is " +
   cualquierCadena.indexOf("w"))        // Muestra 8

document.write("<P>The index of the first w from the end is " +
   cualquierCadena.lastIndexOf("w"))   // Muestra 10

document.write("<P>The index of 'new' from the beginning is " +
   cualquierCadena.indexOf("new"))     // Muestra 6

document.write("<P>The index of 'new' from the end is " +
   cualquierCadena.lastIndexOf("new")) // Muestra 6
```

Ejemplo: Usando indexOf para contar ocurrencias de una letra en una cadena

```
cuenta = 0;
posicion = miCadena.indexOf("x");
while ( posicion != -1 ) {
    cuenta++;
    posicion = miCadena.indexOf("x",posicion+1);
}
```

HOISTING

JavaScript Declarations are Hoisted

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

```
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element

var x; // Declare x
```

To understand this, you have to understand the term "hoisting".

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

JavaScript Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

```
var x = 5; // Initialize x
var y = 7; // Initialize y

elem = document.getElementById("demo"); // Find an element

elem.innerHTML = x + " " + y;           // Display x and y
-----------------------------------------------------------
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y


var y = 7; // Initialize y
```

Does it make sense that y is undefined in the last example?

This is because only the declaration (var y), not the initialization (=7) is hoisted to the top.

Because of hoisting, y has been declared before it is used, but because initializations are not hoisted, the value of y is undefined.

```
var x = 5; // Initialize x
var y;     // Declare y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

y = 7;    // Assign 7 to y
```

JavaScript Use Strict

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

The "use strict" Directive

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.

Declaring Strict Mode

```
"use strict";
x = 3.14;       // This will cause an error
myFunction();   // This will also cause an error

function myFunction() {
      x = 3.14;
}
---------------------------------------------------
x = 3.14;       // This will not cause an error.
myFunction();   // This will cause an error

function myFunction() {
   "use strict";
      x = 3.14;
}
```

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?

1. Strict mode makes it easier to write "secure" JavaScript.
2. Strict mode changes previously accepted "bad syntax" into real errors.
3. 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.
4. In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.
5. 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 (property or object) without declaring it, is not allowed:

```
"use strict";
x = 3.14;                  // This will cause an error (if x has not been declared)
```

Deleting a variable, a function, or an argument, is not allowed.

```
"use strict";

x = 3.14;

delete x;                  // This will cause an error
```

Defining a property more than once, is not allowed:

```
"use strict";

var x = {p1:10, p1:20};   // This will cause an error
```

Duplicating a parameter name is not allowed:

```
"use strict";

function x(p1, p1) {};    // This will cause an error
```

Octal numeric literals and escape characters are not allowed:

```
"use strict";

var x = 010;             // This will cause an error

var y = \010;            // This will cause an error
```

Writing to a read-only property is not allowed:

```
"use strict";

var obj = {};

obj.defineProperty(obj, "x", {value:0, writable:false});


obj.x = 3.14;             // This will cause an error
```

Writing to a get-only property is not allowed:

```
"use strict";
var obj = {get x() {return 0} };

obj.x = 3.14;            // This will cause an error
```

Deleting an undeletable property is not allowed:

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

The string "eval" cannot be used as a variable:

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

The string "arguments" cannot be used as a variable:

```
"use strict";

var arguments = 3.14;    // This will cause an error
```

The with statement is not allowed:

```
"use strict";

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

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

```
"use strict";

eval ("var x = 2");

alert (x)                // This will cause an error
```

In function calls like f(), the this value was the global object. In strict mode, it is now undefined.

Future reserved keywords are not allowed. These are:

- implements
- interface
- package
- private
- protected
- public
- static
- yield

JavaScript Common Mistakes

Accidentally Using the Assignment Operator

JavaScript programs may generate unexpected results if a programmer accidentally uses an assignment operator (=), instead of a comparison operator (==) in an if statement.

This if statement returns false (as expected) because x is not equal to 10:

```
var x = 0;

if (x == 10)
```

This if statement returns true (maybe not as expected), because 10 is true:

```
var x = 0;

if (x = 10)
```

This if statement returns false (maybe not as expected), because 0 is false:

```
var x = 0;

if (x = 0)
```

Expecting Loosely Comparison

In regular comparison, data type does not matter. This if statement returns true:

```
var x = 10;

var y = "10";

if (x == y)
```

In strict comparison, data type does matter. This if statement returns false:

```
var x = 10;

var y = "10";

if (x === y)
```

It is a common mistake to forget that switch statements use strict comparison:

This case switch will display an alert:

```
var x = 10;

switch(x) {

    case 10: alert("Hello");

}
```

This case switch will not display an alert:

```
var x = 10;

switch(x) {

    case "10": alert("Hello");

}
```

Confusing Addition & Concatenation

Addition is about adding numbers.

Concatenation is about adding strings.

In JavaScript both operations use the same + operator.

Because of this, when adding a number as a number, will produce a different result from adding a number as a string:

```
var x = 10 + 5;          // the result in x is 15
var x = 10 + "5";        // the result in x is "105"
```

When adding two variables, it can be difficult to anticipate the result:

```
var x = 10;
var y = 5;
var z = x + y;            // the result in z is 15


var x = 10;
var y = "5";
var z = x + y;           // the result in z is "105"
```

Misunderstanding Floats

All numbers in JavaScript are stored as 64-bits Floating point numbers (Floats).

All programming languages, including JavaScript, have difficulties with precise floating point values:

```
var x = 0.1;

var y = 0.2;

var z = x + y             // the result in z will not be 0.3

if (z == 0.3)             // this if test will fail
```

Breaking a JavaScript String

JavaScript will allow you to break a statement into two lines:

```
var x =

"Hello World!";
```

But, breaking a statement in the middle of a string will not work:

```
var x = "Hello

World!";
```

You must use a "backslash" if you must break a statement in a string:

```
var x = "Hello \

World!";
```

Misplacing Semicolon

Because of a misplaced semicolon, this code block will execute regardless of the value of x:

```
if (x == 19);
{
    // code block
}
```

Breaking a Return Statement

It is a default JavaScript behavior to close a statement automatically at the end of a line.

Because of this, these two examples will return the same result:

Example 1

```
function myFunction(a) {

    var power = 10

    return a * power

}
```

Example 2

```
function myFunction(a) {
    var power = 10;
    return a * power;
}
```

JavaScript will also allow you to break a statement into two lines.

Because of this, example 3 will also return the same result:

```
function myFunction(a) {
    var
      power = 10;
    return a * power;
}
```

But, what will happen if you break the return statement in two lines like this:

```
function myFunction(a) {
    var
      power = 10;
    return
      a * power;
}
```

The function will return undefined!

Why? Because JavaScript thinks you meant:

```
function myFunction(a) {
    var
      power = 10;
    return;
      a * power;
}
```

Explanation

If a statement is incomplete like:

```
var
```

JavaScript will try to complete the statement by reading the next line:

```
power = 10;
```

But since this statement is complete:

```
return
```

JavaScript will automatically close it like this:

```
return;
```

This happens because closing (ending) statements with semicolon is optional in JavaScript.

JavaScript will close the return statement at the end of the line, because it is a complete statement.

Accessing Arrays with Named Indexes

Many programming languages support arrays with named indexes.

Arrays with named indexes are called associative arrays (or hashes).

JavaScript does not support arrays with named indexes.

In JavaScript, arrays use numbered indexes:

```
var person = [];

person[0] = "John";

person[1] = "Doe";

person[2] = 46;

var x = person.length;         // person.length will return 3

var y = person[0];             // person[0] will return "John"
```

In JavaScript, objects use named indexes.

If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.

After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

Example:

```
var person = [];

person["firstName"] = "John";

person["lastName"] = "Doe";

person["age"] = 46;

var x = person.length;         // person.length will return 0

var y = person[0];             // person[0] will return undefined
```

Ending an Array Definition with a Comma

Incorrect:

```
points = [40, 100, 1, 5, 25, 10,];
```

Some JSON and JavaScript engines will fail, or behave unexpectedly.

Correct:

```
points = [40, 100, 1, 5, 25, 10];
```

Ending an Object Definition with a Comma

Incorrect:

```
person = {firstName:"John", lastName:"Doe", age:46,}
```

Some JSON and JavaScript engines will fail, or behave unexpectedly.

Correct:

```
person = {firstName:"John", lastName:"Doe", age:46}
```

Undefined is Not Null

With JavaScript, null is for objects, undefined is for variables, properties, and methods.

To be null, an object has to be defined, otherwise it will be undefined.

If you want to test if an object exists, this will throw an error if the object is undefined:

Incorrect:

```
if (myObj !== null && typeof myObj !== "undefined")
```

Because of this, you must test typeof() first:

Correct:

```
if (typeof myObj !== "undefined" && myObj !== null)
```

Expecting Block Level Scope

JavaScript does not create a new scope for each code block.

It is true in many programming languages, but not true in JavaScript.

It is a common mistake, among new JavaScript developers, to believe that this code returns undefined:

Example:

```
for (var i = 0; i < 10; i++) {
    // some code
}
return i;
```

JavaScript Coding Conventions

Variable Names

At W3schools we use camelCase for identifier names (variables and functions).

All names start with a letter.

At the bottom of this page, you will find a wider discussion about naming rules.

```
firstName = "John";

lastName = "Doe";


price = 19.90;

tax = 0.20;


fullPrice = price + (price * tax);
```

Spaces Around Operators

Always put spaces around operators (= + / *), and after commas:

Examples:

```
var x = y + z;

var values = ["Volvo", "Saab", "Fiat"];
```

Code Indentation

Always use 4 spaces for indentation of code blocks:

Functions:

```
function toCelsius(fahrenheit) {

    return (5/9) * (fahrenheit-32);

}
```

Statement Rules

General rules for simple statements:

- Always end simple statement with a semicolon.

Examples:

```
var values = ["Volvo", "Saab", "Fiat"];


var person = {

      firstName: "John",

      lastName: "Doe",

      age: 50,

      eyeColor: "blue"

};
```

General rules for complex (compound) statements:

- Put the opening bracket at the end of the first line.
- Use one space before the opening bracket.
- Put the closing bracket on a new line, without leading spaces.
- Do not end complex statement with a semicolon.

Functions:

```
function toCelsius(fahrenheit) {

    return (5/9) * (fahrenheit-32);

}
```

Loops:

```
for (i = 0; i < 5; i++) {

      x += i;

}
```

Conditionals:

```
if (time < 20) {

      greeting = "Good day";

} else {

      greeting = "Good evening";

}
```

Object Rules

General rules for object definitions:

- Place the opening bracket on the same line as the object name.
- Use colon plus one space between each property and it's value.
- Use quotes around string values, not around numeric values.
- Do not add a comma after the last property-value pair.
- Place the closing bracket, on a new line, without leading spaces.
- Always end an object definition with a semicolon.

Example:

```
var person = {

    firstName: "John",

    lastName: "Doe",

    age: 50,

    eyeColor: "blue"

};
```

Line Length < 80

For readability, avoid lines longer than 80 characters.

If a JavaScript statement does not fit on one line, the best place to break it, is after an operator or a comma.

Example

```
document.getElementById("demo").innerHTML =

    "Hello Dolly.";
```

Json

JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

What is JSON?

- JSON stands for JavaScript Object Notation
- JSON is lightweight data interchange format
- JSON is language independent *
- JSON is "self-describing" and easy to understand

JSON Example

```
{"employees":[
      {"firstName":"John", "lastName":"Doe"},
      {"firstName":"Anna",    "lastName":"Smith"},
      {"firstName":"Peter", "lastName":"Jones"}
]}
```

The JSON format is syntactically identical to the code for creating JavaScript objects.

Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects.

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs, Just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName":"John"
```

JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/values pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

JSON Arrays

JSON arrays are written inside square brackets.

Just like in JavaScript, an array can contain objects:

```
"employees":[
      {"firstName":"John", "lastName":"Doe"},
      {"firstName":"Anna", "lastName":"Smith"},
      {"firstName":"Peter", "lastName":"Jones"}
]
```

Converting a JSON Text to a JavaScript Object

First, create a JavaScript string containing JSON syntax:

```
var text = '{ "employees" : [' +

'{ "firstName":"John" , "lastName":"Doe" },' +

'{ "firstName":"Anna" , "lastName":"Smith" },' +

'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

Then, use the JavaScript built-in function JSON.parse() to convert the string into a JavaScript object:

```
var obj = JSON.parse(text);
```

Performance

Reduce Activity in Loops

Bad Code:

```
for i = 0; i < arr.length; i++) {
```

Better Code:

```
l = arr.length;

for i = 0; i < l; i++) {
```

Reduce DOM Access

Example

```
obj = document.getElementById("demo");
obj.innerHTML = "Hello";
```

Reduce DOM Size

Keep the number of elements in the HTML DOM small.

This will always improve page loading, and speed up rendering (page display), especially on smaller devices.

Every attempt to search the DOM (like getElementsByTagName) will benefit from a smaller DOM.

Avoid Unnecessary Variables

Often you can replace code like this:

```
var fullName = firstName + " " + lastName;

document.getElementById("demo").innerHTML = fullName;
```

With this:

```
document.getElementById("demo").innerHTML = firstName + " " + lastName
```

Delay JavaScript Loading

Putting your scripts at the bottom of the page body, lets the browser load the page first.

While a script is downloading, the browser will not start any other downloads. In addition all parsing and rendering activity might be blocked.

An alternative is to use defer="true" in the script tag. The defer attribute specifies that the script should be executed before the page has finished parsing, but it only works for external scripts.

If possible, you can add your script to the page by code, after the page has loaded:

```
<script>

window.onload = downScripts;

function downScripts() {

      var element = document.createElement("script");

      element.src = "myScript.js";

      document.body.appendChild(element);

}

</script>
```

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).

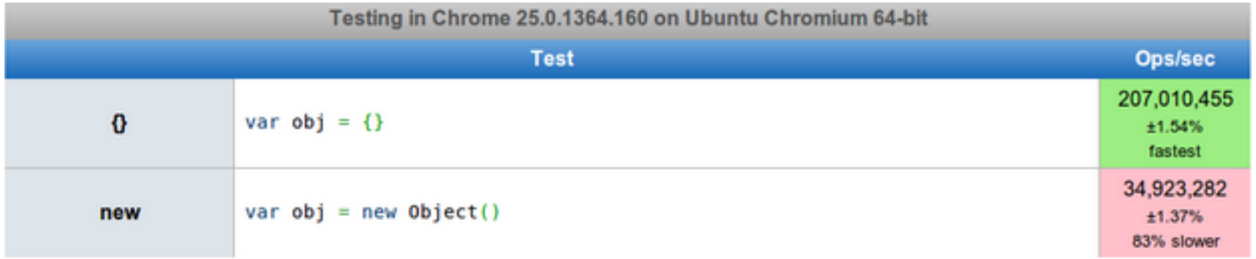
Empty strings convert to 0.

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
```

In the chapter Number Methods, you will find more methods that can be used to convert strings to numbers:

Method	Description
parseFloat()	Parses a string and returns a floating point number
parseInt()	Parses a string and returns an integer



Testing in Chrome 25.0.1364.160 on Ubuntu Chromium 64-bit

	Test	Ops/sec
{}	`var obj = {}`	207,010,455 ±1.54% fastest
new	`var obj = new Object()`	34,923,282 ±1.37% 83% slower

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" + 1     // returns "51"      because 1 is converted to "1"
"5" - 1     // returns 4         because "5" is converted to 5
```

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
```