# 6. Methods

1. Arrays

**array.concat(item...)**

Concat produces a new array containing a shallow copy of *this* array with the *items* appended to it. If an *item* is an array, then each of its elements is appended individually.

var a = ['a', 'b', 'c'];

var b = ['x', 'y', 'z'];

var c = a.concat(b, true);

// c is ['a', 'b', 'c', 'x', 'y', 'z', true]

**array.join(separator)**

Join method makes a string from an array. It does this by making a string of each of the array’s elements, and then concatenating them all together with a separator between them. The default separator is ','. To join without separation, use an empty string as the separator.

var a = ['a', 'b', 'c'];

var c = a.join(''); // c is 'abc';

**array.pop( )**

Pop method removes and returns the last element in this array. If the array is empty, it returns undefined.

var a = ['a', 'b', 'c'];

var c = a.pop(); // a is ['a', 'b'] & c is 'c';

**array.push(item...)**

Push method appends items to the end of an array. Unlike the concat method, it modifies the array and appends array items whole. It returns the new length of the array:

var a = ['a', 'b', 'c'];

var b = ['x', 'y', 'z'];

var c = a.push(b, true);

// a is ['a', 'b', 'c', ['x', 'y', 'z'], true]

// c is 5;

**array.reverse( )**

The reverse method modifies the array by reversing the order of the elements. It returns the array:

var a = ['a', 'b', 'c'];

var b = a.reverse();

// Both **a** and **b** are ['c', 'b', 'a']

**array.shift( )**

The shift method removes the first element from an array and returns it. If the array is empty, it returns *undefined*. Shift is usually much slower than pop:

var a = ['a', 'b', 'c'];

var c = a.shift(); // a is ['b', 'c'] & c is 'a'

**array.slice(start, end)**

Slice makes a shallow copy of a portion of an array. The first element copied will be array[*start*] and It will stop before copying array[*end*]. The end parameter is optional, and the default is array.length. If either parameter is negative, array.length will be added to them in an attempt to make them nonnegative. If start is greater than or equal to array.length, the result will be a new empty array.

var a = ['a', 'b', 'c'];

var b = a.slice(0, 1); // b is ['a']

var c = a.slice(1); // c is ['b', 'c']

var d = a.slice(1, 2); // d is ['b']

var e = a.slice(a.length, a.length); // e is []

**array.sort(comparefn)**

The sort method sorts the contents of an array in place.

var m = ['aa', 'bb', 'a', 4, 8, 15, 16, 23, 42];

m.sort(function (a, b) {

if (a === b) {

return 0;

}

if (typeof a === typeof b) {

return a < b ? -1 : 1;

}

return typeof a < typeof b ? -1 : 1;

});

// m is [4, 8, 15, 16, 23, 42, 'a', 'aa', 'bb']

If case is not significant, your comparison function should convert the operands to lower-case before comparing them.

**array.splice(start, deleteCount, item...)**

The splice method removes elements from an array, replacing them with new items. The start parameter is the number of a position within the array. The deleteCount parameter is the number of elements to delete starting from that position. If there are additional parameters, those items will be inserted at the position. It returns an array containing the deleted elements.

var a = ['a', 'b', 'c'];

var r = a.splice(1, 1, 'ache', 'bug');

// a is ['a', 'ache', 'bug', 'c']

// r is ['b']

**array.unshift(item...)**

The unshift method is like the push method except that it shoves the items onto the front of this array instead of at the end. It returns the array’s new length:

var a = ['a', 'b', 'c'];

var r = a.unshift('?', '@');

// a is ['?', '@', 'a', 'b', 'c']

// r is 5

**function.apply(thisArg, argArray)**

The apply method invokes a function, passing in the object that will be bound to *this* and an optional array of arguments.

var arr = ['1px','solid', 'white'];

var Obj = function(border, type, color){

this.border = border;

this.type = type;

this.color = color;

}

Obj.prototype.kind = function(a, b, c){

console.log(

this.border + " " + a,

this.type + " " + b,

this.color + " " + c

)

}

var a = {

border: '2',

type: '4',

color: '6',

}

var foo = new Obj(1,3,4);

Obj.prototype.kind.apply(a, arr) // "2 1px" "4 solid" "6 white"

foo.kind.apply(a, arr) // "2 1px" "4 solid" "6 white"

**function.call(thisArg, arg1, arg2…)**

The call method invokes a function, passing in the object that will be bound to *this* and optional comma-separated arguments.

**function.bind(thisArg, arg1, arg2…)**

The call method copies a function, passing in the object that will be bound to *this* and optional comma-separated arguments.

**number.toExponential(fractionDigits)**

The toExponential method converts this number to a string in the exponential form. The optional fractionDigits parameter controls the number of decimal places. It should be between 0 and 20:

document.writeln(Math.PI.toExponential(0)); //3e+0 document.writeln(Math.PI.toExponential(2)); //3.14e+0 document.writeln(Math.PI.toExponential(7)); //3.1415927e+0

document.writeln(Math.PI.toExponential(16)); //3.1415926535897930e+0

**number.toFixed(fractionDigits)**

The toFixed method converts this number to a string in the decimal form. The optional fractionDigits parameter controls the number of decimal places. It should be between 0 and 20. The default is 0:

document.writeln(Math.PI.toFixed(0)); // 3

document.writeln(Math.PI.toFixed(2)); // 3.14

document.writeln(Math.PI.toFixed(7)); // 3.1415927

**number.toPrecision(precision)**

The toPrecision method converts this number to a string in the decimal form. The optional precision parameter controls the number of digits of precision. It should be between 1 and 21:

document.writeln(Math.PI.toPrecision(2)); // 3.1 document.writeln(Math.PI.toPrecision(7)); // 3.141593 document.writeln(Math.PI.toPrecision(16)); // 3.141592653589793 document.writeln(Math.PI.toPrecision( )); // 3.141592653589793

**number.toString(radix)**

The toString method converts this number to a string. The optional radix parameter controls radix, or base. It should be between 2 and 36. The default radix is base 10.

The most common case, number.toString( ), can be written more simply as String(number):

document.writeln(Math.PI.toString( )); // 3.141592653589793

**Object object.hasOwnProperty(name)**

The hasOwnProperty method returns true if the object contains a property having the name. The prototype chain is not examined. This method is useless if the name is hasOwnProperty:

var a = {member: true};

var b = Object.create(a); // from Chapter 3

var t = a.hasOwnProperty('member'); // t is true

var u = b.hasOwnProperty('member'); // u is false

var v = b.member; // v is true // (member is inside the proto property, so member is not a property of b itself);

**RegExp regexp.exec(string)**

The exec method is the most powerful (and slowest) of the methods that use regular expressions. If it successfully matches the regexp and the string, it returns an array.

**String string.charAt(pos)**

The charAt method returns the character at position *pos* in this string. If *pos* is less than zero or greater than or equal to string.length, it returns the empty string. The result of this method is a string:

var name = 'Curly';

var initial = name.charAt(0); // initial is 'C'

**string.charCodeAt(pos)**

The charCodeAt method is the same as charAt except that instead of returning a string, it returns an integer representation of the code point value of the character at position pos in that string. If pos is less than zero or greater than or equal to string.length, it returns NaN:

var name = 'Curly';

var initial = name.charCodeAt(0); // initial is 67

**String.fromCharCode(char...)**

The String.fromCharCode function produces a string from a series of numbers.

var a = String.fromCharCode(67, 97, 116);

// a is 'Cat'

**string.concat(string...)**

The concat method makes a new string by concatenating other strings together. It is rarely used because the + operator is more convenient:

var s = 'C'.concat('a', 't'); // s is 'Cat'

**string.indexOf(searchString, position)**

The indexOf method searches for a *searchString* within a string. If it is found, it returns the position of the first matched character; otherwise, it returns –1. The optional position parameter causes the search to begin at some specified position in the string:

var text = 'Mississippi'; //

var p = text.indexOf('ss'); // p is 2

p = text.indexOf('ss', 3); // p is 5

p = text.indexOf('ss', 6); // p is -1

**string.lastIndexOf(searchString, position)**

The lastIndexOf method is like the indexOf method, except that it searches from the end of the string instead of the front:

var text = 'Mississippi';

var p = text.lastIndexOf('ss'); // p is 5

p = text.lastIndexOf('ss', 3); // p is 2

p = text.lastIndexOf('ss', 6); // p is 5

**string.match(regexp)**

The match method matches a string and a regular expression. How it does this depends on the g flag. If there is no g flag, then the result of calling string.match(regexp) is the same as calling regexp.exec(string). However, if the regexp has the g flag, then it produces an array of all the matches but excludes the capturing groups.

**string.replace(searchValue, replaceValue)**

The replace method does a search and replace operation on this string, producing a new string. The searchValue argument can be a string or a regular expression object. If it is a string, only the first occurrence of the searchValue is replaced, so:

var result = "mother\_in\_law".replace('\_', '-');

will produce "mother-in\_law", which might be a disappointment.

If searchValue is a regular expression and if it has the g flag, then it will replace all occurrences. If it does not have the g flag, then it will replace only the first occurrence. The replaceValue can be a string or a function.

**string.search(regexp)**

The search method is like the indexOf method, except that it takes a regular expression object instead of a string. It returns the position of the first character of the first match, if there is one, or –1 if the search fails. The g flag is ignored. There is no position parameter:

var text = 'and in it he says "Any damn fool could';

var pos = text.search(/["']/); // pos is 18

**string.slice(start, end)**

The slice method makes a new string by copying a portion of another string. If the start parameter is negative, it adds string.length to it. The end parameter is optional, and its default value is string.length. If the end parameter is negative, then string.length is added to it. The end parameter is one greater than the position of the last character. To get n characters starting at position p, use string.slice(p,p+n).

var text = 'and in it he says "Any damn fool could';

var a = text.slice(18); // a is '"Any damn fool could'

var b = text.slice(0, 3); // b is 'and'

**var c = text.slice(-5) // c is 'could'**

var d = text.slice(19, 32); // d is 'Any damn fool'

**string.split(separator, limit)**

The split method creates an array of strings by splitting this string into pieces. The optional limit parameter can limit the number of pieces that will be split. The separator parameter can be a string or a regular expression.

If the separator is the empty string, an array of single characters is produced:

var digits = '0123456789';

var a = digits.split('', 5); // a is ['0', '1', '2', '3', '456789']

var ip = '192.168.1.0';

var b = ip.split('.'); // b is ['192', '168', '1', '0']

**string.toLowerCase( )**

The toLowerCase method produces a new string that is made by converting this string to lowercase.

**string.toUpperCase( )**

The toUpperCase method produces a new string that is made by converting this string to uppercase.