**ARRAY & STRING**

**ARRAY**

**Create an Array**

let fruits = ['Apple', 'Banana']

console.log(fruits.length)

// 2

**Access an Array item using the index position**

let first = fruits[0]

// Apple

let last = fruits[fruits.length - 1]

// Banana

**Loop over an Array**

fruits.forEach(function(item, index, array) {

console.log(item, index)

})

// Apple 0

// Banana 1

**Add an item to the end of an Array**

let newLength = fruits.push('Orange')

// ["Apple", "Banana", "Orange"]

**Remove an item from the end of an Array**

let last = fruits.pop() // remove Orange (from the end)

// ["Apple", "Banana"]

**Remove an item from the beginning of an Array**

let first = fruits.shift() // remove Apple from the front

// ["Banana"]

**Add an item to the beginning of an Array**

let newLength = fruits.unshift('Strawberry') // add to the front

// ["Strawberry", "Banana"]

**Find the index of an item in the Array**

fruits.push('Mango')

// ["Strawberry", "Banana", "Mango"]

let pos = fruits.indexOf('Banana')

// 1

**Remove an item by index position**

let removedItem = fruits.splice(pos, 1) // this is how to remove an item

// ["Strawberry", "Mango"]

**Remove items from an index position**

let vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot']

console.log(vegetables)

// ["Cabbage", "Turnip", "Radish", "Carrot"]

let pos = 1

let n = 2

let removedItems = vegetables.splice(pos, n)

// this is how to remove items, *n* defines the number of items to be removed,

// starting at the index position specified by *pos* and progressing toward the end of array.

console.log(vegetables)

// ["Cabbage", "Carrot"] (the original array is changed)

console.log(removedItems)

// ["Turnip", "Radish"]

**Copy an Array**

let shallowCopy = fruits.slice() // this is how to make a copy

// ["Strawberry", "Mango"]

Accessing array elements

JavaScript arrays are zero-indexed: the first element of an array is at index 0, and the last element is at the index equal to the value of the array's [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length) property minus 1.

Using an invalid index number returns undefined.

let arr = ['this is the first element', 'this is the second element', 'this is the last element']

console.log(arr[0]) // logs 'this is the first element'

console.log(arr[1]) // logs 'this is the second element'

console.log(arr[arr.length - 1]) // logs 'this is the last element'

Array elements are object properties in the same way that toString is a property (to be specific, however, toString() is a method). Nevertheless, trying to access an element of an array as follows throws a syntax error because the property name is not valid:

console.log(arr.0) // a syntax error

There is nothing special about JavaScript arrays and the properties that cause this. JavaScript properties that begin with a digit cannot be referenced with dot notation and must be accessed using bracket notation.

For example, if you had an object with a property named 3d, it can only be referenced using bracket notation.

let years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]

console.log(years.0) // a syntax error

console.log(years[0]) // works properly

renderer.3d.setTexture(model, 'character.png') // a syntax error

renderer['3d'].setTexture(model, 'character.png') // works properly

Note that in the 3d example, '3d' had to be quoted. It's possible to quote the JavaScript array indexes as well (e.g., years['2'] instead of years[2]), although it's not necessary.

The 2 in years[2] is coerced into a string by the JavaScript engine through an implicit toString conversion. As a result, '2' and '02' would refer to two different slots on the years object, and the following example could be true:

console.log(years['2'] != years['02'])

Relationship between length and numerical properties

A JavaScript array's [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length) property and numerical properties are connected.

Several of the built-in array methods (e.g., [join()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/join), [slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice), [indexOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf), etc.) take into account the value of an array's [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length) property when they're called.

Other methods (e.g., [push()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push), [splice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice), etc.) also result in updates to an array's [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length) property.

const fruits = []

fruits.push('banana', 'apple', 'peach')

console.log(fruits.length) // 3

When setting a property on a JavaScript array when the property is a valid array index and that index is outside the current bounds of the array, the engine will update the array's [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length) property accordingly:

fruits[5] = 'mango'

console.log(fruits[5]) // 'mango'

console.log(Object.keys(fruits)) // ['0', '1', '2', '5']

console.log(fruits.length) // 6

Increasing the [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length).

fruits.length = 10

console.log(fruits) // ['banana', 'apple', 'peach', undefined, 'mango', <5 empty items>]

console.log(Object.keys(fruits)) // ['0', '1', '2', '5']

console.log(fruits.length) // 10

console.log(fruits[8]) // undefined

Decreasing the [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length) property does, however, delete elements.

fruits.length = 2

console.log(Object.keys(fruits)) // ['0', '1']

console.log(fruits.length) // 2

This is explained further on the [Array.length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length) page.

**STRINGS**

## Description

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/length), to build and concatenate them using the [+ and += string operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/String_Operators), checking for the existence or location of substrings with the [indexOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf) method, or extracting substrings with the [substring()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/substring) method.

### Creating strings

Strings can be created as primitives, from string literals, or as objects, using the [String()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/String) constructor:

const string1 = "A string primitive";

const string2 = 'Also a string primitive';

const string3 = `Yet another string primitive`;

const string4 = new String("A String object");

String primitives and string objects can be used interchangeably in most situations. See "[String primitives and String objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#String_primitives_and_String_objects)" below.

String literals can be specified using single or double quotes, which are treated identically, or using the backtick character **`**. This last form specifies a [template literal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals): with this form you can interpolate expressions.

### Character access

There are two ways to access an individual character in a string. The first is the [charAt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/charAt) method:

return 'cat'.charAt(1) // returns "a"

The other way (introduced in ECMAScript 5) is to treat the string as an array-like object, where individual characters correspond to a numerical index:

return 'cat'[1] // returns "a"

When using bracket notation for character access, attempting to delete or assign a value to these properties will not succeed. The properties involved are neither writable nor configurable. (See [Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) for more information.)

### Comparing strings

In C, the strcmp() function is used for comparing strings. In JavaScript, you just use the [less-than and greater-than operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators):

let a = 'a'

let b = 'b'

if (a < b) { // true

console.log(a + ' is less than ' + b)

} else if (a > b) {

console.log(a + ' is greater than ' + b)

} else {

console.log(a + ' and ' + b + ' are equal.')

}

A similar result can be achieved using the [localeCompare()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare) method inherited by String instances.

Note that a == b compares the strings in a and b for being equal in the usual case-sensitive way. If you wish to compare without regard to upper or lower case characters, use a function similar to this:

function isEqual(str1, str2)

{

return str1.toUpperCase() === str2.toUpperCase()

} // isEqual

Upper case is used instead of lower case in this function, due to problems with certain UTF-8 character conversions.

### String primitives and String objects

Note that JavaScript distinguishes between String objects and primitive string values. (The same is true of [Boolean](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean) and [Numbers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number).)

String literals (denoted by double or single quotes) and strings returned from String calls in a non-constructor context (that is, called without using the [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) keyword) are primitive strings. JavaScript automatically converts primitives to String objects, so that it's possible to use String object methods for primitive strings. In contexts where a method is to be invoked on a primitive string or a property lookup occurs, JavaScript will automatically wrap the string primitive and call the method or perform the property lookup.

let s\_prim = 'foo'

let s\_obj = new String(s\_prim)

console.log(typeof s\_prim) // Logs "string"

console.log(typeof s\_obj) // Logs "object"

String primitives and String objects also give different results when using [eval()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval). Primitives passed to eval are treated as source code; String objects are treated as all other objects are, by returning the object. For example:

let s1 = '2 + 2' // creates a string primitive

let s2 = new String('2 + 2') // creates a String object

console.log(eval(s1)) // returns the number 4

console.log(eval(s2)) // returns the string "2 + 2"

For these reasons, the code may break when it encounters String objects when it expects a primitive string instead, although generally, authors need not worry about the distinction.

A String object can always be converted to its primitive counterpart with the [valueOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/valueOf) method.

console.log(eval(s2.valueOf())) // returns the number 4