Chapter 7: Strings

Section 7.1: Basic Info and String Concatenation

Strings in JavaScript can be enclosed in Single quotes `'hello'`, Double quotes `"Hello"` and (from ES2015, ES6) in Template Literals (*backticks*) `` `hello` ``.

```
var hello  = "Hello";
var world  = 'world';
var helloW = `Hello World`;              // ES2015 / ES6
```

Strings can be created from other types using the `String()` function.

```
var intString = String(32); // "32"
var booleanString = String(true); // "true"
var nullString = String(null); // "null"
```

Or, `toString()` can be used to convert Numbers, Booleans or Objects to Strings.

```
var intString = (5232).toString(); // "5232"
var booleanString = (false).toString(); // "false"
var objString = ({}).toString(); // "[object Object]"
```

Strings also can be created by using `String.fromCharCode` method.

```
String.fromCharCode(104,101,108,108,111) //"hello"
```

Creating a String object using **`new`** keyword is allowed, but is not recommended as it behaves like Objects unlike primitive strings.

```
var objectString = new String("Yes, I am a String object");
typeof objectString;//"object"
typeof objectString.valueOf();//"string"
```

Concatenating Strings

String concatenation can be done with the `+` concatenation operator, or with the built-in `concat()` method on the String object prototype.

```
var foo = "Foo";
var bar = "Bar";
console.log(foo + bar);        // => "FooBar"
console.log(foo + " " + bar);  // => "Foo Bar"

foo.concat(bar)            // => "FooBar"
"a".concat("b", " ", "d")  // => "ab d"
```

Strings can be concatenated with non-string variables but will type-convert the non-string variables into strings.

```
var string = "string";
var number = 32;
var boolean = true;

console.log(string + number + boolean); // "string32true"
```

String Templates

Version ≥ 6

Strings can be created using template literals (*backticks*) `` `hello` ``.

```
var greeting = `Hello`;
```

With template literals, you can do string interpolation using `${variable}` inside template literals:

```
var place = `World`;
var greet = `Hello ${place}!`

console.log(greet); // "Hello World!"
```

You can use String.raw to get backslashes to be in the string without modification.

```
`a\\b` // =  a\b
String.raw`a\\b` // = a\\b
```

Section 7.2: Reverse String

The most "popular" way of reversing a string in JavaScript is the following code fragment, which is quite common:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}

reverseString('string');    // "gnirts"
```

However, this will work only so long as the string being reversed does not contain surrogate pairs. Astral symbols, i.e. characters outside of the basic multilingual plane, may be represented by two code units, and will lead this naive technique to produce wrong results. Moreover, characters with combining marks (e.g. diaeresis) will appear on the logical "next" character instead of the original one it was combined with.

```
'?????.'.split('').reverse().join(''); //fails
```

While the method will work fine for most languages, a truly accurate, encoding respecting algorithm for string reversal is slightly more involved. One such implementation is a tiny library called Esrever, which uses regular expressions for matching combining marks and surrogate pairs in order to perform the reversing perfectly.

Explanation

Section	Explanation	Result
`str`	The input string	`"string"`
`String.prototype.split( deliminator )`	Splits string `str` into an array. The parameter `""` means to split between each character.	`["s","t","r","i","n","g"]`
`Array.prototype.reverse()`	Returns the array from the split string with its elements in reverse order.	`["g","n","i","r","t","s"]`
`Array.prototype.join( deliminator )`	Joins the elements in the array together into a string. The `""` parameter means an empty deliminator (i.e., the elements of the array are put right next to each other).	`"gnirts"`

Using spread operator

Version ≥ 6

```
function reverseString(str) {
    return [...String(str)].reverse().join('');
}

console.log(reverseString('stackoverflow'));  // "wolfrevokcats"
console.log(reverseString(1337));             // "7331"
console.log(reverseString([1, 2, 3]));        // "3,2,1"
```

Custom `reverse()` function

```
function reverse(string) {
    var strRev = "";
    for (var i = string.length - 1; i >= 0; i--) {
        strRev += string[i];
    }
    return strRev;
}

reverse("zebra");  // "arbez"
```

Section 7.3: Comparing Strings Lexicographically

To compare strings alphabetically, use `localeCompare()`. This returns a negative value if the reference string is lexicographically (alphabetically) before the compared string (the parameter), a positive value if it comes afterwards, and a value of `0` if they are equal.

```
var a = "hello";
var b = "world";

console.log(a.localeCompare(b)); // -1
```

The `>` and `<` operators can also be used to compare strings lexicographically, but they cannot return a value of zero (this can be tested with the `==` equality operator). As a result, a form of the `localeCompare()` function can be written like so:

```
function strcmp(a, b) {
    if(a === b) {
        return 0;
    }

    if (a > b) {
        return 1;
    }

    return -1;
}

console.log(strcmp("hello", "world")); // -1
console.log(strcmp("hello", "hello")); //  0
console.log(strcmp("world", "hello")); //  1
```

This is especially useful when using a sorting function that compares based on the sign of the return value (such as `sort`).

```
var arr = ["bananas", "cranberries", "apples"];
arr.sort(function(a, b) {
    return a.localeCompare(b);
```

```
});
console.log(arr); // [ "apples", "bananas", "cranberries" ]
```

Section 7.4: Access character at index in string

Use `charAt()` to get a character at the specified index in the string.

```
var string = "Hello, World!";
console.log( string.charAt(4) ); // "o"
```

Alternatively, because strings can be treated like arrays, use the index via bracket notation.

```
var string = "Hello, World!";
console.log( string[4] ); // "o"
```

To get the character code of the character at a specified index, use `charCodeAt()`.

```
var string = "Hello, World!";
console.log( string.charCodeAt(4) ); // 111
```

Note that these methods are all getter methods (return a value). Strings in JavaScript are immutable. In other words, none of them can be used to set a character at a position in the string.

Section 7.5: Escaping quotes

If your string is enclosed (i.e.) in single quotes you need to escape the inner literal quote with *backslash* `\`

```
var text = 'L\'albero means tree in Italian';
console.log( text ); \\ "L'albero means tree in Italian"
```

Same goes for double quotes:

```
var text = "I feel \"high\"";
```

Special attention must be given to escaping quotes if you're storing HTML representations within a String, since HTML strings make large use of quotations i.e. in attributes:

```
var content = "<p class=\"special\">Hello World!</p>";        // valid String
var hello   = '<p class="special">I\'d like to say "Hi"</p>'; // valid String
```

Quotes in HTML strings can also be represented using `&apos;` (or `&#39;`) as a single quote and `&quot;` (or `&#34;`) as double quotes.

```
var hi    = "<p class='special'>I'd like to say &quot;Hi&quot;</p>"; // valid String
var hello = '<p class="special">I&apos;d like to say "Hi"</p>';      // valid String
```

Note: The use of `&apos;` and `&quot;` will not overwrite double quotes that browsers can automatically place on attribute quotes. For example `<p class=special>` being made to `<p class="special">`, using `&quot;` can lead to `<p class=""special"">` where `\"` will be `<p class="special">`.

`Version ≥ 6`

If a string has `'` and `"` you may want to consider using template literals (*also known as template strings in previous ES6 editions*), which do not require you to escape `'` and `"`. These use backticks (`` ` ``) instead of single or double quotes.

```
var x = `"Escaping " and ' can become very annoying`;
```

Section 7.6: Word Counter

Say you have a **<textarea>** and you want to retrieve info about the number of:

- Characters (total)
- Characters (no spaces)
- Words
- Lines

```
function wordCount( val ){
    var wom = val.match(/\S+/g);
    return {
        charactersNoSpaces : val.replace(/\s+/g, '').length,
        characters         : val.length,
        words              : wom ? wom.length : 0,
        lines              : val.split(/\r*\n/).length
    };
}


// Use like:
wordCount( someMultilineText ).words;   // (Number of words)
```

jsFiddle example

Section 7.7: Trim whitespace

To trim whitespace from the edges of a string, use `String.prototype.trim`:

```
"    some whitespaced string  ".trim();  // "some whitespaced string"
```

Many JavaScript engines, but not Internet Explorer, have implemented non-standard `trimLeft` and `trimRight` methods. There is a proposal, currently at Stage 1 of the process, for standardised `trimStart` and `trimEnd` methods, aliased to `trimLeft` and `trimRight` for compatibility.

```
// Stage 1 proposal
"    this is me    ".trimStart();  // "this is me    "
"    this is me    ".trimEnd();  // "    this is me"

// Non-standard methods, but currently implemented by most engines
"    this is me    ".trimLeft();  // "this is me    "
"    this is me    ".trimRight();  // "    this is me"
```

Section 7.8: Splitting a string into an array

Use `.split` to go from strings to an array of the split substrings:

```
var s = "one, two, three, four, five"
s.split(", ");  // ["one", "two", "three", "four", "five"]
```

Use the **array method** `.join` to go back to a string:

```
s.split(", ").join("--");  // "one--two--three--four--five"
```

Section 7.9: Strings are unicode

All JavaScript strings are unicode!

```
var s = "some ∆≈ƒ unicode ¡™£¢¢¢";
s.charCodeAt(5);  // 8710
```

There are no raw byte or binary strings in JavaScript. To effectively handle binary data, use Typed Arrays.

Section 7.10: Detecting a string

To detect whether a parameter is a *primitive* string, use `typeof`:

```
var aString = "my string";
var anInt = 5;
var anObj = {};
typeof aString === "string";   // true
typeof anInt === "string";     // false
typeof anObj === "string";     // false
```

If you ever have a `String` object, via `new String("somestr")`, then the above will not work. In this instance, we can use `instanceof`:

```
var aStringObj = new String("my string");
aStringObj instanceof String;    // true
```

To cover both instances, we can write a simple helper function:

```
var isString = function(value) {
    return typeof value === "string" || value instanceof String;
};

var aString = "Primitive String";
var aStringObj = new String("String Object");
isString(aString); // true
isString(aStringObj); // true
isString({}); // false
isString(5); // false
```

Or we can make use of `toString` function of `Object`. This can be useful if we have to check for other types as well say in a switch statement, as this method supports other datatypes as well just like `typeof`.

```
var pString = "Primitive String";
var oString = new String("Object Form of String");
Object.prototype.toString.call(pString);//"[object String]"
Object.prototype.toString.call(oString);//"[object String]"
```

A more robust solution is to not *detect* a string at all, rather only check for what functionality is required. For example:

```
var aString = "Primitive String";
// Generic check for a substring method
if(aString.substring) {

}
// Explicit check for the String substring prototype method
```

```
if(aString.substring === String.prototype.substring) {
    aString.substring(0, );
}
```

Section 7.11: Substrings with slice

Use `.slice()` to extract substrings given two indices:

```
var s = "0123456789abcdefg";
s.slice(0, 5);  // "01234"
s.slice(5, 6);  // "5"
```

Given one index, it will take from that index to the end of the string:

```
s.slice(10);    // "abcdefg"
```

Section 7.12: Character code

The method `charCodeAt` retrieves the Unicode character code of a single character:

```
var charCode = "µ".charCodeAt(); // The character code of the letter µ is 181
```

To get the character code of a character in a string, the 0-based position of the character is passed as a parameter to `charCodeAt`:

```
var charCode = "ABCDE".charCodeAt(3); // The character code of "D" is 68
```

Version ≥ 6

Some Unicode symbols don't fit in a single character, and instead require two UTF-16 surrogate pairs to encode. This is the case of character codes beyond 216 - 1 or 63553. These extended character codes or *code point* values can be retrieved with `codePointAt`:

```
// The Grinning Face Emoji has code point 128512 or 0x1F600
var codePoint = "????".codePointAt();
```

Section 7.13: String Representations of Numbers

JavaScript has native conversion from *Number* to its *String representation* for any base from *2 to 36*.

The most common representation after *decimal (base 10)* is *hexadecimal (base 16)*, but the contents of this section work for all bases in the range.

In order to convert a *Number* from decimal (base 10) to its hexadecimal (base 16) *String representation* the *toString* method can be used with *radix 16*.

```
// base 10 Number
var b10 = 12;

// base 16 String representation
var b16 = b10.toString(16); // "c"
```

If the number represented is an integer, the inverse operation for this can be done with `parseInt` and the *radix 16* again

```
// base 16 String representation
var b16 = 'c';

// base 10 Number
var b10 = parseInt(b16, 16); // 12
```

To convert an arbitrary number (i.e. non-integer) from its *String representation* into a *Number*, the operation must be split into two parts; the integer part and the fraction part.

Version ≥ 6

```
let b16 = '3.243f3e0370cdc';
// Split into integer and fraction parts
let [i16, f16] = b16.split('.');

// Calculate base 10 integer part
let i10 = parseInt(i16, 16); // 3

// Calculate the base 10 fraction part
let f10 = parseInt(f16, 16) / Math.pow(16, f16.length); // 0.14158999999999988

// Put the base 10 parts together to find the Number
let b10 = i10 + f10; // 3.14159
```

Note 1: Be careful as small errors may be in the result due to differences in what is possible to be represented in different bases. It may be desirable to perform some kind of rounding afterwards.
Note 2: Very long representations of numbers may also result in errors due to the accuracy and maximum values of *Numbers* of the environment the conversions are happening in.

Section 7.14: String Find and Replace Functions

To search for a string inside a string, there are several functions:

`indexOf( searchString )` and `lastIndexOf( searchString )`

`indexOf()` will return the index of the first occurrence of `searchString` in the string. If `searchString` is not found, then `-1` is returned.

```
var string = "Hello, World!";
console.log( string.indexOf("o") ); // 4
console.log( string.indexOf("foo") ); // -1
```

Similarly, `lastIndexOf()` will return the index of the last occurrence of `searchstring` or `-1` if not found.

```
var string = "Hello, World!";
console.log( string.lastIndexOf("o") );   // 8
console.log( string.lastIndexOf("foo") ); // -1
```

`includes( searchString, start )`

`includes()` will return a boolean that tells whether `searchString` exists in the string, starting from index `start` (defaults to 0). This is better than `indexOf()` if you simply need to test for existence of a substring.

```
var string = "Hello, World!";
console.log( string.includes("Hello") ); // true
console.log( string.includes("foo") );   // false
```

`replace( regexp|substring, replacement|replaceFunction )`

`replace()` will return a string that has all occurrences of substrings matching the RegExp `regexp` or string `substring` with a string `replacement` or the returned value of `replaceFunction`.

Note that this does not modify the string in place, but returns the string with replacements.

```
var string = "Hello, World!";
string = string.replace( "Hello", "Bye" );
console.log( string ); // "Bye, World!"

string = string.replace( /W.{3}d/g, "Universe" );
console.log( string ); // "Bye, Universe!"
```

`replaceFunction` can be used for conditional replacements for regular expression objects (i.e., with use with `regexp`). The parameters are in the following order:

Parameter	Meaning
`match`	the substring that matches the entire regular expressiong
`g1, g2, g3, ...`	the matching groups in the regular expression
`offset`	the offset of the match in the entire string
`string`	the entire string

Note that all parameters are optional.

```
var string = "heLlo, woRlD!";
string = string.replace( /([a-zA-Z])([a-zA-Z]+)/g, function(match, g1, g2) {
    return g1.toUpperCase() + g2.toLowerCase();
});
console.log( string ); // "Hello, World!"
```

Section 7.15: Find the index of a substring inside a string

The `.indexOf` method returns the index of a substring inside another string (if exists, or -1 if otherwise)

```
'Hellow World'.indexOf('Wor');    // 7
```

`.indexOf` also accepts an additional numeric argument that indicates on what index should the function start looking

```
"harr dee harr dee harr".indexOf("dee", 10); // 14
```

You should note that `.indexOf` is case sensitive

```
 'Hellow World'.indexOf('WOR');    // -1
```

Section 7.16: String to Upper Case

> String.prototype.toUpperCase():

```
console.log('qwerty'.toUpperCase()); // 'QWERTY'
```

Section 7.17: String to Lower Case

String.prototype.toLowerCase()

```
console.log('QWERTY'.toLowerCase()); // 'qwerty'
```

Section 7.18: Repeat a String

Version ≥ 6

This can be done using the .repeat() method:

```
"abc".repeat(3);  // Returns "abcabcabc"
"abc".repeat(0);  // Returns ""
"abc".repeat(-1); // Throws a RangeError
```

Version < 6

In the general case, this should be done using a correct polyfill for the ES6 String.prototype.repeat() method. Otherwise, the idiom `new Array(n + 1).join(myString)` can repeat `n` times the string `myString`:

```
var myString = "abc";
var n = 3;

new Array(n + 1).join(myString);  // Returns "abcabcabc"
```