Strings

In JavaScript, the textual data is stored as strings. There is no separate type for a single character.

The internal format for strings is always UTF-16, it is not tied to the page encoding.

Quotes

Let's recall the kinds of quotes.

Strings can be enclosed within either single quotes, double quotes or backticks:

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

Single and double quotes are essentially the same. Backticks, however, allow us to embed any expression into the string, by wrapping it in `${…}`:

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

Another advantage of using backticks is that they allow a string to span multiple lines:

```
let guestList = `Guests:
 * John
 * Pete
 * Mary
`;

alert(guestList); // a list of guests, multiple lines
```

Looks natural, right? But single or double quotes do not work this way.

If we use them and try to use multiple lines, there'll be an error:

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL
  * John";
```

Single and double quotes come from ancient times of language creation when the need for multiline strings was not taken into account. Backticks appeared much later and thus are more versatile.

Backticks also allow us to specify a "template function" before the first backtick. The syntax is: `func`string``. The function `func` is called automatically, receives the string and embedded expressions and can process them. This is called "tagged templates". This feature makes it easier to implement custom templating, but is rarely used in practice. You can read more about it in the manual.

Special characters

It is still possible to create multiline strings with single and double quotes by using a so-called “newline character”, written as `\n`, which denotes a line break:

```
let guestList = "Guests:\n * John\n * Pete\n * Mary";

alert(guestList); // a multiline list of guests
```

For example, these two lines are equal, just written differently:

```
let str1 = "Hello\nWorld"; // two lines using a "newline symbol"

// two lines using a normal newline and backticks
let str2 = `Hello
World`;

alert(str1 == str2); // true
```

There are other, less common “special” characters.

Here’s the full list:

Character	Description
`\n`	New line
`\r`	Carriage return: not used alone. Windows text files use a combination of two characters `\r\n` to represent a line break.
`\'`, `\"`	Quotes
`\\`	Backslash
`\t`	Tab
`\b`, `\f`, `\v`	Backspace, Form Feed, Vertical Tab – kept for compatibility, not used nowadays.
`\xXX`	Unicode character with the given hexadecimal Unicode `XX`, e.g. `'\x7A'` is the same as `'z'`.
`\uXXXX`	A Unicode symbol with the hex code `XXXX` in UTF-16 encoding, for instance `\u00A9` – is a Unicode for the copyright symbol ©. It must be exactly 4 hex digits.
`\u{X…XXXXXX}` (1 to 6 hex characters)	A Unicode symbol with the given UTF-32 encoding. Some rare characters are encoded with two Unicode symbols, taking 4 bytes. This way we can insert long codes.

Examples with Unicode:

```
alert( "\u00A9" ); // ©
alert( "\u{20331}" ); // 佫, a rare Chinese hieroglyph (long Unicode)
alert( "\u{1F60D}" ); //   , a smiling face symbol (another long Unicode)
```

All special characters start with a backslash character `\`. It is also called an “escape character”.

We might also use it if we wanted to insert a quote into the string.

For instance:

```
alert( 'I\'m the Walrus!' ); // I'm the Walrus!
```

As you can see, we have to prepend the inner quote by the backslash `\'`, because otherwise it would indicate the string end.

Of course, only the quotes that are the same as the enclosing ones need to be escaped. So, as a more elegant solution, we could switch to double quotes or backticks instead:

```
alert( `I'm the Walrus!` ); // I'm the Walrus!
```

Note that the backslash `\` serves for the correct reading of the string by JavaScript, then disappears. The in-memory string has no `\`. You can clearly see that in `alert` from the examples above.

But what if we need to show an actual backslash `\` within the string?

That's possible, but we need to double it like `\\`:

```
alert( `The backslash: \\` ); // The backslash: \
```

String length

The `length` property has the string length:

```
alert( `My\n`.length ); // 3
```

Note that `\n` is a single "special" character, so the length is indeed `3`.

`length` is a property

> People with a background in some other languages sometimes mistype by calling `str.length()` instead of just `str.length`. That doesn't work.
>
> Please note that `str.length` is a numeric property, not a function. There is no need to add parenthesis after it.

Accessing characters

To get a character at position `pos`, use square brackets `[pos]` or call the method str.charAt(pos). The first character starts from the zero position:

```
let str = `Hello`;

// the first character
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// the last character
alert( str[str.length - 1] ); // o
```

The square brackets are a modern way of getting a character, while `charAt` exists mostly for historical reasons.

The only difference between them is that if no character is found, `[]` returns `undefined`, and `charAt` returns an empty string:

```
let str = `Hello`;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (an empty string)
```
We can also iterate over characters using `for..of`:

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char becomes "H", then "e", then "l" etc)
}
```

Strings are immutable

Strings can't be changed in JavaScript. It is impossible to change a character.

Let's try it to show that it doesn't work:

```
let str = 'Hi';

str[0] = 'h'; // error
alert( str[0] ); // doesn't work
```
The usual workaround is to create a whole new string and assign it to `str` instead of the old one.

For instance:

```
let str = 'Hi';

str = 'h' + str[1]; // replace the string

alert( str ); // hi
```
In the following sections we'll see more examples of this.

Changing the case

Methods toLowerCase() and toUpperCase() change the case:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```
Or, if we want a single character lowercased:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

Searching for a substring

There are multiple ways to look for a substring within a string.

str.indexOf

The first method is str.indexOf(substr, pos).

It looks for the `substr` in `str`, starting from the given position `pos`, and returns the position where the match was found or `-1` if nothing can be found.

For instance:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the beginning
alert( str.indexOf('widget') ); // -1, not found, the search is case-sensitive

alert( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget with id)
```

The optional second parameter allows us to start searching from a given position.

For instance, the first occurrence of `"id"` is at position `1`. To look for the next occurrence, let's start the search from position `2`:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

If we're interested in all occurrences, we can run `indexOf` in a loop. Every new call is made with the position after the previous match:

```
let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // let's look for it

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Found at ${foundPos}` );
  pos = foundPos + 1; // continue the search from the next position
}
```

The same algorithm can be layed out shorter:

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

`str.lastIndexOf(substr, position)`

> There is also a similar method str.lastIndexOf(substr, position) that searches from the end of a string to its beginning.

It would list the occurrences in the reverse order.

There is a slight inconvenience with `indexOf` in the `if` test. We can't put it in the `if` like this:

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
    alert("We found it"); // doesn't work!
}
```

The `alert` in the example above doesn't show
because `str.indexOf("Widget")` returns `0` (meaning that it found the match at the starting position). Right, but `if` considers `0` to be `false`.

So, we should actually check for `-1`, like this:

```
let str = "Widget with id";

if (str.indexOf("Widget") != -1) {
    alert("We found it"); // works now!
}
```

The bitwise NOT trick

One of the old tricks used here is the bitwise NOT `~` operator. It converts the number to a 32-bit integer (removes the decimal part if exists) and then reverses all bits in its binary representation.

In practice, that means a simple thing: for 32-bit integers `~n` equals `-(n+1)`.

For instance:

```
alert( ~2 ); // -3, the same as -(2+1)
alert( ~1 ); // -2, the same as -(1+1)
alert( ~0 ); // -1, the same as -(0+1)
alert( ~-1 ); // 0, the same as -(-1+1)
```

As we can see, `~n` is zero only if `n == -1` (that's for any 32-bit signed integer `n`).

So, the test `if ( ~str.indexOf("...") )` is truthy only if the result of `indexOf` is not `-1`. In other words, when there is a match.

People use it to shorten `indexOf` checks:

```
let str = "Widget";

if (~str.indexOf("Widget")) {
  alert( 'Found it!' ); // works
}
```

It is usually not recommended to use language features in a non-obvious way, but this particular trick is widely used in old code, so we should understand it.

Just remember: `if (~str.indexOf(...))` reads as "if found".

To be precise though, as big numbers are truncated to 32 bits by `~` operator, there exist other numbers that give `0`, the smallest is `~4294967295=0`. That makes such check correct only if a string is not that long.

Right now we can see this trick only in the old code, as modern JavaScript provides `.includes` method (see below).

includes, startsWith, endsWith

The more modern method str.includes(substr, pos) returns `true/false` depending on whether `str` contains `substr` within.

It's the right choice if we need to test for the match, but don't need its position:

```
alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false
```

The optional second argument of `str.includes` is the position to start searching from:

```
alert( "Widget".includes("id") ); // true
alert( "Widget".includes("id", 3) ); // false, from position 3 there is
no "id"
```

The methods str.startsWith and str.endsWith do exactly what they say:

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" starts with
"Wid"
alert( "Widget".endsWith("get") ); // true, "Widget" ends with "get"
```

Getting a substring

There are 3 methods in JavaScript to get a substring: `substring`, `substr` and `slice`.

`str.slice(start [, end])`

> Returns the part of the string from `start` to (but not including) `end`.
>
> For instance:
>
> ```
> let str = "stringify";
> alert( str.slice(0, 5) ); // 'strin', the substring from 0 to 5
> (not including 5)
> alert( str.slice(0, 1) ); // 's', from 0 to 1, but not including
> 1, so only character at 0
> ```
>
> If there is no second argument, then `slice` goes till the end of the string:
>
> ```
> let str = "stringify";
> alert( str.slice(2) ); // 'ringify', from the 2nd position till
> the end
> ```
>
> Negative values for `start/end` are also possible. They mean the position is counted from the string end:
>
> ```
> let str = "stringify";
> ```

```
// start at the 4th position from the right, end at the 1st from the right
alert( str.slice(-4, -1) ); // 'gif'
```

`str.substring(start [, end])`

Returns the part of the string between `start` and `end`.

This is almost the same as `slice`, but it allows `start` to be greater than `end`.

For instance:

```
let str = "stringify";

// these are same for substring
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// ...but not for slice:
alert( str.slice(2, 6) ); // "ring" (the same)
alert( str.slice(6, 2) ); // "" (an empty string)
```

Negative arguments are (unlike slice) not supported, they are treated as `0`.

`str.substr(start [, length])`

Returns the part of the string from `start`, with the given `length`.

In contrast with the previous methods, this one allows us to specify the `length` instead of the ending position:

```
let str = "stringify";
alert( str.substr(2, 4) ); // 'ring', from the 2nd position get 4 characters
```

The first argument may be negative, to count from the end:

```
let str = "stringify";
alert( str.substr(-4, 2) ); // 'gi', from the 4th position get 2 characters
```

Let's recap these methods to avoid any confusion:

method	selects…	negatives
`slice(start, end)`	from `start` to `end` (not including `end`)	allows negatives
`substring(start, end)`	between `start` and `end`	negative values mean `0`
`substr(start, length)`	from `start` get `length` characters	allows negative `start`

Which one to choose?

All of them can do the job. Formally, `substr` has a minor drawback: it is described not in the core JavaScript specification, but in Annex B, which covers browser-only features that exist mainly for historical reasons. So, non-browser environments may fail to support it. But in practice it works everywhere.

Of the other two variants, `slice` is a little bit more flexible, it allows negative arguments and shorter to write. So, it's enough to remember solely `slice` of these three methods.

Comparing strings

As we know from the chapter Comparisons, strings are compared character-by-character in alphabetical order.

Although, there are some oddities.

1. A lowercase letter is always greater than the uppercase:

   ```
   alert( 'a' > 'Z' ); // true
   ```
2. Letters with diacritical marks are "out of order":

   ```
   alert( 'Österreich' > 'Zealand' ); // true
   ```
   This may lead to strange results if we sort these country names. Usually people would expect `Zealand` to come after `Österreich` in the list.

To understand what happens, let's review the internal representation of strings in JavaScript.

All strings are encoded using UTF-16. That is: each character has a corresponding numeric code. There are special methods that allow to get the character for the code and back.

`str.codePointAt(pos)`

Returns the code for the character at position `pos`:

```
// different case letters have different codes
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

`String.fromCodePoint(code)`

Creates a character by its numeric `code`

```
alert( String.fromCodePoint(90) ); // Z
```

We can also add Unicode characters by their codes using `\u` followed by the hex code:

```
// 90 is 5a in hexadecimal system
alert( '\u005a' ); // Z
```

Now let's see the characters with codes `65..220` (the latin alphabet and a little bit extra) by making a string of them:

```
let str = '';

for (let i = 65; i <= 220; i++) {
  str += String.fromCodePoint(i);
}
```

```
alert( str );
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ OOO
// ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ
```
See? Capital characters go first, then a few special ones, then lowercase characters, and `Ö` near the end of the output.

Now it becomes obvious why `a > Z`.

The characters are compared by their numeric code. The greater code means that the character is greater. The code for `a` (97) is greater than the code for `Z` (90).

- All lowercase letters go after uppercase letters because their codes are greater.
- Some letters like `Ö` stand apart from the main alphabet. Here, its code is greater than anything from `a` to `z`.

Correct comparisons

The “right” algorithm to do string comparisons is more complex than it may seem, because alphabets are different for different languages.

So, the browser needs to know the language to compare.

Luckily, all modern browsers (IE10- requires the additional library Intl.js) support the internationalization standard ECMA-402.

It provides a special method to compare strings in different languages, following their rules.

The call str.localeCompare(str2) returns an integer indicating whether `str` is less, equal or greater than `str2` according to the language rules:

- Returns a negative number if `str` is less than `str2`.
- Returns a positive number if `str` is greater than `str2`.
- Returns `0` if they are equivalent.

For instance:

```
alert( 'Österreich'.localeCompare('Zealand') ); // -1
```
This method actually has two additional arguments specified in the documentation, which allows it to specify the language (by default taken from the environment, letter order depends on the language) and setup additional rules like case sensitivity or should `"a"` and `"á"` be treated as the same etc.

Internals, Unicode

Advanced knowledge

> The section goes deeper into string internals. This knowledge will be useful for you if you plan to deal with emoji, rare mathematical or hieroglyphic characters or other rare symbols.
>
> You can skip the section if you don't plan to support them.

Surrogate pairs

All frequently used characters have 2-byte codes. Letters in most european languages, numbers, and even most hieroglyphs, have a 2-byte representation.

But 2 bytes only allow 65536 combinations and that's not enough for every possible symbol. So rare symbols are encoded with a pair of 2-byte characters called "a surrogate pair".

The length of such symbols is `2`:

```
alert( '𝒳'.length ); // 2, MATHEMATICAL SCRIPT CAPITAL X
alert( '😂'.length ); // 2, FACE WITH TEARS OF JOY
alert( '𩷶'.length ); // 2, a rare Chinese hieroglyph
```

Note that surrogate pairs did not exist at the time when JavaScript was created, and thus are not correctly processed by the language!

We actually have a single symbol in each of the strings above, but the `length` shows a length of `2`.

`String.fromCodePoint` and `str.codePointAt` are few rare methods that deal with surrogate pairs right. They recently appeared in the language. Before them, there were only String.fromCharCode and str.charCodeAt. These methods are actually the same as `fromCodePoint/codePointAt`, but don't work with surrogate pairs.

Getting a symbol can be tricky, because surrogate pairs are treated as two characters:

```
alert( '𝒳'[0] ); // strange symbols...
alert( '𝒳'[1] ); // ...pieces of the surrogate pair
```

Note that pieces of the surrogate pair have no meaning without each other. So the alerts in the example above actually display garbage.

Technically, surrogate pairs are also detectable by their codes: if a character has the code in the interval of `0xd800..0xdbff`, then it is the first part of the surrogate pair. The next character (second part) must have the code in interval `0xdc00..0xdfff`. These intervals are reserved exclusively for surrogate pairs by the standard.

In the case above:

```
// charCodeAt is not surrogate-pair aware, so it gives codes for parts

alert( '𝒳'.charCodeAt(0).toString(16) ); // d835, between 0xd800 and 0xdbff
alert( '𝒳'.charCodeAt(1).toString(16) ); // dcb3, between 0xdc00 and 0xdfff
```

You will find more ways to deal with surrogate pairs later in the chapter Iterables. There are probably special libraries for that too, but nothing famous enough to suggest here.

Diacritical marks and normalization

In many languages there are symbols that are composed of the base character with a mark above/under it.

For instance, the letter `a` can be the base character for: `àáâäãåā`. Most common "composite" character have their own code in the UTF-16 table. But not all of them, because there are too many possible combinations.

To support arbitrary compositions, UTF-16 allows us to use several Unicode characters: the base character followed by one or many "mark" characters that "decorate" it.

For instance, if we have `S` followed by the special "dot above" character (code `\u0307`), it is shown as Ṡ.

```
alert( 'S\u0307' ); // Ṡ
```

If we need an additional mark above the letter (or below it) – no problem, just add the necessary mark character.

For instance, if we append a character "dot below" (code `\u0323`), then we'll have "S with dots above and below": Ṩ.

For example:

```
alert( 'S\u0307\u0323' ); // Ṩ
```

This provides great flexibility, but also an interesting problem: two characters may visually look the same, but be represented with different Unicode compositions.

For instance:

```
let s1 = 'S\u0307\u0323'; // Ṩ, S + dot above + dot below
let s2 = 'S\u0323\u0307'; // Ṩ, S + dot below + dot above

alert( `s1: ${s1}, s2: ${s2}` );

alert( s1 == s2 ); // false though the characters look identical (?!)
```

To solve this, there exists a "Unicode normalization" algorithm that brings each string to the single "normal" form.

It is implemented by str.normalize().

```
alert( "S\u0307\u0323".normalize() == "S\u0323\u0307".normalize() ); // true
```

It's funny that in our situation `normalize()` actually brings together a sequence of 3 characters to one: `\u1e68` (S with two dots).

```
alert( "S\u0307\u0323".normalize().length ); // 1

alert( "S\u0307\u0323".normalize() == "\u1e68" ); // true
```

In reality, this is not always the case. The reason being that the symbol Ṩ is “common enough”, so UTF-16 creators included it in the main table and gave it the code.

If you want to learn more about normalization rules and variants – they are described in the appendix of the Unicode standard: Unicode Normalization Forms, but for most practical purposes the information from this section is enough.

Summary

- There are 3 types of quotes. Backticks allow a string to span multiple lines and embed expressions `${…}`.
- Strings in JavaScript are encoded using UTF-16.
- We can use special characters like `\n` and insert letters by their Unicode using `\u...`.
- To get a character, use: `[]`.
- To get a substring, use: `slice` or `substring`.
- To lowercase/uppercase a string, use: `toLowerCase/toUpperCase`.
- To look for a substring, use: `indexOf`, or `includes/startsWith/endsWith` for simple checks.
- To compare strings according to the language, use: `localeCompare`, otherwise they are compared by character codes.

There are several other helpful methods in strings:

- `str.trim()` – removes (“trims”) spaces from the beginning and end of the string.
- `str.repeat(n)` – repeats the string `n` times.
- …and more to be found in the manual.

Strings also have methods for doing search/replace with regular expressions. But that’s big topic, so it’s explained in a separate tutorial section Regular expressions.