Intro:

JavaScript can be written right in a web page's HTML and run automatically as the page loads.
Scripts are provided and executed as plain text. They don't need special preparation or compilation to run.

In this aspect, JavaScript is very different from another language called Java.

Created by Netscape, follows ECMAScript i.e. the std on which JS is parsed.

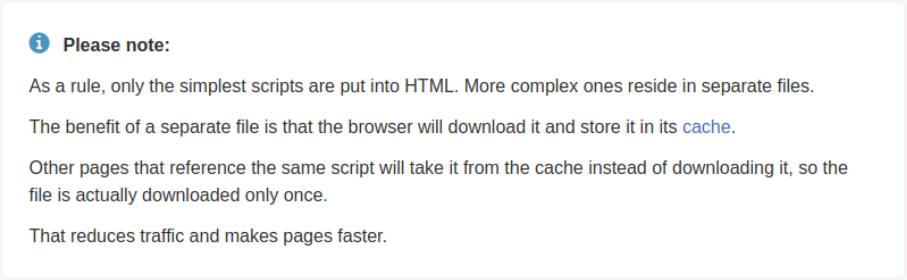
How browser renders html:

how browser renders

Note that the elements with the display: none are not the part of render tree.

In layout phase, a box model is generated that knows the exact positions and size of the elements.

JS is the default browser lang



Please note:

As a rule, only the simplest scripts are put into HTML. More complex ones reside in separate files.

The benefit of a separate file is that the browser will download it and store it in its cache.

Other pages that reference the same script will take it from the cache instead of downloading it, so the file is actually downloaded only once.

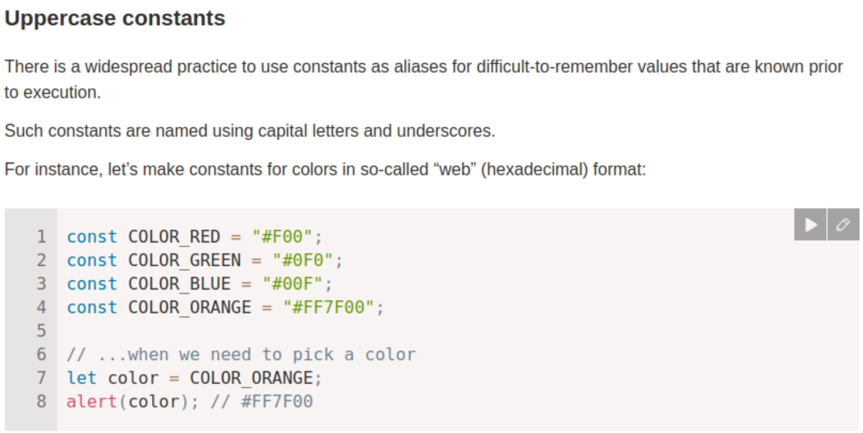
That reduces traffic and makes pages faster.

"use strict" : to apply the modern script i.e. ES5.

JS classes and modules uses use strict by default.

Variables:

Can assign vars w/o let keyword when NOT using 'use strict'.



Uppercase constants

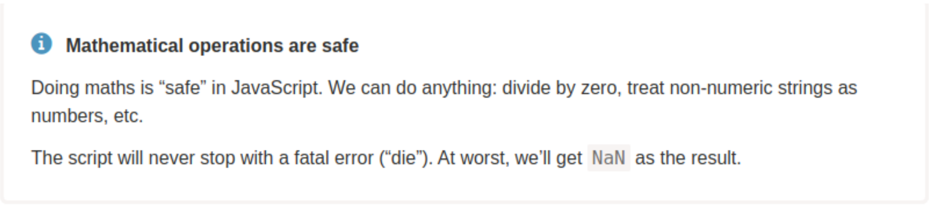
There is a widespread practice to use constants as aliases for difficult-to-remember values that are known prior to execution.

Such constants are named using capital letters and underscores.

For instance, let's make constants for colors in so-called "web" (hexadecimal) format:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

- So, if there's a `NaN` somewhere in a mathematical expression, it propagates to the whole result (there's only one exception to that: `NaN ** 0` is `1`).

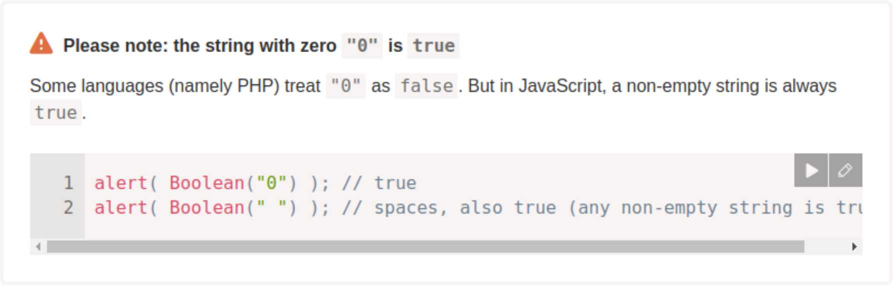


Mathematical operations are safe

Doing maths is "safe" in JavaScript. We can do anything: divide by zero, treat non-numeric strings as numbers, etc.

The script will never stop with a fatal error ("die"). At worst, we'll get `NaN` as the result.

Type Conversion:



Please note: the string with zero `"0"` is `true`

Some languages (namely PHP) treat `"0"` as `false`. But in JavaScript, a non-empty string is always `true`.

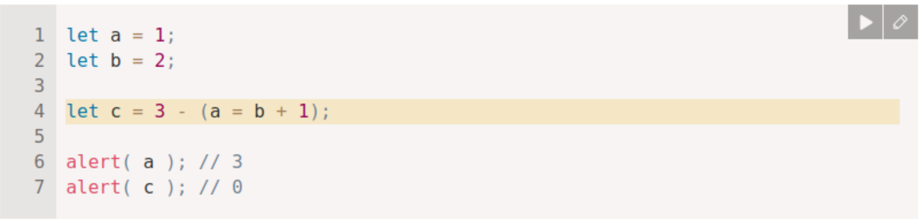
```
alert( Boolean("0") ); // true
alert( Boolean(" ") ); // spaces, also true (any non-empty string is tru
```

Whenever JS comes across a mathematical operator, it tries to convert the operands to number, if not possible, NaN is shown.

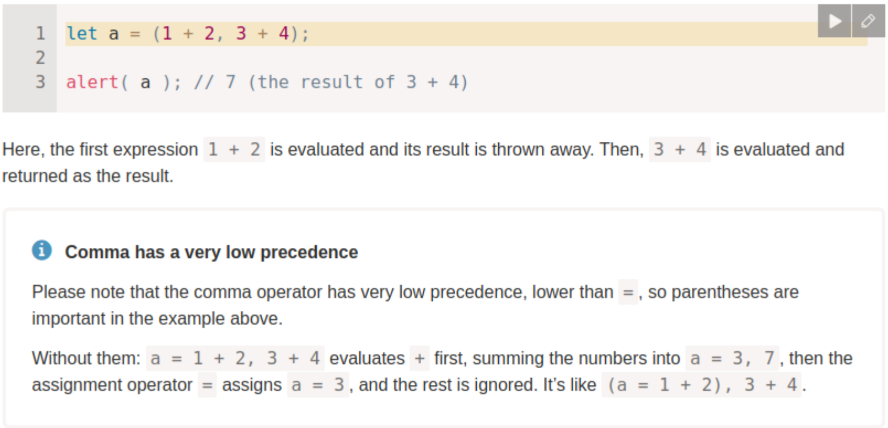
Ex: "5" + "2" gives 7

Can also convert str to nums using + i.e. Number(str) == +(str)

Assignment operator stores the value in LHS as well as returns it's value



```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```



```
let a = (1 + 2, 3 + 4);

alert( a ); // 7 (the result of 3 + 4)
```

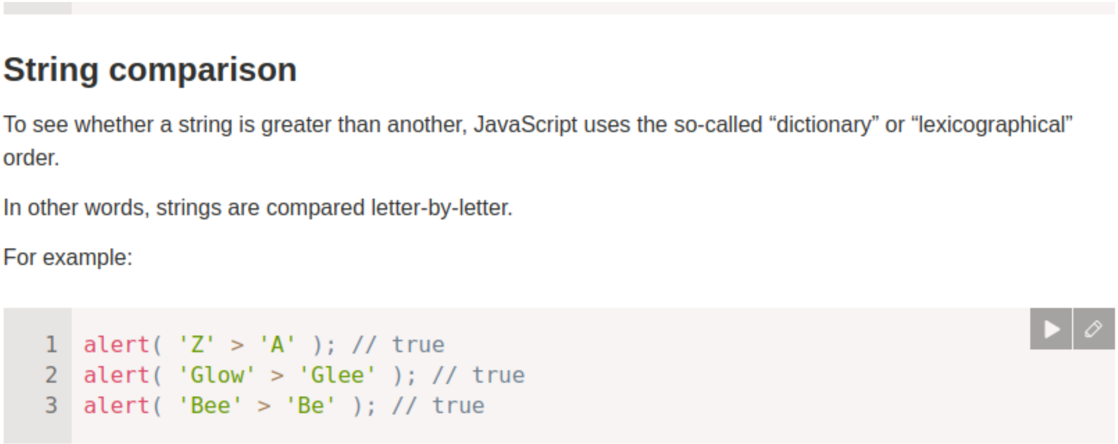
Here, the first expression `1 + 2` is evaluated and its result is thrown away. Then, `3 + 4` is evaluated and returned as the result.

Comma has a very low precedence

Please note that the comma operator has very low precedence, lower than `=`, so parentheses are important in the example above.

Without them: `a = 1 + 2, 3 + 4` evaluates `+` first, summing the numbers into `a = 3, 7`, then the assignment operator `=` assigns `a = 3`, and the rest is ignored. It's like `(a = 1 + 2), 3 + 4`.

String Comparison:



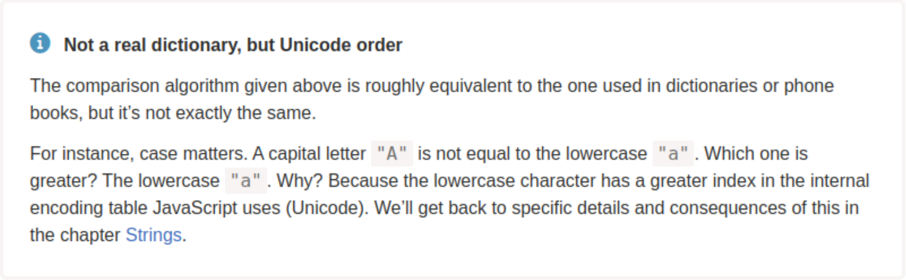
String comparison

To see whether a string is greater than another, JavaScript uses the so-called "dictionary" or "lexicographical" order.

In other words, strings are compared letter-by-letter.

For example:

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```



Not a real dictionary, but Unicode order

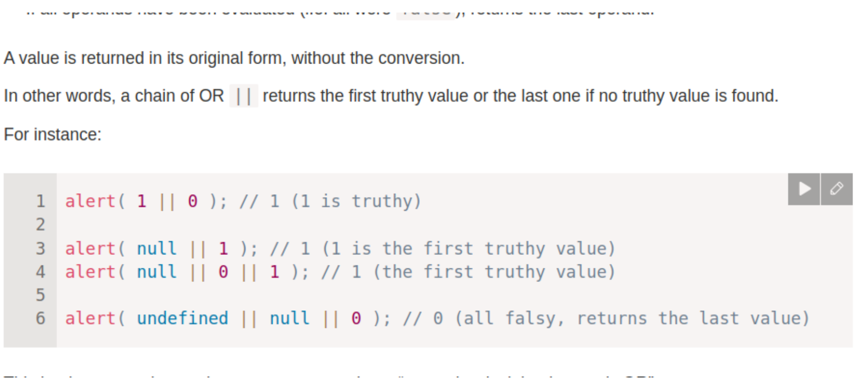
The comparison algorithm given above is roughly equivalent to the one used in dictionaries or phone books, but it's not exactly the same.

For instance, case matters. A capital letter `"A"` is not equal to the lowercase `"a"`. Which one is greater? The lowercase `"a"`. Why? Because the lowercase character has a greater index in the internal encoding table JavaScript uses (Unicode). We'll get back to specific details and consequences of this in the chapter Strings.

Conditional Statements:

The purpose of ? Operator is to return one value or another depending on the condition. Use it for that purpose only and NOT as a substitution of if statement.

Logical Operator:



- If all operands have been evaluated (i.e. all were `false`), returns the last operand.

A value is returned in its original form, without the conversion.

In other words, a chain of OR `||` returns the first truthy value or the last one if no truthy value is found.

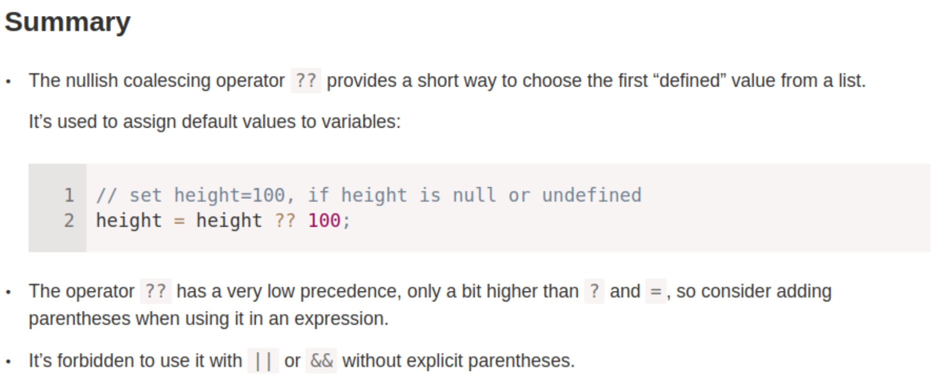
For instance:

```
alert( 1 || 0 ); // 1 (1 is truthy)

alert( null || 1 ); // 1 (1 is the first truthy value)
alert( null || 0 || 1 ); // 1 (the first truthy value)

alert( undefined || null || 0 ); // 0 (all falsy, returns the last value)
```

Or can also be used as if statement.
Or = returns the first truthy value OR the last falsy value.
And = returns the first falsy value OR the last truthy value.
Precedence of && is Higher then ||

A double not !! Is sometimes used to convert a value to boolean type:

!!”string” // true



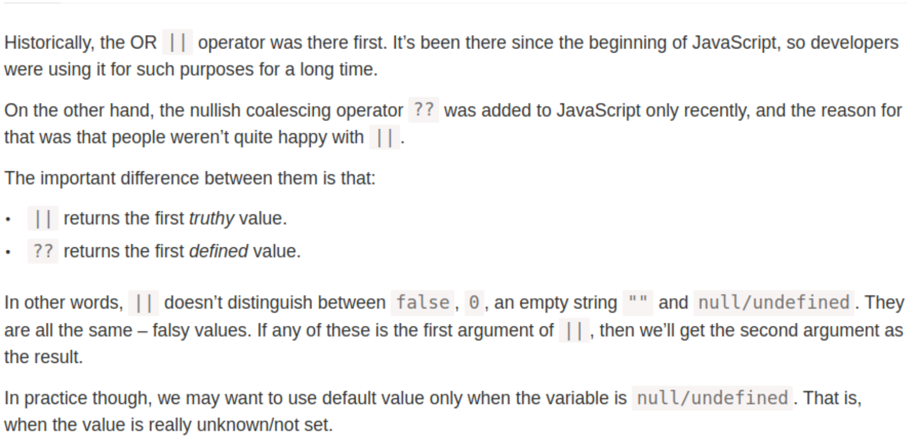
Summary

- The nullish coalescing operator `??` provides a short way to choose the first “defined” value from a list.

  It’s used to assign default values to variables:

  ```
  // set height=100, if height is null or undefined
  height = height ?? 100;
  ```

- The operator `??` has a very low precedence, only a bit higher than `?` and `=`, so consider adding parentheses when using it in an expression.
- It’s forbidden to use it with `||` or `&&` without explicit parentheses.



Historically, the OR `||` operator was there first. It’s been there since the beginning of JavaScript, so developers were using it for such purposes for a long time.

On the other hand, the nullish coalescing operator `??` was added to JavaScript only recently, and the reason for that was that people weren’t quite happy with `||`.

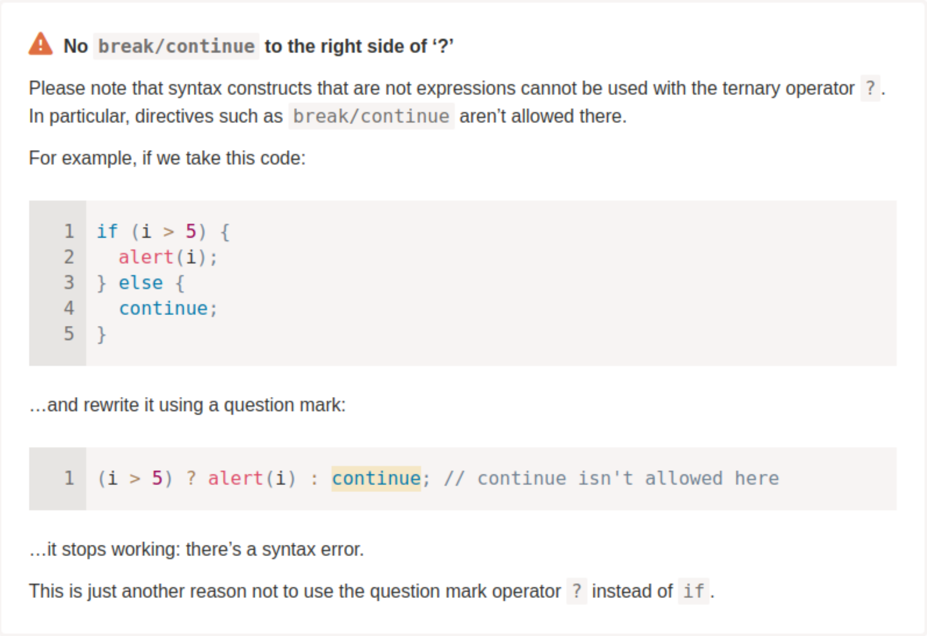
The important difference between them is that:

- `||` returns the first *truthy* value.
- `??` returns the first *defined* value.

In other words, `||` doesn’t distinguish between `false`, `0`, an empty string `""` and `null/undefined`. They are all the same – falsy values. If any of these is the first argument of `||`, then we’ll get the second argument as the result.

In practice though, we may want to use default value only when the variable is `null/undefined`. That is, when the value is really unknown/not set.

Loops:



⚠ No `break/continue` to the right side of '?'

Please note that syntax constructs that are not expressions cannot be used with the ternary operator `?`. In particular, directives such as `break/continue` aren't allowed there.

For example, if we take this code:

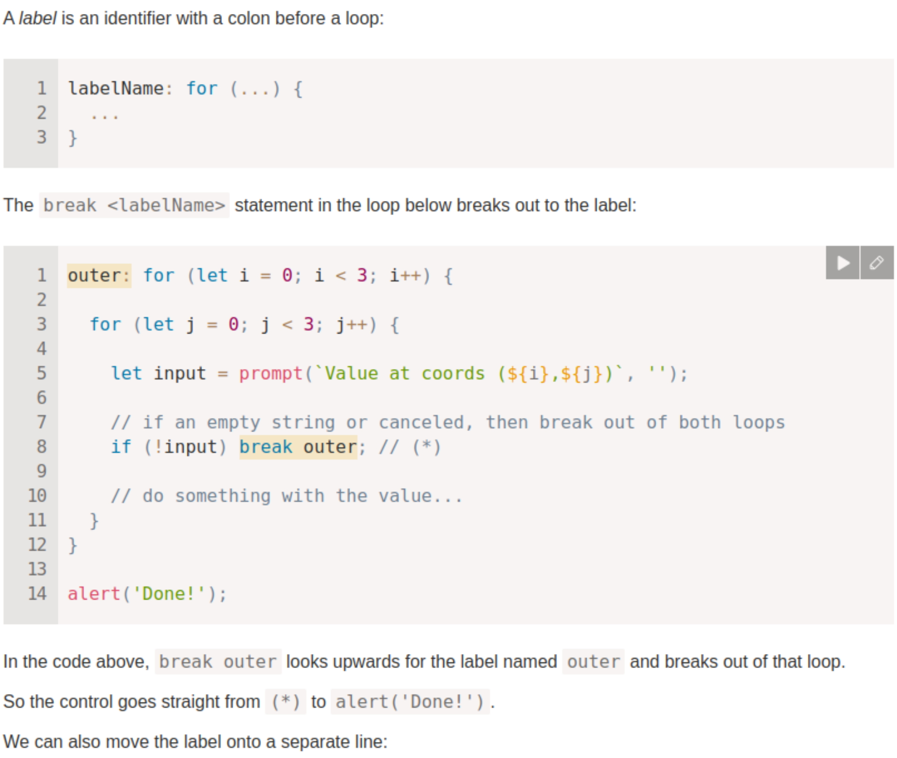
```
if (i > 5) {
  alert(i);
} else {
  continue;
}
```

…and rewrite it using a question mark:

```
(i > 5) ? alert(i) : continue; // continue isn't allowed here
```

…it stops working: there's a syntax error.

This is just another reason not to use the question mark operator `?` instead of `if`.



A *label* is an identifier with a colon before a loop:

```
labelName: for (...) {
  ...
}
```

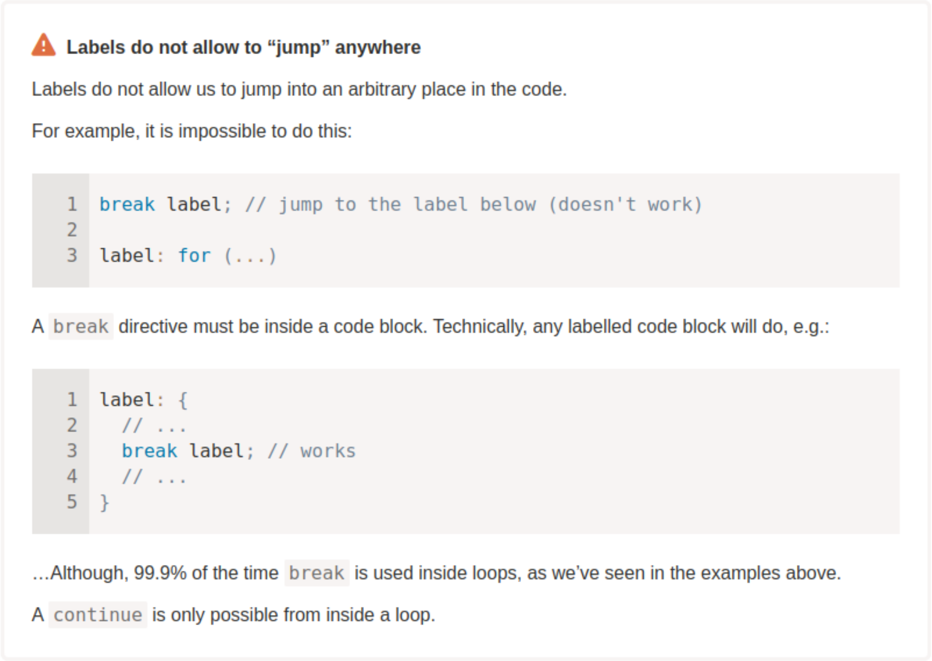
The `break <labelName>` statement in the loop below breaks out to the label:

```
outer: for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 3; j++) {

    let input = prompt(`Value at coords (${i},${j})`, '');

    // if an empty string or canceled, then break out of both loops
    if (!input) break outer; // (*)

    // do something with the value...
  }
}

alert('Done!');
```

In the code above, `break outer` looks upwards for the label named `outer` and breaks out of that loop.

So the control goes straight from `(*)` to `alert('Done!')`.

We can also move the label onto a separate line:



Labels do not allow to “jump” anywhere

Labels do not allow us to jump into an arbitrary place in the code.

For example, it is impossible to do this:

```
break label; // jump to the label below (doesn't work)

label: for (...)
```

A `break` directive must be inside a code block. Technically, any labelled code block will do, e.g.:

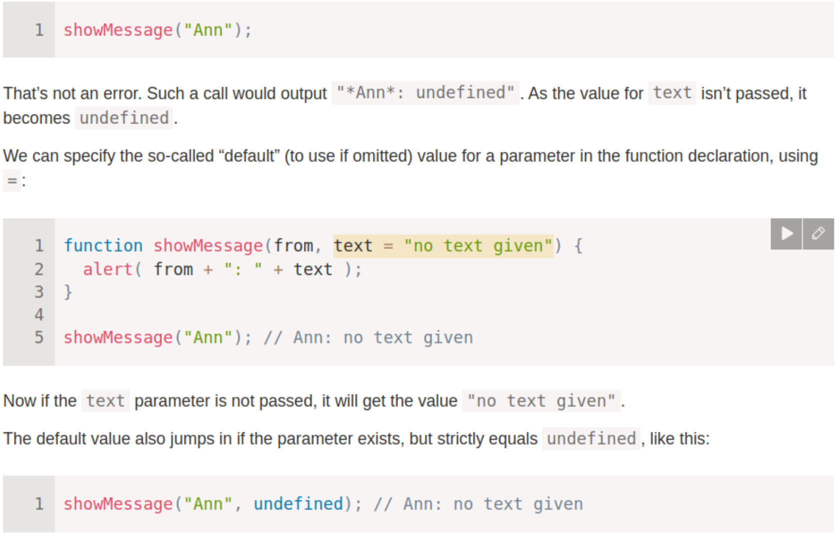
```
label: {
  // ...
  break label; // works
  // ...
}
```

…Although, 99.9% of the time `break` is used inside loops, as we’ve seen in the examples above.

A `continue` is only possible from inside a loop.

Functions:

The functions have a full access to the global vars and they can modify them too.



```
showMessage("Ann");
```

That’s not an error. Such a call would output `"*Ann*: undefined"`. As the value for `text` isn’t passed, it becomes `undefined`.

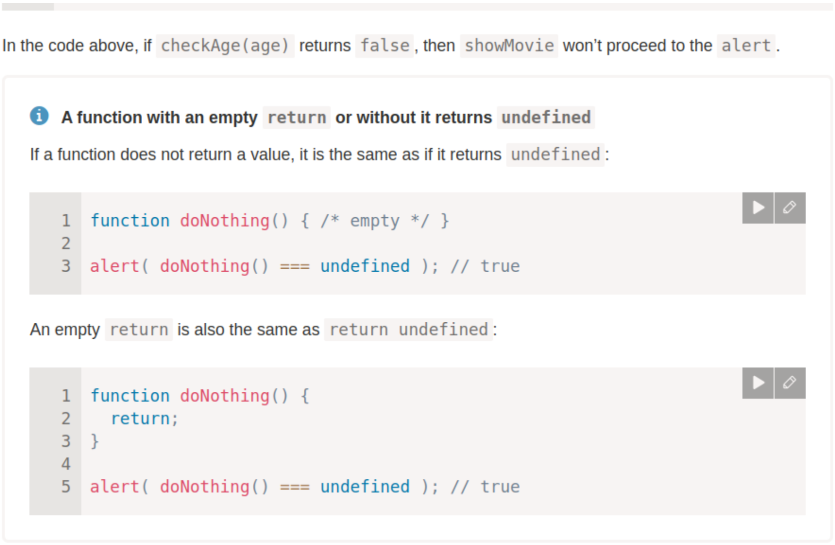
We can specify the so-called “default” (to use if omitted) value for a parameter in the function declaration, using `=`:

```
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}

showMessage("Ann"); // Ann: no text given
```

Now if the `text` parameter is not passed, it will get the value `"no text given"`.

The default value also jumps in if the parameter exists, but strictly equals `undefined`, like this:

```
showMessage("Ann", undefined); // Ann: no text given
```



In the code above, if `checkAge(age)` returns `false`, then `showMovie` won't proceed to the `alert`.

A function with an empty `return` or without it returns `undefined`

If a function does not return a value, it is the same as if it returns `undefined`:

```
function doNothing() { /* empty */ }

alert( doNothing() === undefined ); // true
```

An empty `return` is also the same as `return undefined`:

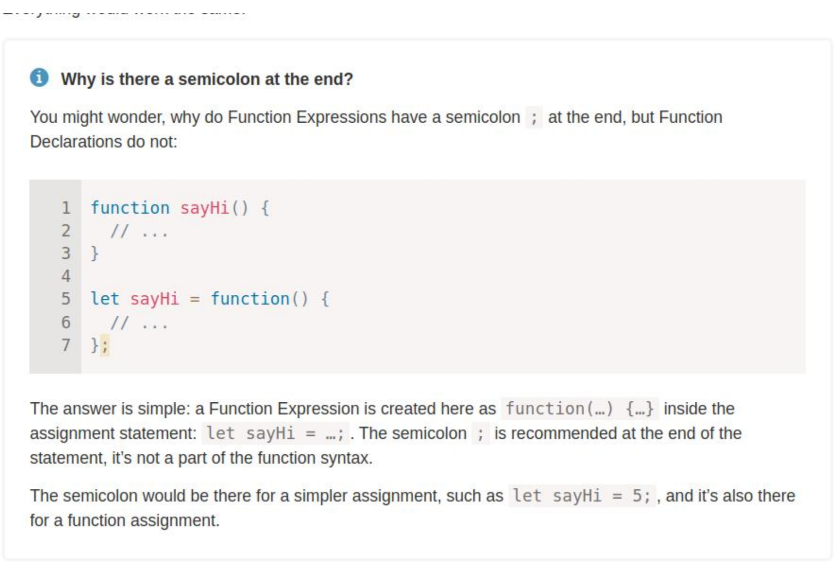
```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

Function Expressions:

Creating a function and put it into a variable.

In JavaScript, a function is a VALUE.

A function is a value representing an action just like a string or num represents data.



Everything would work the same.

Why is there a semicolon at the end?

You might wonder, why do Function Expressions have a semicolon `;` at the end, but Function Declarations do not:

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

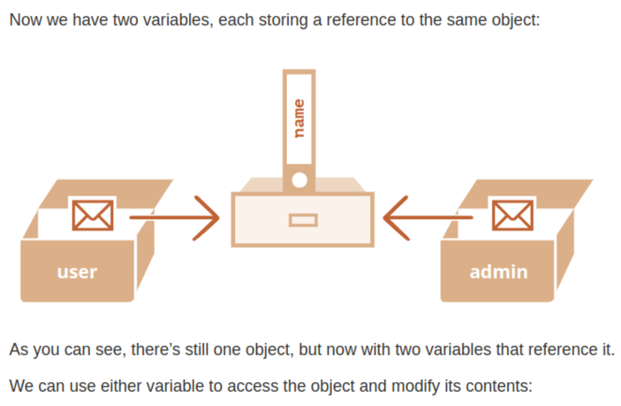
The answer is simple: a Function Expression is created here as `function(…) {…}` inside the assignment statement: `let sayHi = …;`. The semicolon `;` is recommended at the end of the statement, it's not a part of the function syntax.

The semicolon would be there for a simpler assignment, such as `let sayHi = 5;`, and it's also there for a function assignment.

Callback fns: fns that are passed as arguments.

Objects:

Objects are copied via reference i.e. both the variables will refer to same memory location. A copy of object is NOT made.



Now we have two variables, each storing a reference to the same object:

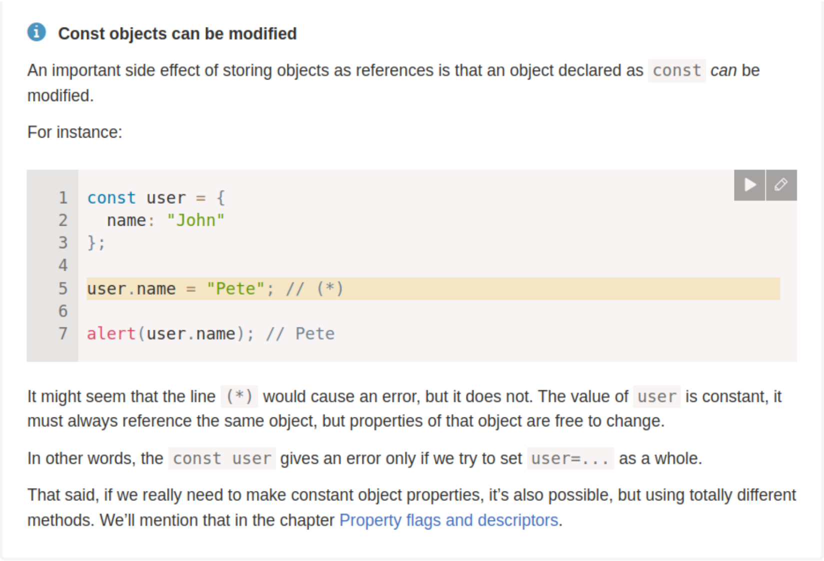
As you can see, there’s still one object, but now with two variables that reference it.

We can use either variable to access the object and modify its contents:

Two objects are equal only if they refer to the same memory location.

Pass by value: most times when we copy a varaible or pass params to a function, they are passed as a COPY of the original.

Pass by ref: the data types Objects and Arrays are copied/passed as reference. I.e. no new object is created for only the reference to the object is shared.



Const objects can be modified

An important side effect of storing objects as references is that an object declared as `const` *can* be modified.

For instance:

```
const user = {
  name: "John"
};

user.name = "Pete"; // (*)

alert(user.name); // Pete
```

It might seem that the line `(*)` would cause an error, but it does not. The value of `user` is constant, it must always reference the same object, but properties of that object are free to change.

In other words, the `const user` gives an error only if we try to set `user=...` as a whole.

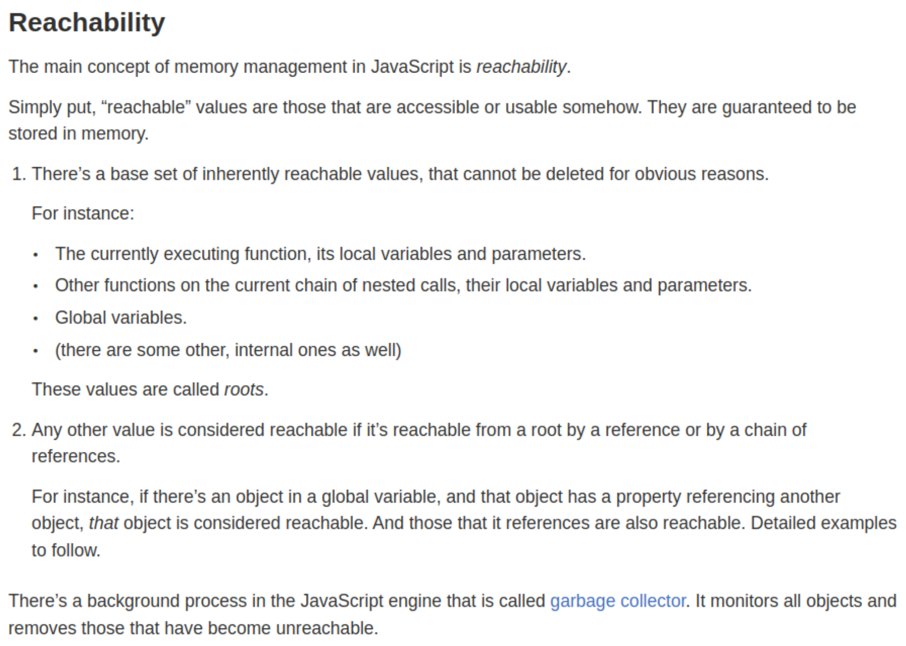
That said, if we really need to make constant object properties, it’s also possible, but using totally different methods. We’ll mention that in the chapter Property flags and descriptors.

We can clone an object using Object.assign(dest, ...srcs). It returns the destination object. It creates a shallow copy i.e nested objects are copied by reference.

Deep cloning: to check for nested objects and then clone them too.

Garbage Collection:

The main concept of mem mangnt in JS is reachability.



Reachability

The main concept of memory management in JavaScript is *reachability*.

Simply put, “reachable” values are those that are accessible or usable somehow. They are guaranteed to be stored in memory.

1. There’s a base set of inherently reachable values, that cannot be deleted for obvious reasons.

   For instance:

   - The currently executing function, its local variables and parameters.
   - Other functions on the current chain of nested calls, their local variables and parameters.
   - Global variables.
   - (there are some other, internal ones as well)

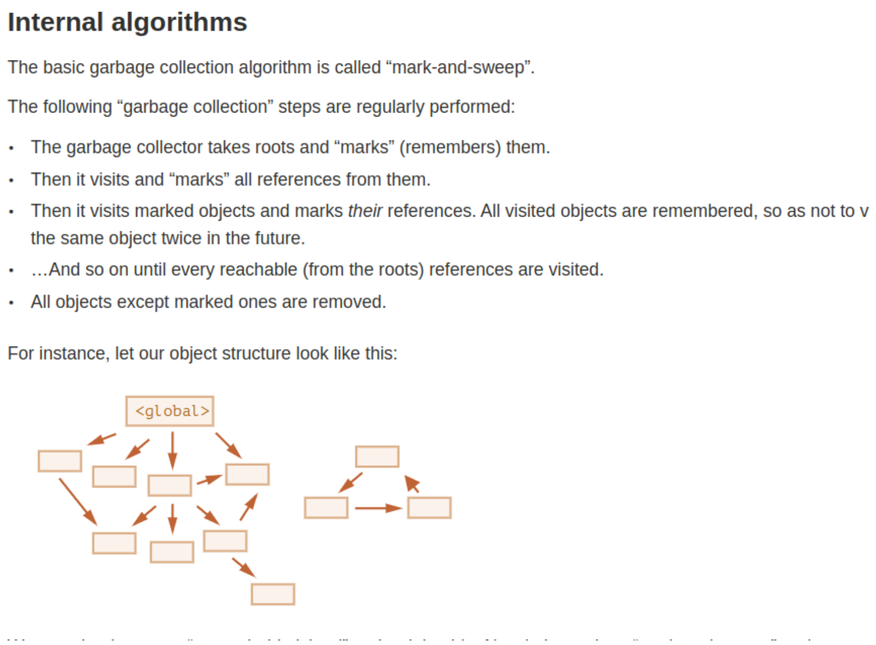
   These values are called *roots*.

2. Any other value is considered reachable if it’s reachable from a root by a reference or by a chain of references.

   For instance, if there’s an object in a global variable, and that object has a property referencing another object, *that* object is considered reachable. And those that it references are also reachable. Detailed examples to follow.

There’s a background process in the JavaScript engine that is called garbage collector. It monitors all objects and removes those that have become unreachable.

All the references in the roots are considered as reachable.



Internal algorithms

The basic garbage collection algorithm is called “mark-and-sweep”.

The following “garbage collection” steps are regularly performed:

- The garbage collector takes roots and “marks” (remembers) them.
- Then it visits and “marks” all references from them.
- Then it visits marked objects and marks *their* references. All visited objects are remembered, so as not to v the same object twice in the future.
- …And so on until every reachable (from the roots) references are visited.
- All objects except marked ones are removed.

For instance, let our object structure look like this:

Object Methods, This:

Actions r represented in JS by functions in properties.

A function that is a property of an obj is called it’s method.

To access the parent obj, the method can use **this** keyword.

We can also access the obj by using it’s name like user.age but that is Unreliable as we may change the reference value of user like user = null;

Arrow Funcs have NO this. They use the reference to the Outer Normal Function.

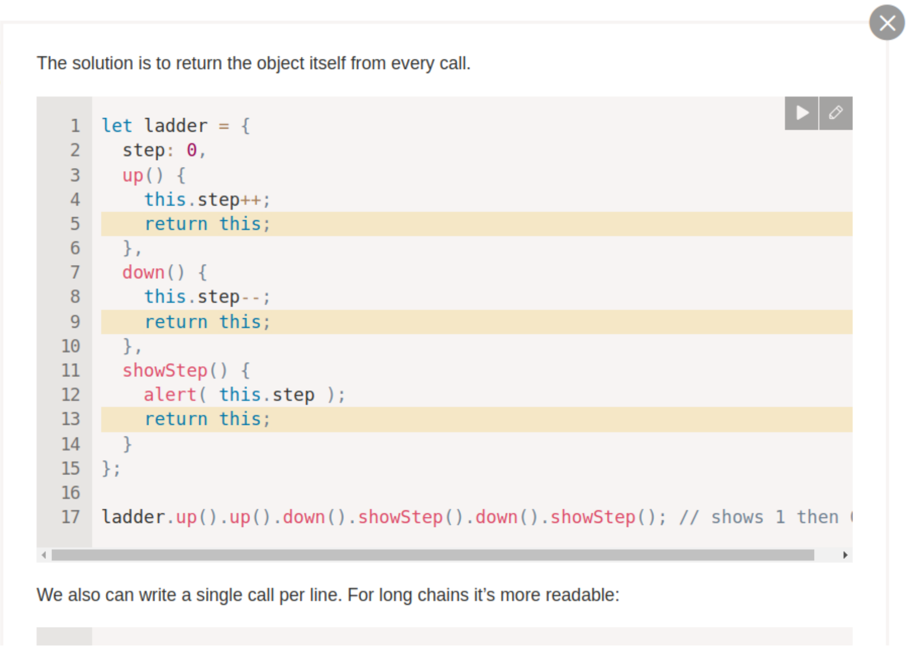
The moment at which this is called matters. Suppose we do

```
function makeUser() {

      return {

            name: "John",

            ref: this

      };

}
```

Here the this keyword is called as a FUNCTION and NOT as a method i.e. this.name so it does NOT assign any value.

To use this, we need to define OR return it Inside a method.

Chaining:



The solution is to return the object itself from every call.

```
let ladder = {
  step: 0,
  up() {
    this.step++;
    return this;
  },
  down() {
    this.step--;
    return this;
  },
  showStep() {
    alert( this.step );
    return this;
  }
};

ladder.up().up().down().showStep().down().showStep(); // shows 1 then
```

We also can write a single call per line. For long chains it's more readable:

Constructors:

First letter of object must be capital and must always be used with the 'new' keyword. Note that the capital letter convention is not compulsory.

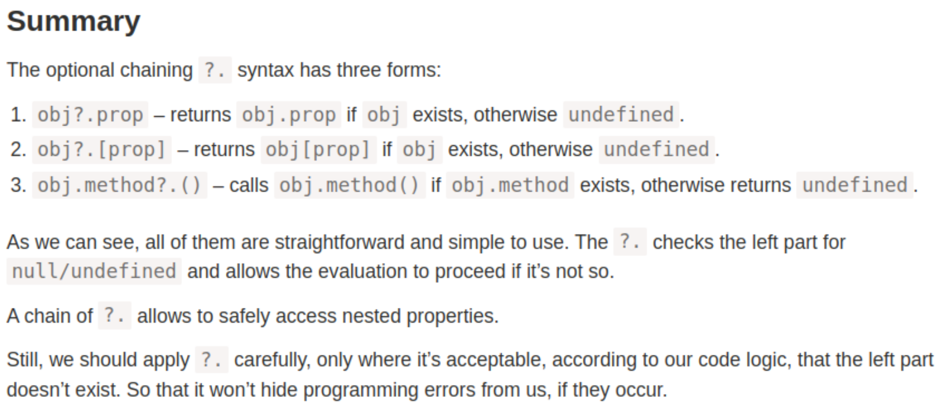
ANY function can be used as a constructor using new, it does the same process.

New creates an empty object, references it to this and then returns this.

New.target can be used Inside a function to know whether it is created using new keyword or not. Returns undefined for false and the function itself for true.

Usually constructors do not have a return type. But we can return an object, in this case the reference of return obj is assigned and not the this.

Optional chaining



Summary

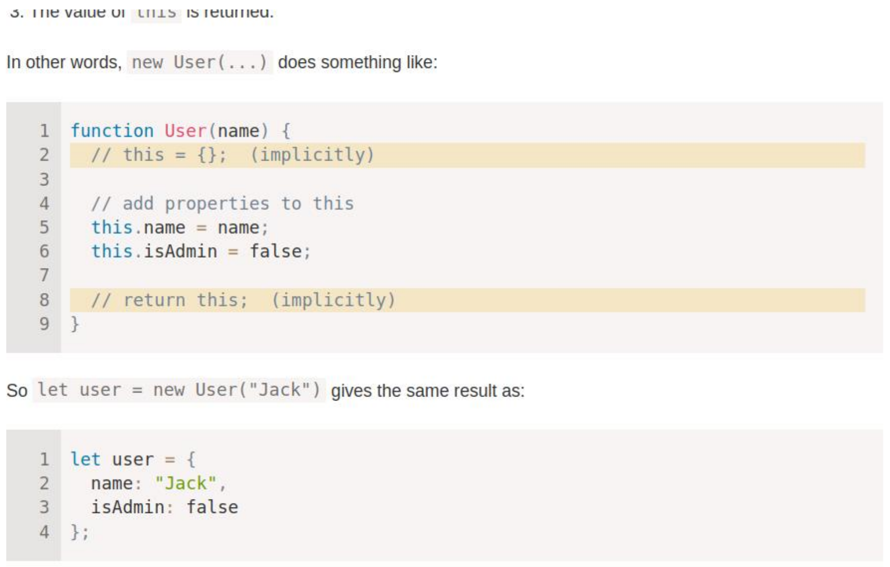
The optional chaining `?.` syntax has three forms:

1. `obj?.prop` – returns `obj.prop` if `obj` exists, otherwise `undefined`.
2. `obj?.[prop]` – returns `obj[prop]` if `obj` exists, otherwise `undefined`.
3. `obj.method?.()` – calls `obj.method()` if `obj.method` exists, otherwise returns `undefined`.

As we can see, all of them are straightforward and simple to use. The `?.` checks the left part for `null/undefined` and allows the evaluation to proceed if it's not so.

A chain of `?.` allows to safely access nested properties.

Still, we should apply `?.` carefully, only where it's acceptable, according to our code logic, that the left part doesn't exist. So that it won't hide programming errors from us, if they occur.



3. The value of `this` is returned.

In other words, `new User(...)` does something like:

```
function User(name) {
  // this = {};  (implicitly)

  // add properties to this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitly)
}
```

So `let user = new User("Jack")` gives the same result as:

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

Symbols are unique values that can have an optional description.

Let user = Symbol("id");

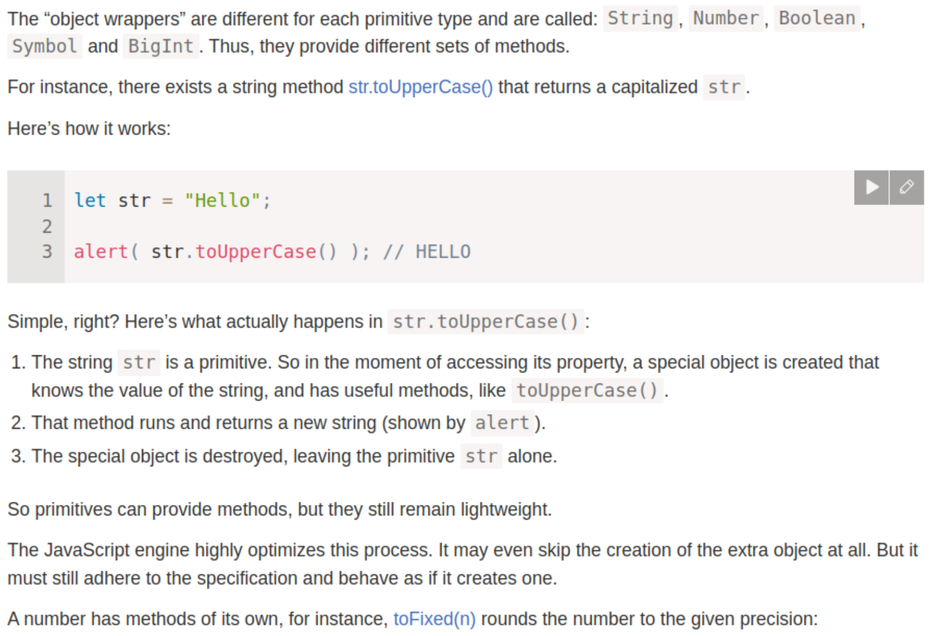
No to symbols can equal each other.

We can use either String OR a Symbol ([key] : value) as the key in an obj but no other data type.

Symbols are skipped in the for...in loop.

The result of obj1 + obj2 or any other Math operation can NEVER be another object

Methods as primitives:



The “object wrappers” are different for each primitive type and are called: `String`, `Number`, `Boolean`, `Symbol` and `BigInt`. Thus, they provide different sets of methods.

For instance, there exists a string method str.toUpperCase() that returns a capitalized `str`.

Here’s how it works:

```
let str = "Hello";

alert( str.toUpperCase() ); // HELLO
```

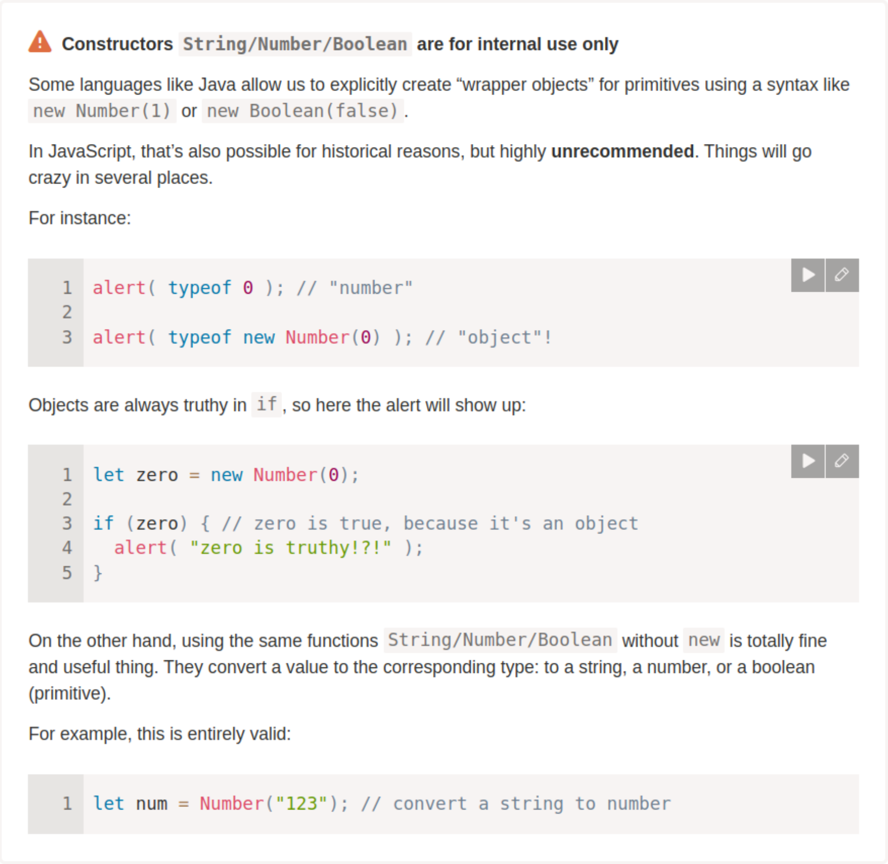
Simple, right? Here’s what actually happens in `str.toUpperCase()`:

1. The string `str` is a primitive. So in the moment of accessing its property, a special object is created that knows the value of the string, and has useful methods, like `toUpperCase()`.
2. That method runs and returns a new string (shown by `alert`).
3. The special object is destroyed, leaving the primitive `str` alone.

So primitives can provide methods, but they still remain lightweight.

The JavaScript engine highly optimizes this process. It may even skip the creation of the extra object at all. But it must still adhere to the specification and behave as if it creates one.

A number has methods of its own, for instance, toFixed(n) rounds the number to the given precision:



Constructors `String/Number/Boolean` are for internal use only

Some languages like Java allow us to explicitly create “wrapper objects” for primitives using a syntax like `new Number(1)` or `new Boolean(false)`.

In JavaScript, that’s also possible for historical reasons, but highly **unrecommended**. Things will go crazy in several places.

For instance:

```
alert( typeof 0 ); // "number"

alert( typeof new Number(0) ); // "object"!
```

Objects are always truthy in `if`, so here the alert will show up:

```
let zero = new Number(0);

if (zero) { // zero is true, because it's an object
  alert( "zero is truthy!?!" );
}
```

On the other hand, using the same functions `String/Number/Boolean` without `new` is totally fine and useful thing. They convert a value to the corresponding type: to a string, a number, or a boolean (primitive).

For example, this is entirely valid:

```
let num = Number("123"); // convert a string to number
```

Null and undefined primitives have no methods.

Whenever we use a dot with the primitives, its wrapper obj is Created.

Numbers:

Ways of writing a number:

A = 1_00_000 // underscore is ignored

A = 1e4 // similar to 10000 i.e. 4 zeros

A = 1e-4 // 0.0001

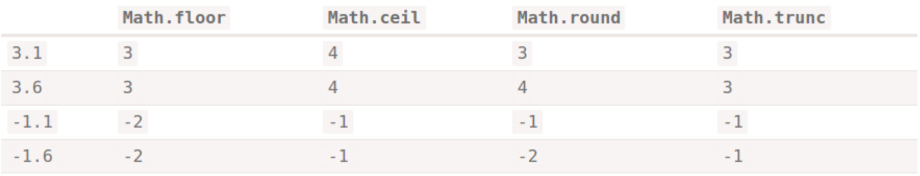
A = 0xNum / 0bNum / 0oNum // for hex, binary and octal

The toString(base) is used to convert the num to a required base and return as a string.

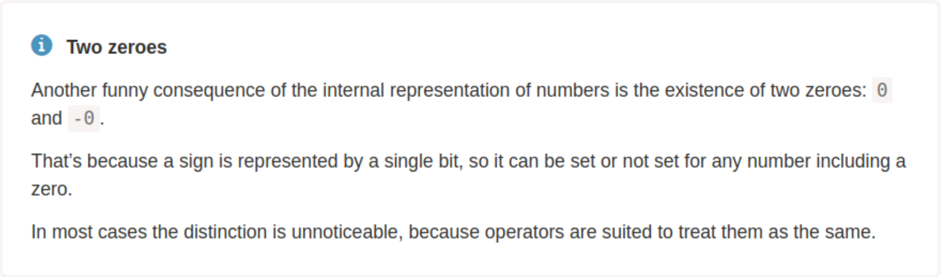
(123).toString(2)

123..toString(2) => a valid syntax

Rounding:



	`Math.floor`	`Math.ceil`	`Math.round`	`Math.trunc`
`3.1`	`3`	`4`	`3`	`3`
`3.6`	`3`	`4`	`4`	`3`
`-1.1`	`-2`	`-1`	`-1`	`-1`
`-1.6`	`-2`	`-1`	`-2`	`-1`



ℹ **Two zeroes**

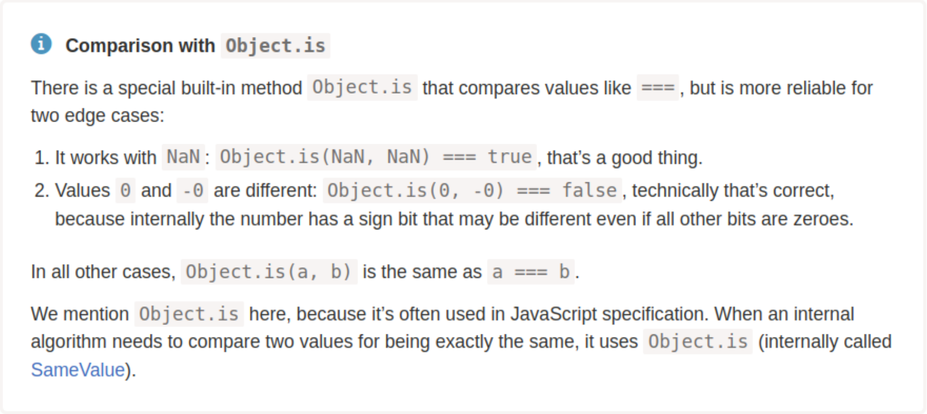
Another funny consequence of the internal representation of numbers is the existence of two zeroes: `0` and `-0`.

That's because a sign is represented by a single bit, so it can be set or not set for any number including a zero.

In most cases the distinction is unnoticeable, because operators are suited to treat them as the same.

NaN === NaN is FALSE. Same for undefined they Only Equal Each Other

The isFinite and isNaN. IsFinite returns false for +/-infinite and NaN.



ℹ **Comparison with `Object.is`**

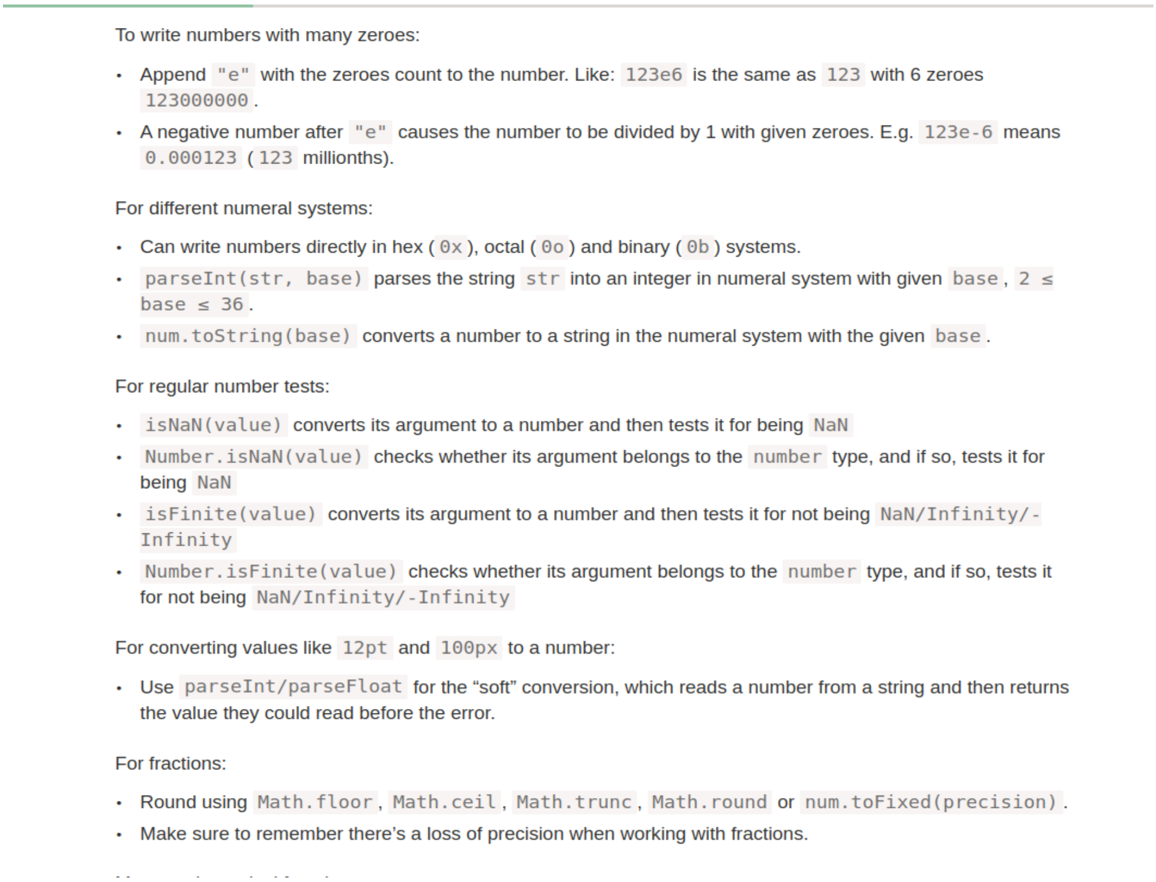
There is a special built-in method `Object.is` that compares values like `===`, but is more reliable for two edge cases:

1. It works with `NaN`: `Object.is(NaN, NaN) === true`, that's a good thing.
2. Values `0` and `-0` are different: `Object.is(0, -0) === false`, technically that's correct, because internally the number has a sign bit that may be different even if all other bits are zeroes.

In all other cases, `Object.is(a, b)` is the same as `a === b`.

We mention `Object.is` here, because it's often used in JavaScript specification. When an internal algorithm needs to compare two values for being exactly the same, it uses `Object.is` (internally called SameValue).

The parseInt and parseFloat returns the int and float part of the string.



To write numbers with many zeroes:

- Append `"e"` with the zeroes count to the number. Like: `123e6` is the same as `123` with 6 zeroes `123000000`.
- A negative number after `"e"` causes the number to be divided by 1 with given zeroes. E.g. `123e-6` means `0.000123` (`123` millionths).

For different numeral systems:

- Can write numbers directly in hex (`0x`), octal (`0o`) and binary (`0b`) systems.
- `parseInt(str, base)` parses the string `str` into an integer in numeral system with given `base`, `2 ≤ base ≤ 36`.
- `num.toString(base)` converts a number to a string in the numeral system with the given `base`.

For regular number tests:

- `isNaN(value)` converts its argument to a number and then tests it for being `NaN`
- `Number.isNaN(value)` checks whether its argument belongs to the `number` type, and if so, tests it for being `NaN`
- `isFinite(value)` converts its argument to a number and then tests it for not being `NaN/Infinity/-Infinity`
- `Number.isFinite(value)` checks whether its argument belongs to the `number` type, and if so, tests it for not being `NaN/Infinity/-Infinity`

For converting values like `12pt` and `100px` to a number:

- Use `parseInt/parseFloat` for the "soft" conversion, which reads a number from a string and then returns the value they could read before the error.

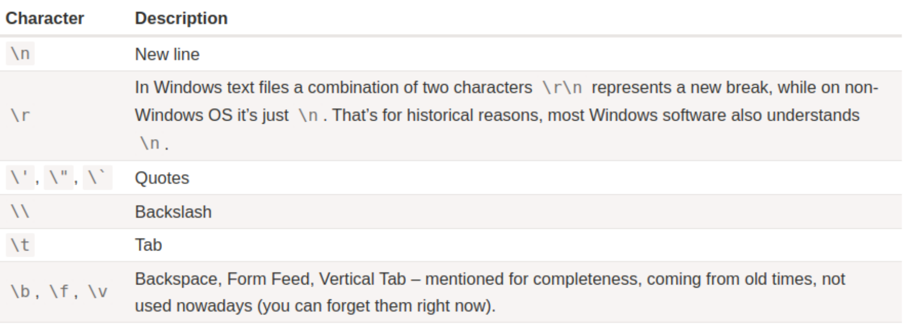
For fractions:

- Round using `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` or `num.toFixed(precision)`.
- Make sure to remember there's a loss of precision when working with fractions.

To fix the round off problem, multiply the number with the 10 power of the digit we need to round, apply round method and then divide it by the same number.

Strings:

Special chars:



Character	Description
`\n`	New line
`\r`	In Windows text files a combination of two characters `\r\n` represents a new break, while on non-Windows OS it's just `\n`. That's for historical reasons, most Windows software also understands `\n`.
`\'`, `\"`, ``\` ``	Quotes
`\\`	Backslash
`\t`	Tab
`\b`, `\f`, `\v`	Backspace, Form Feed, Vertical Tab – mentioned for completeness, coming from old times, not used nowadays (you can forget them right now).

Length is a property in JS and not a functoin. “str”.length

Accessing chars:

Str.at(-1) // allowed

Str[-1] // not allowed

Strings are immutable.

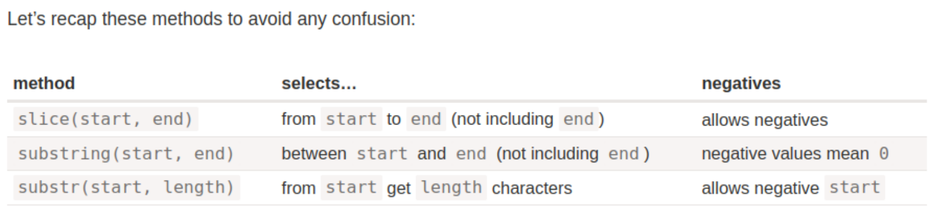
Can change strs or chars(str[0]) .toUpperCase or .toLowerCase

.indexOf(‘str’, pos(def 0)) // returns –1 for no occurence and index of 1st occurence from the given pos

.lastIndexOf => works from last to first

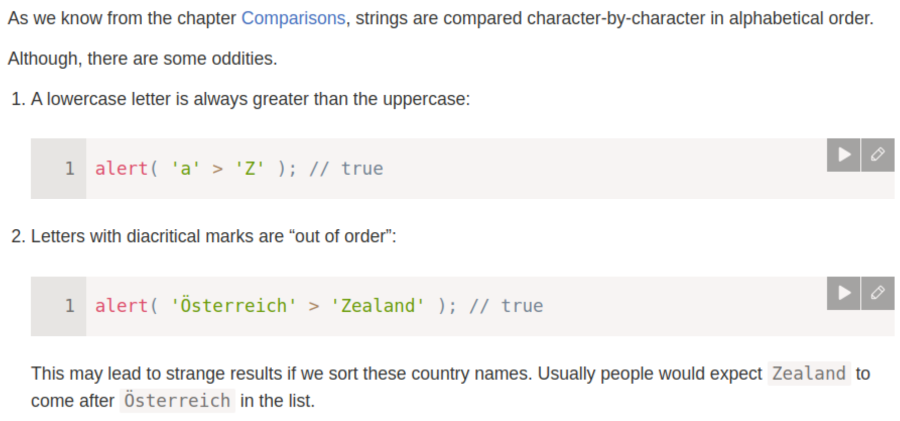
Str.slice(0,5) // str from 0 t 5 Excluding 5

Str.substring(5,2) // is same but allows start>end. It just swaps the nums



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` (not including `end`)	negative values mean `0`
`substr(start, length)`	from `start` get `length` characters	allows negative `start`



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:

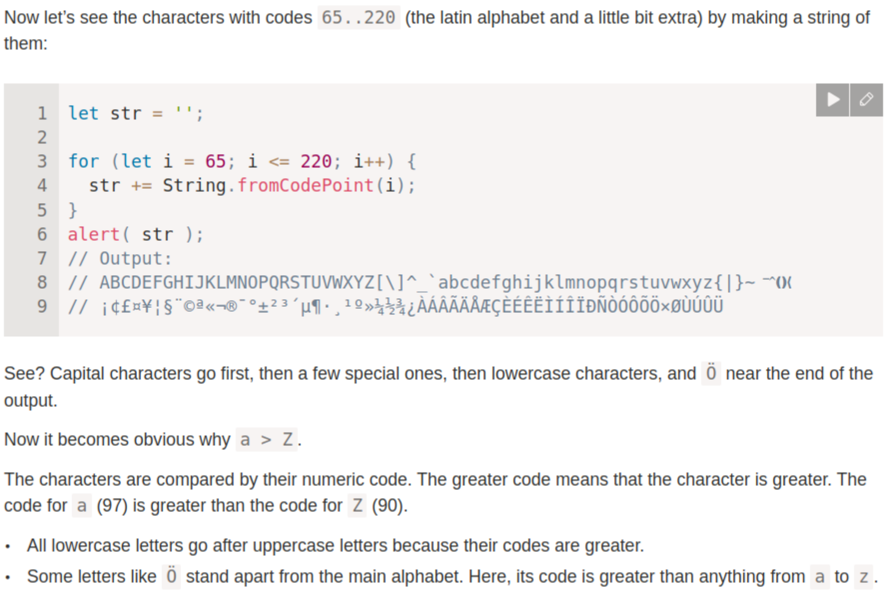
```
alert( 'a' > 'Z' ); // true
```

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.

Str.codePointAt(1) // returns code of char at 1



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 );
// Output:
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~ ¯^()(
// ¡¢£¤¥¦§¨©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`.

Arrays:

The new Array(num) creates an empty array with the given length.

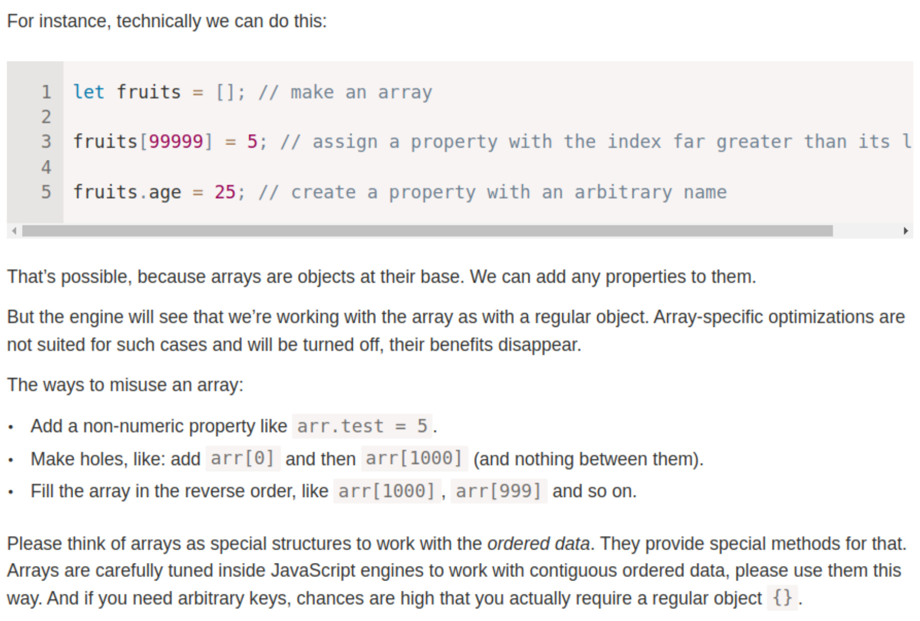
If we shorten the arr length manually, then the arr gets Truncated.

Arrays can also have elements of type Function.

Methods that works with the end of the array: pop(), push()

Methods that works with the start of the array: shift(), unshift()

Push and unshift can add multiple elems in the arr



For instance, technically we can do this:

```
let fruits = []; // make an array

fruits[99999] = 5; // assign a property with the index far greater than its l

fruits.age = 25; // create a property with an arbitrary name
```

That's possible, because arrays are objects at their base. We can add any properties to them.

But the engine will see that we're working with the array as with a regular object. Array-specific optimizations are not suited for such cases and will be turned off, their benefits disappear.

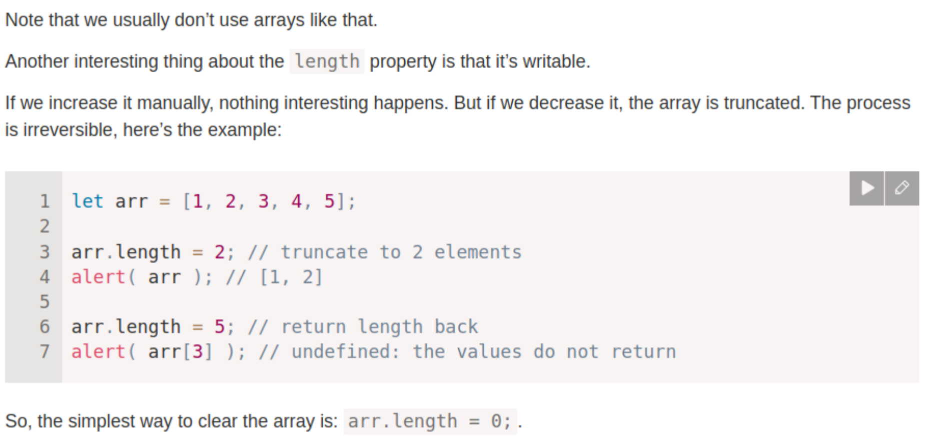
The ways to misuse an array:

- Add a non-numeric property like `arr.test = 5`.
- Make holes, like: add `arr[0]` and then `arr[1000]` (and nothing between them).
- Fill the array in the reverse order, like `arr[1000]`, `arr[999]` and so on.

Please think of arrays as special structures to work with the *ordered data*. They provide special methods for that. Arrays are carefully tuned inside JavaScript engines to work with contiguous ordered data, please use them this way. And if you need arbitrary keys, chances are high that you actually require a regular object `{}`.

Push and pop are Faster than shift and unshift.

To loop through the arrays, we can use for...in loop as arrays are Objs only. But it is not recommended, instead we should use for(let elem of arr){} loop(Here elem is the Value and not the index).



Note that we usually don't use arrays like that.

Another interesting thing about the `length` property is that it's writable.

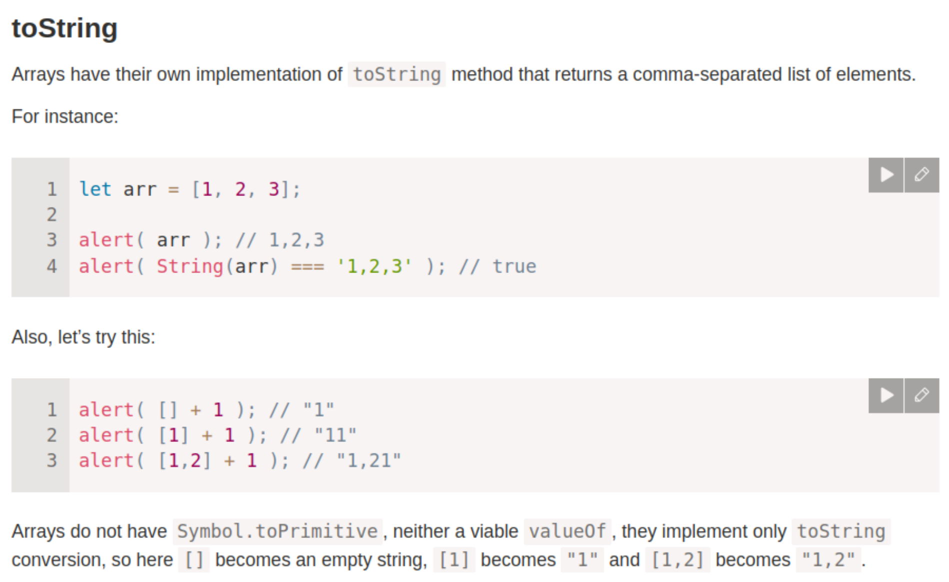
If we increase it manually, nothing interesting happens. But if we decrease it, the array is truncated. The process is irreversible, here's the example:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

So, the simplest way to clear the array is: `arr.length = 0;`.

To clear whole arr, just do arr.length =0;

Arrays have toString property.



toString

Arrays have their own implementation of `toString` method that returns a comma-separated list of elements.

For instance:

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

Also, let's try this:

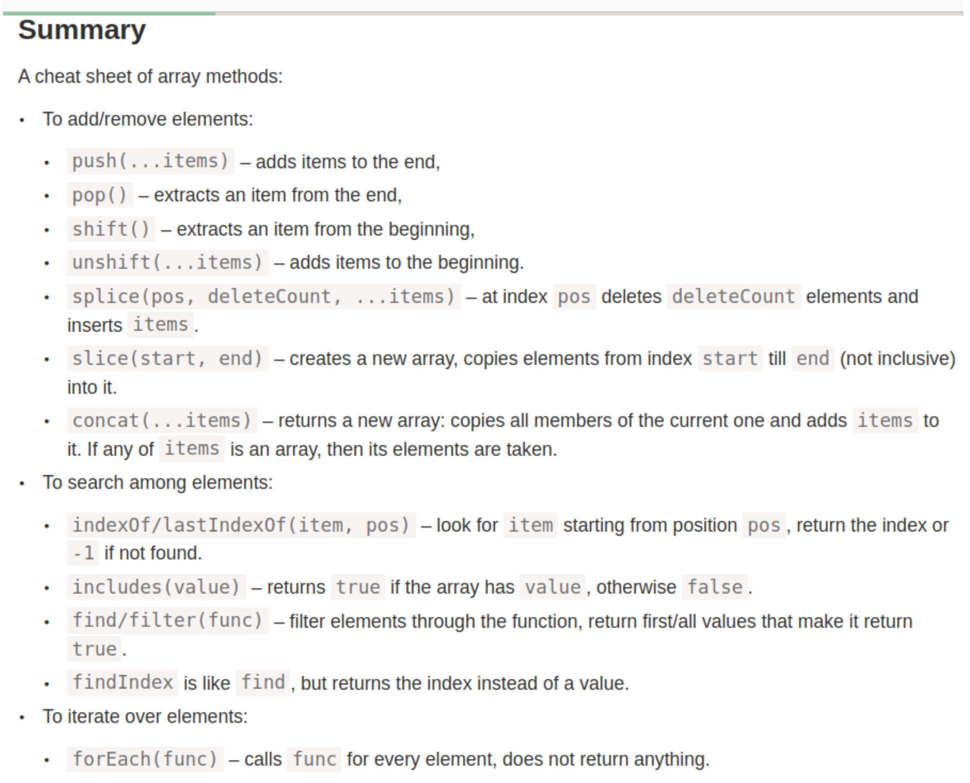
```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Arrays do not have `Symbol.toPrimitive`, neither a viable `valueOf`, they implement only `toString` conversion, so here `[]` becomes an empty string, `[1]` becomes `"1"` and `[1,2]` becomes `"1,2"`.

Array Methods:

Sort, reverse and splice modifies the arr Itself.

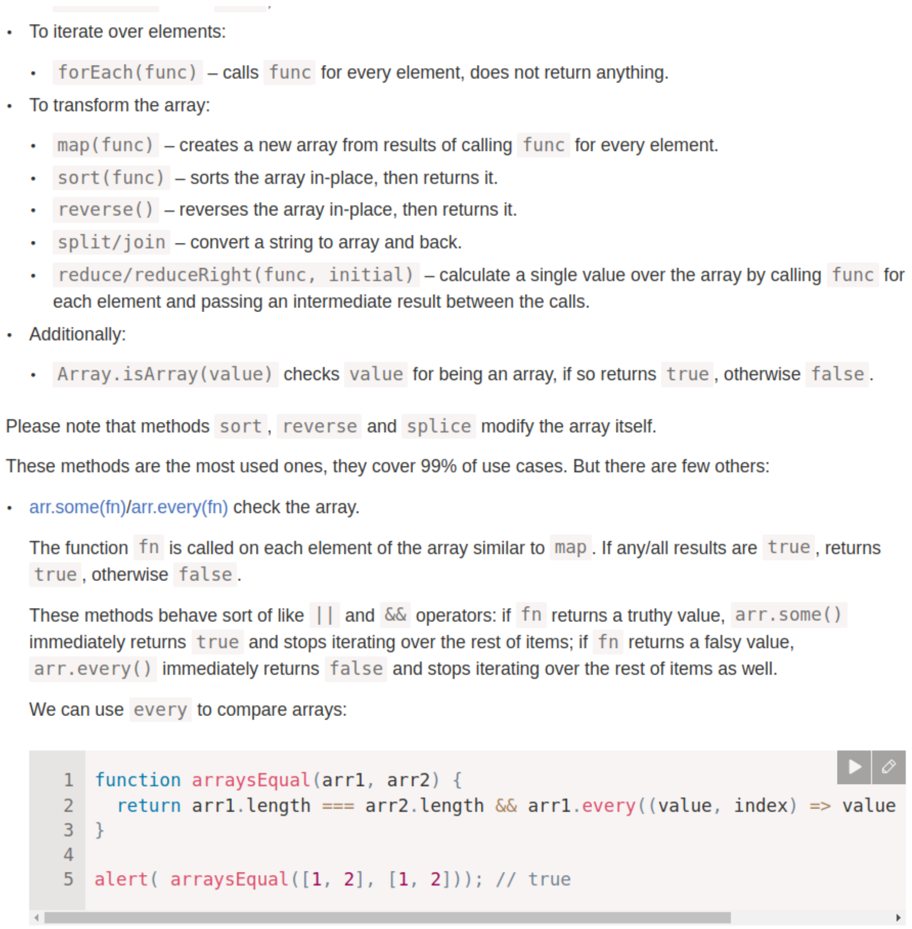
Concat, map, slice, filter creates a new arr.



Summary

A cheat sheet of array methods:

- To add/remove elements:
  - `push(...items)` – adds items to the end,
  - `pop()` – extracts an item from the end,
  - `shift()` – extracts an item from the beginning,
  - `unshift(...items)` – adds items to the beginning.
  - `splice(pos, deleteCount, ...items)` – at index `pos` deletes `deleteCount` elements and inserts `items`.
  - `slice(start, end)` – creates a new array, copies elements from index `start` till `end` (not inclusive) into it.
  - `concat(...items)` – returns a new array: copies all members of the current one and adds `items` to it. If any of `items` is an array, then its elements are taken.
- To search among elements:
  - `indexOf/lastIndexOf(item, pos)` – look for `item` starting from position `pos`, return the index or `-1` if not found.
  - `includes(value)` – returns `true` if the array has `value`, otherwise `false`.
  - `find/filter(func)` – filter elements through the function, return first/all values that make it return `true`.
  - `findIndex` is like `find`, but returns the index instead of a value.
- To iterate over elements:
  - `forEach(func)` – calls `func` for every element, does not return anything.



- To iterate over elements:
  - `forEach(func)` – calls `func` for every element, does not return anything.
- To transform the array:
  - `map(func)` – creates a new array from results of calling `func` for every element.
  - `sort(func)` – sorts the array in-place, then returns it.
  - `reverse()` – reverses the array in-place, then returns it.
  - `split/join` – convert a string to array and back.
  - `reduce/reduceRight(func, initial)` – calculate a single value over the array by calling `func` for each element and passing an intermediate result between the calls.
- Additionally:
  - `Array.isArray(value)` checks `value` for being an array, if so returns `true`, otherwise `false`.

Please note that methods `sort`, `reverse` and `splice` modify the array itself.

These methods are the most used ones, they cover 99% of use cases. But there are few others:

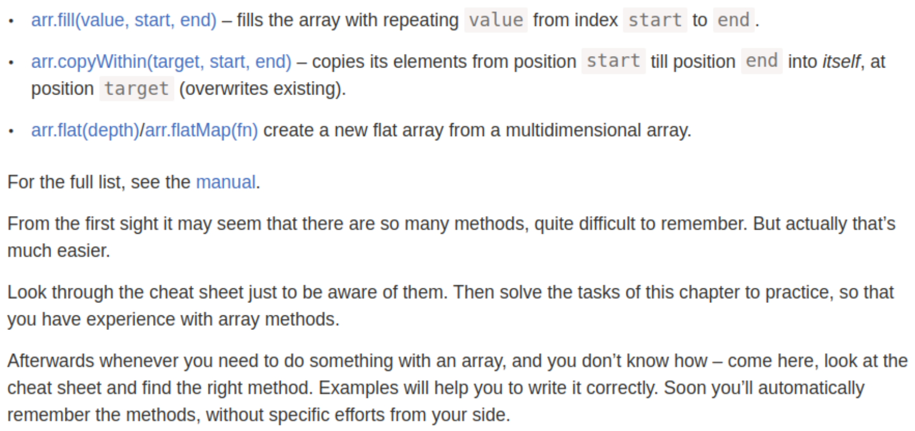
- arr.some(fn)/arr.every(fn) check the array.

  The function `fn` is called on each element of the array similar to `map`. If any/all results are `true`, returns `true`, otherwise `false`.

  These methods behave sort of like `||` and `&&` operators: if `fn` returns a truthy value, `arr.some()` immediately returns `true` and stops iterating over the rest of items; if `fn` returns a falsy value, `arr.every()` immediately returns `false` and stops iterating over the rest of items as well.

  We can use `every` to compare arrays:

```
function arraysEqual(arr1, arr2) {
  return arr1.length === arr2.length && arr1.every((value, index) => value
}

alert( arraysEqual([1, 2], [1, 2])); // true
```



- arr.fill(value, start, end) – fills the array with repeating `value` from index `start` to `end`.
- arr.copyWithin(target, start, end) – copies its elements from position `start` till position `end` into *itself*, at position `target` (overwrites existing).
- arr.flat(depth)/arr.flatMap(fn) create a new flat array from a multidimensional array.

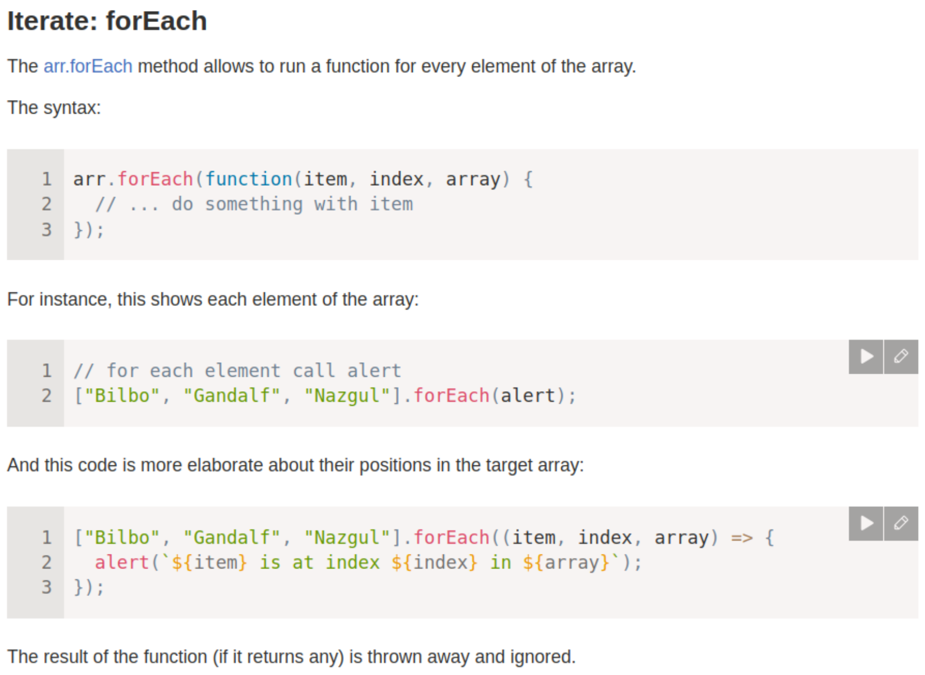
For the full list, see the manual.

From the first sight it may seem that there are so many methods, quite difficult to remember. But actually that's much easier.

Look through the cheat sheet just to be aware of them. Then solve the tasks of this chapter to practice, so that you have experience with array methods.

Afterwards whenever you need to do something with an array, and you don't know how – come here, look at the cheat sheet and find the right method. Examples will help you to write it correctly. Soon you'll automatically remember the methods, without specific efforts from your side.

Splice returns the deleted items and allows negative integers.



Iterate: forEach

The arr.forEach method allows to run a function for every element of the array.

The syntax:

```
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

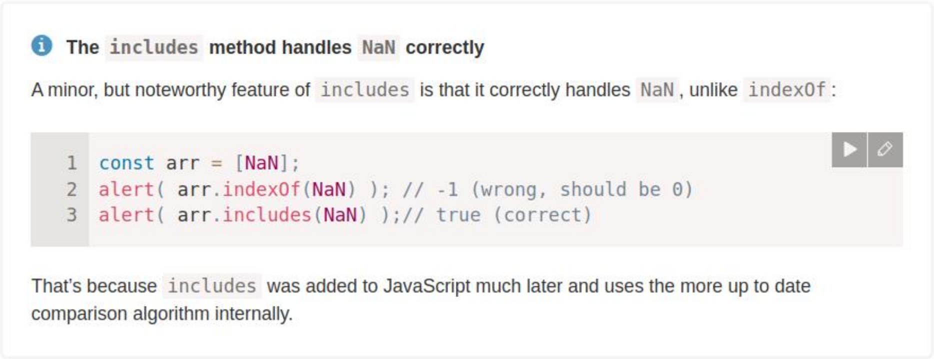
For instance, this shows each element of the array:

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

And this code is more elaborate about their positions in the target array:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} is at index ${index} in ${array}`);
});
```

The result of the function (if it returns any) is thrown away and ignored.

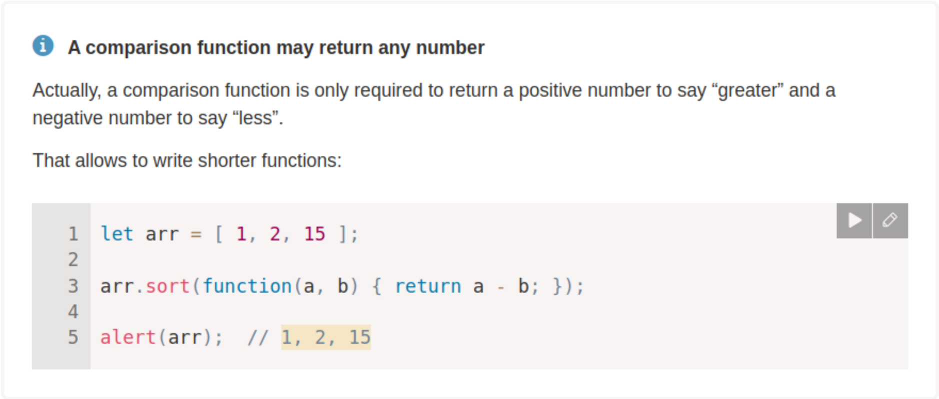


The `includes` method handles `NaN` correctly

A minor, but noteworthy feature of `includes` is that it correctly handles `NaN`, unlike `indexOf`:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (wrong, should be 0)
alert( arr.includes(NaN) );// true (correct)
```

That's because `includes` was added to JavaScript much later and uses the more up to date comparison algorithm internally.



A comparison function may return any number

Actually, a comparison function is only required to return a positive number to say "greater" and a negative number to say "less".

That allows to write shorter functions:

```
let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr);  // 1, 2, 15
```

Sort compares the array elems as string so the sorting may not be correct, so we may need to provide a comparison.

- Arr.splice(index, 1) to delete elem at index.
- Arr.slice() // creates a copy of arr.
- [Symbol.isConcatSpreadable] : true

Length : n

Treats the object as an array of length n.

Date and Time:

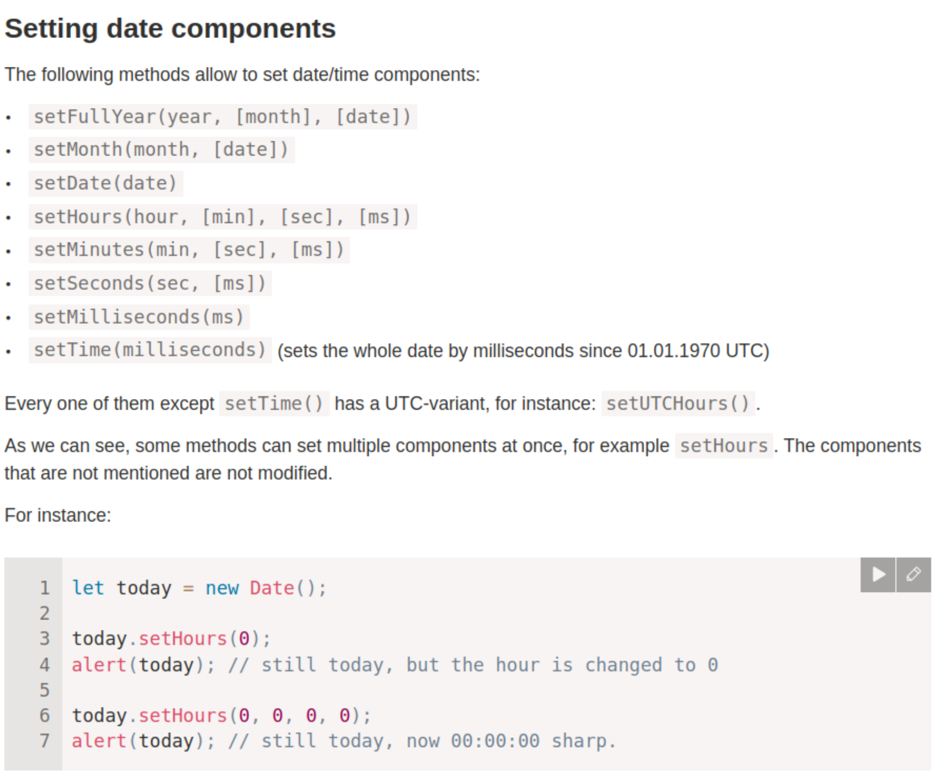
Date obj always carries both, date and time.

Months and days of week in getDay() are always counted from zero.

When dates are subtracted ans is in millisecs. That's coz date becomes timestamp on number conversion.

Date.now() returns the current timestamp

Putting negative numbers or zero in the setDate() will give the dates of prev month



Setting date components

The following methods allow to set date/time components:

- `setFullYear(year, [month], [date])`
- `setMonth(month, [date])`
- `setDate(date)`
- `setHours(hour, [min], [sec], [ms])`
- `setMinutes(min, [sec], [ms])`
- `setSeconds(sec, [ms])`
- `setMilliseconds(ms)`
- `setTime(milliseconds)` (sets the whole date by milliseconds since 01.01.1970 UTC)

Every one of them except `setTime()` has a UTC-variant, for instance: `setUTCHours()`.

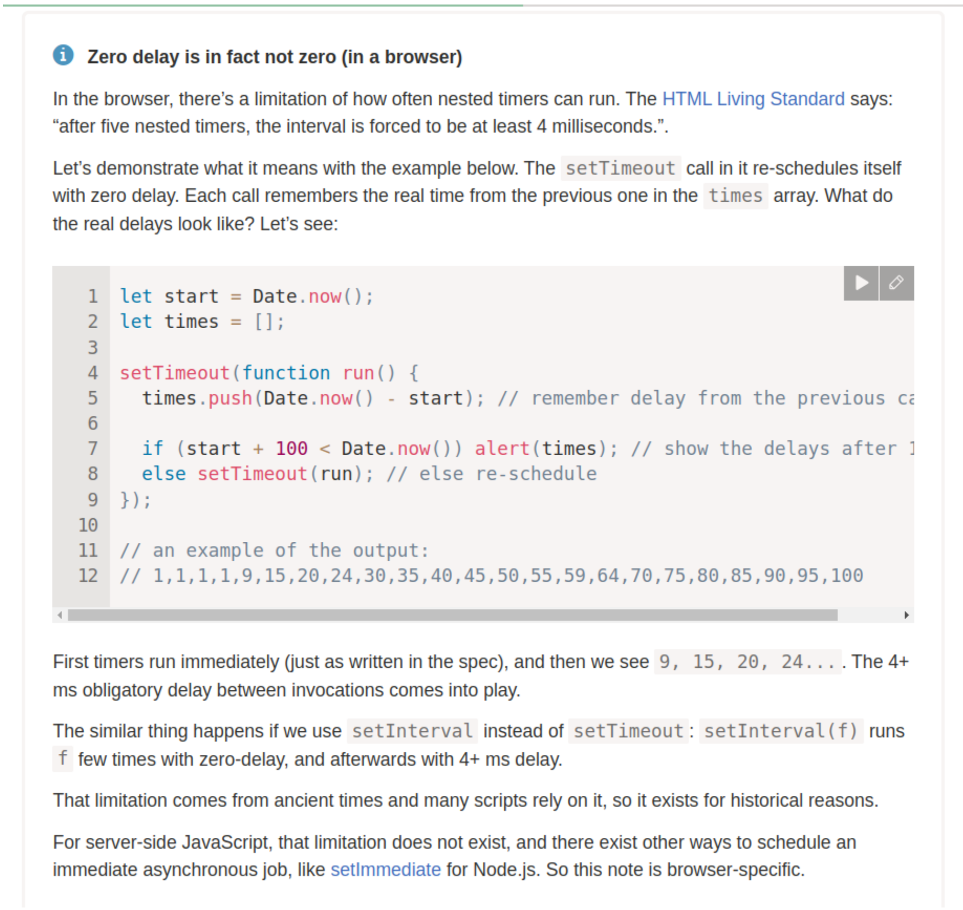
As we can see, some methods can set multiple components at once, for example `setHours`. The components that are not mentioned are not modified.

For instance:

```
let today = new Date();

today.setHours(0);
alert(today); // still today, but the hour is changed to 0

today.setHours(0, 0, 0, 0);
alert(today); // still today, now 00:00:00 sharp.
```

Set Timeout:

SetTimeout expects a reference of the function, i.e. W/O the () brackets.



Zero delay is in fact not zero (in a browser)

In the browser, there’s a limitation of how often nested timers can run. The HTML Living Standard says: “after five nested timers, the interval is forced to be at least 4 milliseconds.”.

Let’s demonstrate what it means with the example below. The `setTimeout` call in it re-schedules itself with zero delay. Each call remembers the real time from the previous one in the `times` array. What do the real delays look like? Let’s see:

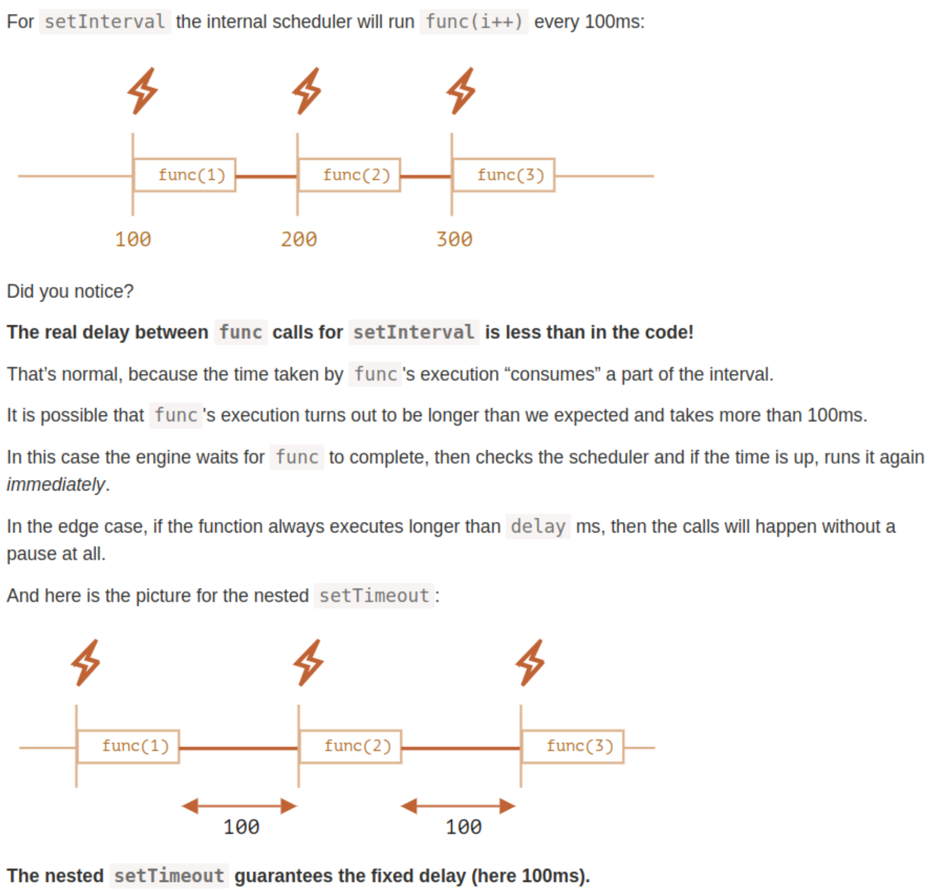
```
let start = Date.now();
let times = [];

setTimeout(function run() {
  times.push(Date.now() - start); // remember delay from the previous ca

  if (start + 100 < Date.now()) alert(times); // show the delays after 1
  else setTimeout(run); // else re-schedule
});

// an example of the output:
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

First timers run immediately (just as written in the spec), and then we see `9, 15, 20, 24...`. The 4+ ms obligatory delay between invocations comes into play.

The similar thing happens if we use `setInterval` instead of `setTimeout`: `setInterval(f)` runs `f` few times with zero-delay, and afterwards with 4+ ms delay.

That limitation comes from ancient times and many scripts rely on it, so it exists for historical reasons.

For server-side JavaScript, that limitation does not exist, and there exist other ways to schedule an immediate asynchronous job, like setImmediate for Node.js. So this note is browser-specific.



For `setInterval` the internal scheduler will run `func(i++)` every 100ms:

Did you notice?

The real delay between `func` calls for `setInterval` is less than in the code!

That's normal, because the time taken by `func`'s execution "consumes" a part of the interval.

It is possible that `func`'s execution turns out to be longer than we expected and takes more than 100ms.

In this case the engine waits for `func` to complete, then checks the scheduler and if the time is up, runs it again *immediately*.

In the edge case, if the function always executes longer than `delay` ms, then the calls will happen without a pause at all.

And here is the picture for the nested `setTimeout`:

The nested `setTimeout` guarantees the fixed delay (here 100ms).

Notes:

- JS does not assume a semicolon before [] i.e.

  Alert()

  [].forEach();

  JS interprets it as alert()[].forEach(); which causes error.

- Shift+enter to enter multiple lines in console.
- Empty string is converted to ZERO.
- **Null** becomes zero after the numeric conversion and **undefined** becomes NaN.
- Space chars(\t \n) are trimmed in the number conversion.
- Values null and undefined equals each other only.
- Alert returns undefined.
- JavaScript does NOT support operator overloading.