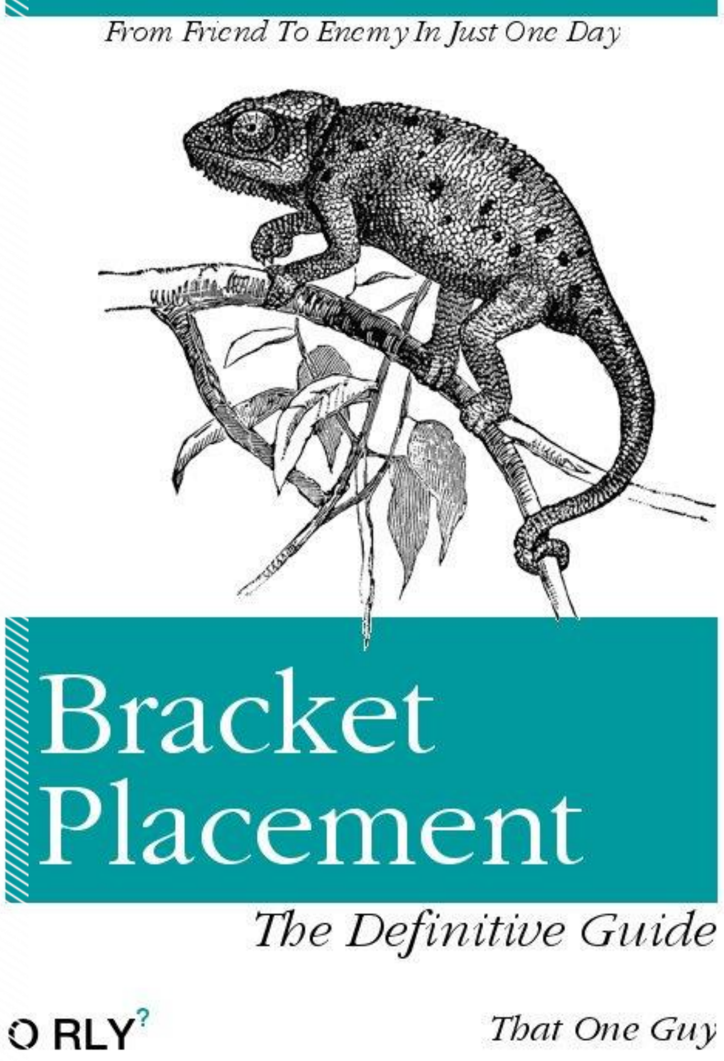
JS Data Types and Structures

Frontend Junior Program - 2022



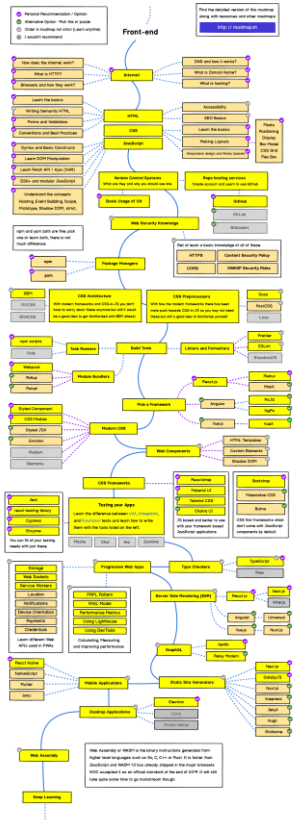
Agenda

Preface

Programming can be confusing sometimes – and that is especially true for JavaScript for several reasons. To survive the odyssey between Scylla and Charybdis a hero need to know: why?

- As the language of web, JavaScript was designed to be an effective and powerful, dynamically-typed language, and it has implicit type conversion.
- JavaScript is surrounded by its ecosystem - infinite number of ever-changing tools and frameworks, so it is easy to be eaten by the hydra.
- As the most used programming language in the world, the internet is overloaded by superficial and misleading information about it.

In the next slides we will eliminate all of these…



yes, these are the essentials, *"you need to know"* – they say

Step 1: implicit type conversion

For the applications, we develop, we usually don't need dynamic typing*. As a consequence, we simply should not use implicit type conversion.

"But creator of this mighty lecture," – you may ask – *"it is still the nature of the language, so how can we avoid?"*

The answer is: with proper program design,

careful development and rigorous code review.
That is, however, require experience, discipline, and still cost a lot and error prone (depends on human factor).

Also, with TypeScript.

Using tools for enforcing static typing could prevent from committing mistakes** in this regard.

JAVASCRIPT
ME
Strongly Typed inherently Safe Compiled Programming Languages

*academic and Brendan Eich (he is the culprit for JavaScript) uses *"untyped"*.

** we try to be precise here: it prevents mistakes, not bugs. TypeScript does not lead to significantly less bugs. A mistake could be, when we don't want to use a feature (because it could be problematic), still, we use it, accidentally.

TypeScript won't save you from bugs

Step 1: not anymore

Still, implicit type conversion is something you need to be aware of – just to be able to prevent that.

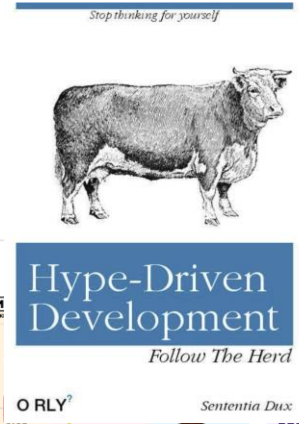
But you definitely don’t need to learn all of its hocus-pocus.

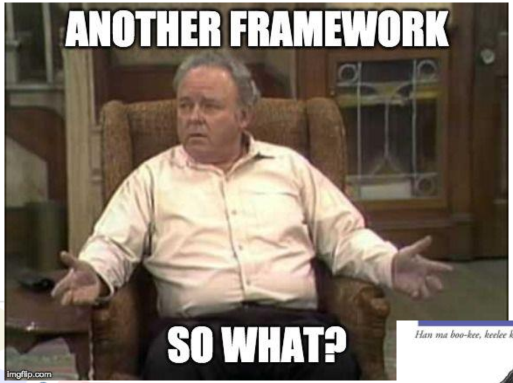
The JavaScript Standard is 860 pages long (in pdf) at this point. From this, 16 pages alone are dedicated to the implicit type conversion. Yes, I counted. ¯_(ツ)_/¯

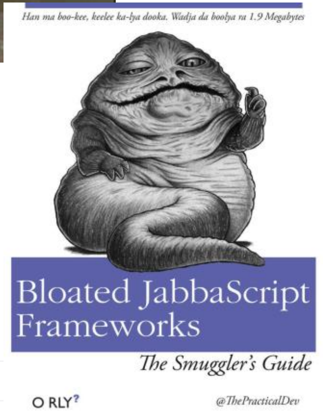
no, you won’t ;)

Step 2: The JavaScript Fatigue

CHOOSE YOUR OWN FRAMEWORK
HOW WILL YOUR PROJECT DIE?
CHOOSE FROM 2,147,483,647 POSSIBLE ENDINGS.
THE CURSE OF JAVASCRIPT FATIGUE
JS

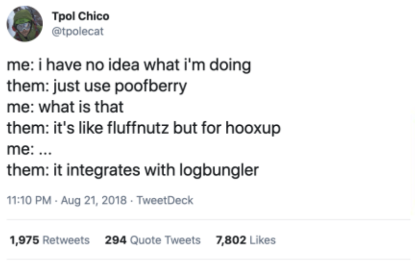





Han ma boo-kee, keelee ka-lya dooka. Wadja da boolya na 1.9 Megabytes
Bloated JabbaScript Frameworks
The Smuggler's Guide
O RLY?
@ThePracticalDev

"JavaScript fatigue"

> JavaScript fatigue: the inability to keep up with the latest tools, the fear of becoming obsolete, the constant change of the ecosystem, and the overwhelming choice.
>
> - Kim Maida/Auth0



Tpol Chico
@tpolecat

me: i have no idea what i'm doing
them: just use poofberry
me: what is that
them: it's like fluffnutz but for hooxup
me: ...
them: it integrates with logbungler

11:10 PM · Aug 21, 2018 · TweetDeck

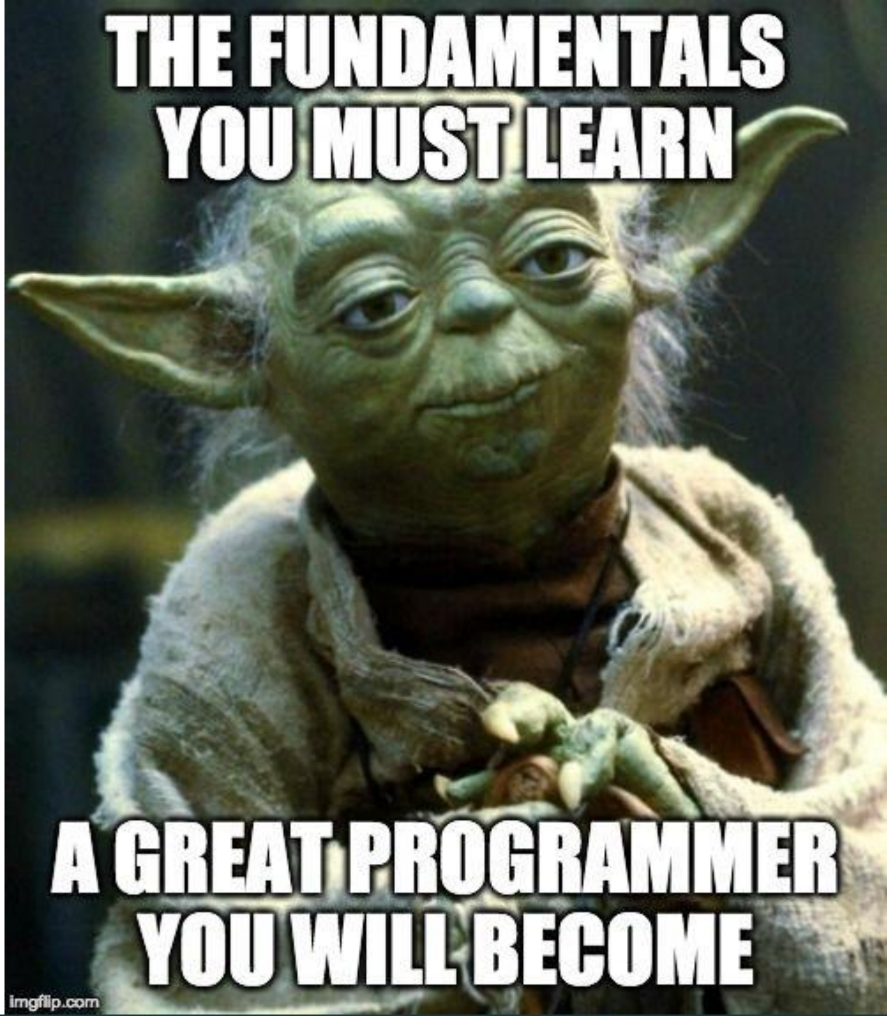
1,975 Retweets 294 Quote Tweets 7,802 Likes

It cannot be emphasized more: tools are just tools*.

What you really need is to understand the basics and the fundamentals. This course is all about that.

Yes, the knowledge of the details of any framework will be obsolete soon. Fundamentals will not.

*Going through the [React tutorial] won't make you a software engineer. Nevertheless, it is an essential step to understand the tool you use, and makes you more competent in that.


THE FUNDAMENTALS YOU MUST LEARN
A GREAT PROGRAMMER YOU WILL BECOME
imgflip.com

Step 3: Single Source of Truth

This one is tough.

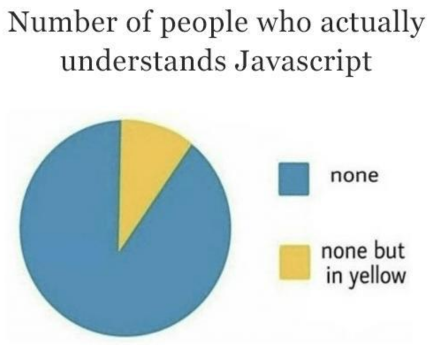
First: you definitely don't want to learn JS from the [Standard](). That is almost non-readable for human beings.

Second: the vast majority of the online resources are confusing, misleading or just simply wrong. Sometimes including the MDN.

Third: online courses. There are many. Simply too many. (also, goto the second point)

Fourth: the catch of 22, while you can find excellent resources, how do you know what is worthwhile?

So what exactly to do?



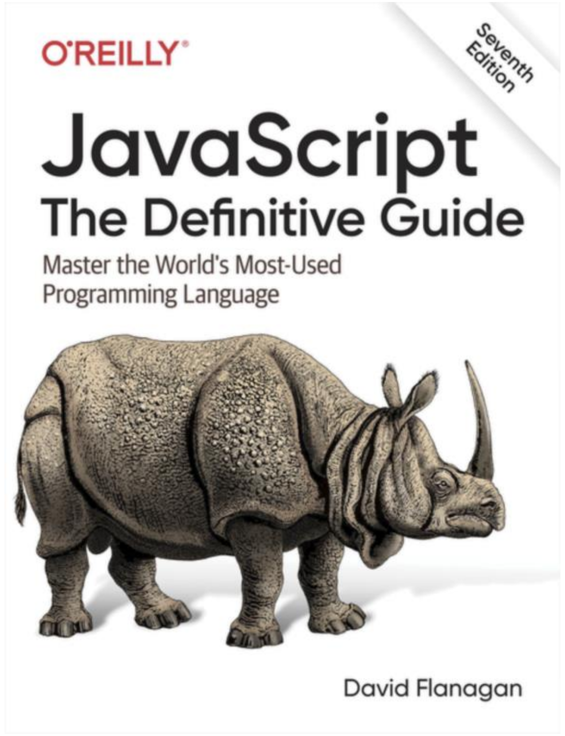
Step 3: the easiest way to learn JavaScript

The best solution is always to simplify the problem*. **

That being said, the simple solution sometimes seems too complex, exhausting and hard to achieve. Sometimes it does not only seems, it really is – but bypasses are always longer.

You may heard about this book – and how long it is.

It is not a light read, indeed. However, it is still much easier to read this, than being lost in the labyrinth forever.



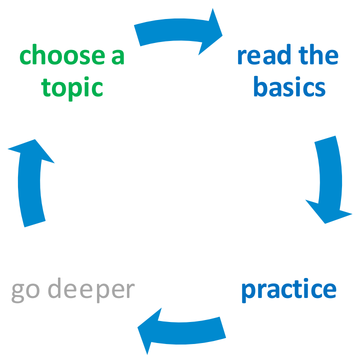
* ”Controlling complexity is the essence of computer programming.”
** KISS, YAGNI, DRY, SOLID

JavaScript: The Definitive Guide

So how to read it then?

"This book covers low-level fundamentals first, and then builds on those to more advanced and higher-level abstractions. The chapters are intended to be read more or less in order. But learning a new programming language is never a linear process, and describing a language is not linear either: each language feature is related to other features, and this book is full of cross-references sometimes backward and sometimes forward to related material."

"When learning a new programming language, it's important to try the examples in the book, then modify them and try them again to test your understanding of the language."



It builds up from basics and encourages to explore.

How to learn?

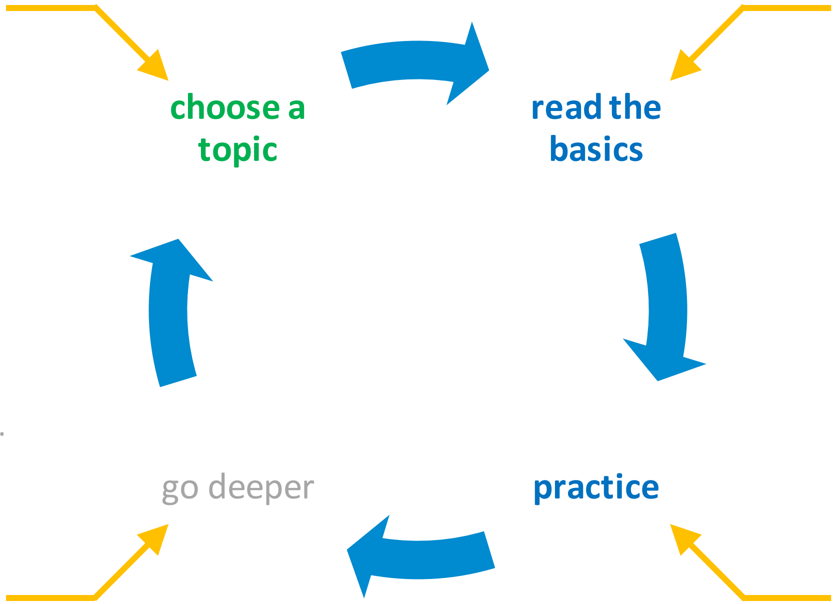
Start on the first page and read as long as you feel comfortable and easy to grasp what you read.

While the first chapters are essential, it is less useful to read the whole as a novel, or in one sit.

This will be the fun part – ***later***.

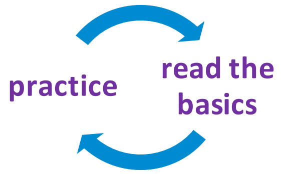
It could give you a lot of “aha” moments, and it will provide you the final cohesion force between the basics.

First, however, focus on the basics and practicing.



Give it a time, don’t hurry!

Don’t think that you *should* understand it quicky: it really requires more time to comprehend a topic deeply.



Start practicing almost immediately! Practically, do it in small iterations.

OBJECT

Object - creating with template literal

Object is the basic data structure in JavaScript.

- *An object is a set of properties (key-value entries).*
- *A property key can only be a string or a symbol.*

Ahh, symbol, not again… we will talk about this later, but it's really not that important. Trust me, I am an engineer.

```
const SOME_CONST = 'some const';
let babaYaga = {
    protagonist: 'John Wick',
    antagonist: 'ViggoTarasov',
    dog: 'Daisy',
    car: {
        year: '1969',
        type: 'Ford Mustang',
    },
    giveAGun: function () {
        return 'Heckler & Koch P30L'
    },
    'This Isn't Vengeance': function () {
        return 'This Is Justice';
    },
    [SOME_CONST]: 'some const'
}
```

an object literal

a property

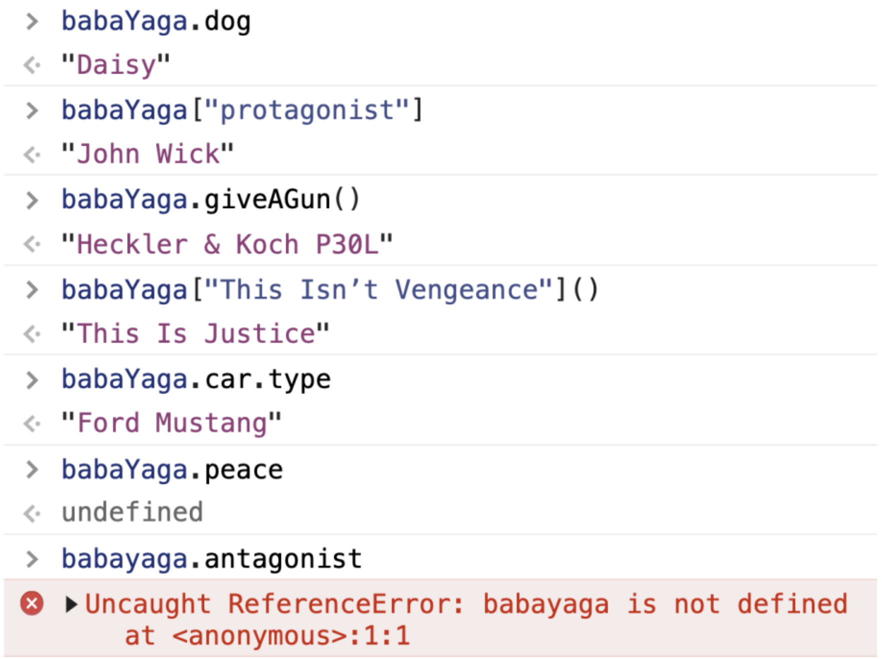
object in an object in an…

keys are really strings

function property: a method

JavaScript for impatient programmers: Object

Object - property access



```
> babaYaga.dog
<· "Daisy"
> babaYaga["protagonist"]
<· "John Wick"
> babaYaga.giveAGun()
<· "Heckler & Koch P30L"
> babaYaga["This Isn’t Vengeance"]()
<· "This Is Justice"
> babaYaga.car.type
<· "Ford Mustang"
> babaYaga.peace
<· undefined
> babayaga.antagonist
✖ ▶Uncaught ReferenceError: babayaga is not defined
     at <anonymous>:1:1
```

- dot notation
- bracket notation
- calling a method
- this is the fun part
- property of a property…
- nah, there is no such a thing
- trying to access a property of undefined

Object - creating with constructor

We can create objects with a constructor function as well:

```
let wickInstance = new WickConstructor();
```

the returned object — new — constructor function, this returns the object

don't worry, we will talk about prototypes and constructors later

We can call the constructor function without new as well - *but usually we should not*:

```
let wickSmtg = WickConstructor();
```

the returned *something* — constructor function, this returns *something*

Please note how consistent we are in naming: we expect *something*, therefore we name the variable accordingly.

Identifier names must be always descriptive

Sometimes, you simple cannot find a good name for a variable or a function. This is pretty normal in software engineering.

Usually, there are 2 root causes for that:

1. *you are a Java developer ;)*
2. *your code is wrong, and you cannot reason about*

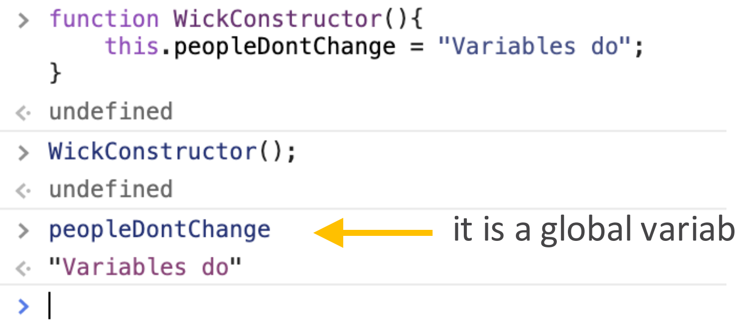
The solution is obviously not to find a name anyway, but to reconsider your solution.

Returns something

So we can call a constructor without new.
However, returning *something* is never a good sign.

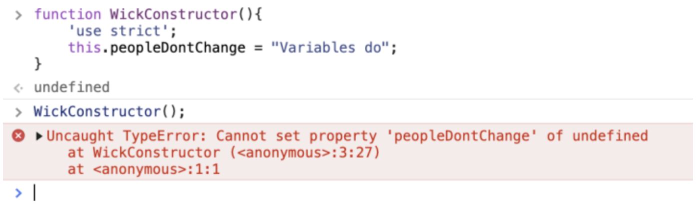
This way, we don't really know what will be returned (usually it is undefined).

To add insult to injury, we have another problem here, i.e. it could expose variables to the global scope!



```
> function WickConstructor(){
      this.peopleDontChange = "Variables do";
  }
< undefined
> WickConstructor();
< undefined
> peopleDontChange
< "Variables do"
> |
```

it is a global variable now



```
> function WickConstructor(){
      'use strict';
      this.peopleDontChange = "Variables do";
  }
< undefined
> WickConstructor();
✖ ▶Uncaught TypeError: Cannot set property 'peopleDontChange' of undefined
      at WickConstructor (<anonymous>:3:27)
      at <anonymous>:1:1
> |
```

strict mode FTW!

We will explain it in detail, later.

The key takeaway now, is that never call a constructor without *new* if it was not designed to be!

William-Thomas-Fredreich!?

Well, then... let it be.

But objects are a crucial part of JavaScript, so be prepared for complete lecture on this topic.

or more....

STRING

String

The String type is the set of all ordered sequences of zero or more 16-bit unsigned integer values (“elements”) up to a maximum length of 2^{53} - 1 elements. The String type is generally used to represent textual data in a running ECMAScript program, in which case each element in the String is treated as a UTF-16 code unit value.

Practically, string is a text, containing 16-bit Unicode characters.

String is data type and a data structure, as it is a collection of integers (interpreted as characters).

Just to confuse you, String also a built-in object in JavaScript, so we can introduce important concepts with it.

```
let a = "Ω"; // "\u03A9"
```

in decimal: 3*16*16 + 10*16 + 9 = 937

String

String

single quotes
double quotes
you can concatenate them

```
let antagonist = 'Viggo Tarasov';
let protagonist = "John Wick";
const saidViggoTarasov = "John" + " is a man of focus, commitment, sheer will";
```

an empty string is falsy
a non-empty string is truthy

```
Boolean("");                // false
Boolean(antagonist);        // true
```

template literals are
sometimes a bit of magic

```
let templateLiteral = `John Wick has a problem with ${antagonist}`;
```

you can mix them

```
let div = '<div class="angry">' + protagonist + '</div>';
```

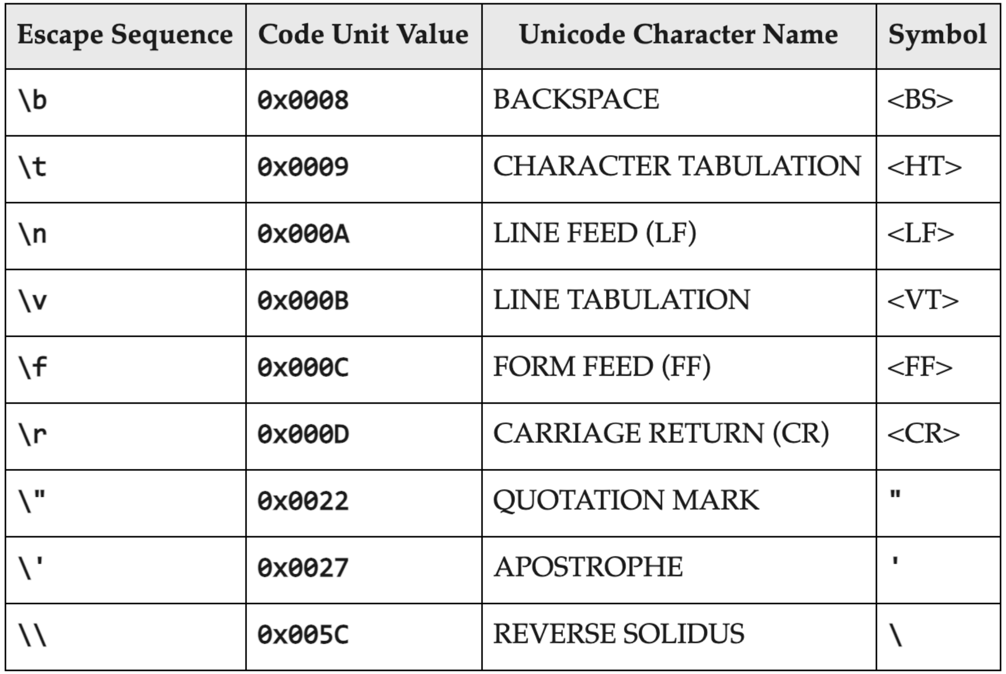
it does have a length

sometimes the length is
more than you'd expect

```
protagonist.length;         // 9
let gClef = "𝄞";            // let gClef = "\u{0001D11E}";
gClef.length;               // 2
```

String – escape Sequences

Table 34: String Single Character Escape Sequences



Escape Sequence	Code Unit Value	Unicode Character Name	Symbol
\b	0x0008	BACKSPACE	<BS>
\t	0x0009	CHARACTER TABULATION	<HT>
\n	0x000A	LINE FEED (LF)	<LF>
\v	0x000B	LINE TABULATION	<VT>
\f	0x000C	FORM FEED (FF)	<FF>
\r	0x000D	CARRIAGE RETURN (CR)	<CR>
\"	0x0022	QUOTATION MARK	"
\'	0x0027	APOSTROPHE	'
\\	0x005C	REVERSE SOLIDUS	\

There is a history behind these sequences. With these you can *escape* from the text and start doing something, usually controlling the output device.

these two, however, are really nasty and will cause you a lot of trouble.

String - accessing characters

To get a character at position *pos*, use square brackets `[pos]` or call the method `.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

str[0] = 'h';                 // doesn't work
alert( str[0] );              // H
```

there is a reason why it does not work, and probably you don’t want to know why…

Wrapper objects

… but let me tell you anyway.

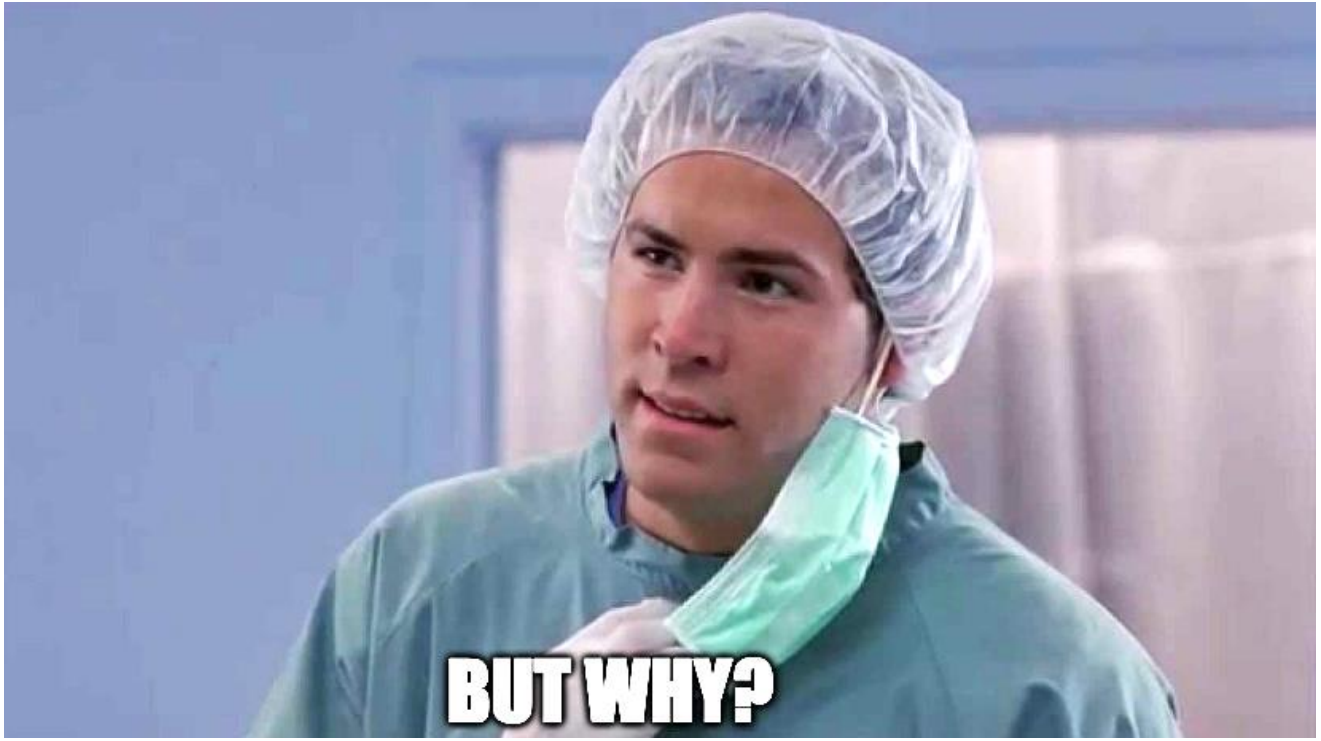
When the interpreter run into a *property accessor* (*dot notation*, or *bracket notation)* with a primitive value, such as a *string*, *number* and *boolean*, it will conveniently wrap you the primitive with a built-in object into a wrapper object, and you will operate on that object.

After the operation, the wrapper object will be destroyed.

property access

```
str[0] = 'h';    // doesn't work, but
                 // there is no error
```

In fact, this works (that's why there is no error), but not on the primitive, rather on the wrapper object, and that will be dropped afterwards.


BUT WHY?

… to be able to use all of the string prototype methods on a string primitive.

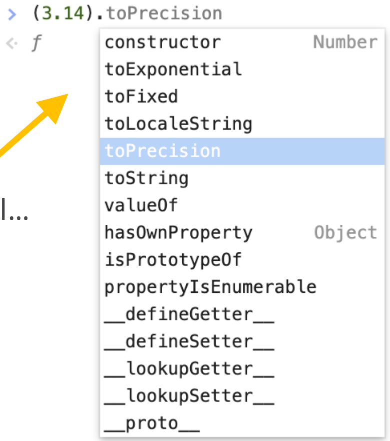
Method	Description
charAt()	Returns the character at the specified index.
charCodeAt()	Returns the Unicode of the character at the specified index.
concat()	Joins two or more strings and returns a new string.
endsWith()	Checks whether a string ends with a specified substring.
includes()	Checks whether a string contains the specified substring.
indexOf()	Returns the index of the first occurrence of the specified value in a string.
lastIndexOf()	Returns the index of the last occurrence of the specified value in a string.
localeCompare()	Compares two strings in the current locale.
match()	Matches a string against a regular expression and returns an array of all matches.
repeat()	Returns a new string which contains the specified number of copies of the original string.
replace()	Replaces the occurrences of a string or pattern inside a string with another string and return a new string without modifying the original string.
search()	Searches a string against a regular expression and returns the index of the first match.
slice()	Extracts a portion of a string and returns it as a new string.
split()	Splits a string into an array of substrings.
startsWith()	Checks whether a string begins with a specified substring.
substr()	Extracts the part of a string between the start index and a number of characters after it.
substring()	Extracts the part of a string between the start and end indexes.
toLocaleLowerCase()	Converts a string to lowercase letters, according to host machine's current locale.
toLocaleUpperCase()	Converts a string to uppercase letters, according to host machine's current locale.
toLowerCase()	Converts a string to lowercase letters.
toUpperCase()	Converts a string to uppercase letters.
trim()	Removes whitespace from both ends of a string.

Built-in objects – String, Number, Boolean

So you are saying that apart from the string, the number and boolean primitives also will be wrapped with their built-in objects?

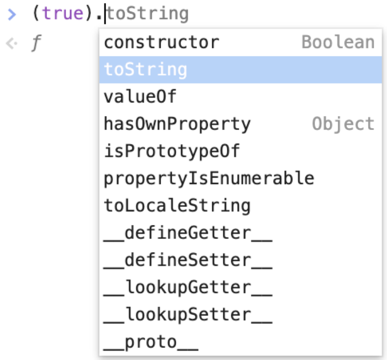
I am glad you listened. Yes – they will.

The easiest way to check:



```
> (3.14).toPrecision
< f   constructor          Number
      toExponential
      toFixed
      toLocaleString
      toPrecision
      toString
      valueOf
      hasOwnProperty       Object
      isPrototypeOf
      propertyIsEnumerable
      __defineGetter__
      __defineSetter__
      __lookupGetter__
      __lookupSetter__
      __proto__
```

these are useful…



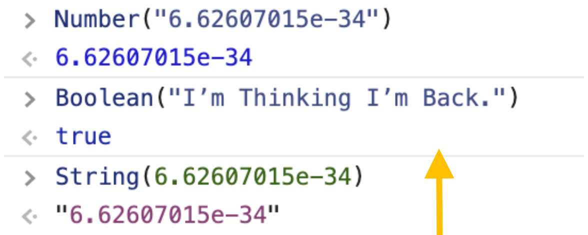
```
> (true).toString
< f   constructor          Boolean
      toString
      valueOf
      hasOwnProperty       Object
      isPrototypeOf
      propertyIsEnumerable
      toLocaleString
      __defineGetter__
      __defineSetter__
      __lookupGetter__
      __lookupSetter__
      __proto__
```

…these are not that much

Calling constructors without new

Yes, this is the special case when the constructor was designed that way, so we can call it without new as well. In fact, this is one of the main use cases of some of the built-in objects.

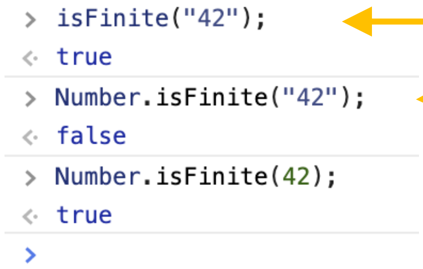
You can use them for explicit type conversions:



```
> Number("6.62607015e-34")
<· 6.62607015e-34
> Boolean("I'm Thinking I'm Back.")
<· true
> String(6.62607015e-34)
<· "6.62607015e-34"
```

this will be very useful combining with the .filter() method, as we'll see it then

Also, we have useful static methods on built-in objects:

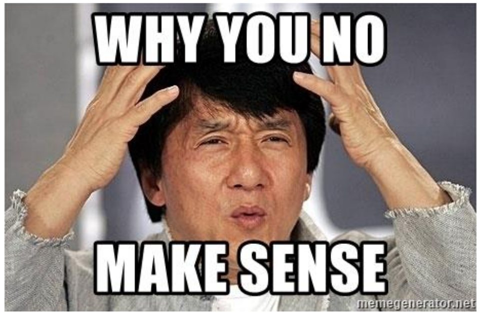


```
> isFinite("42");
<· true
> Number.isFinite("42");
<· false
> Number.isFinite(42);
<· true
>
```

type conversion: *oops, I did it again*

no conversion

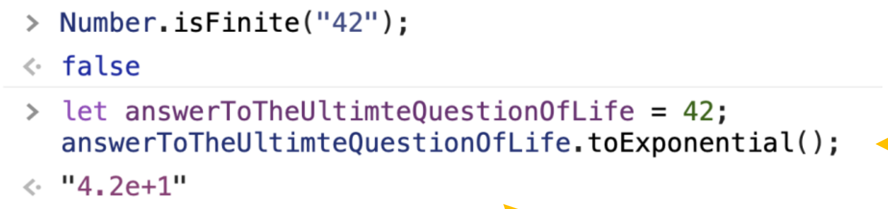
Prototype methods vs static methods



we'll explain later why we call it prototype and static - now focus on the differences in how to call them!

Without getting too deep into the topic, methods can…

… operate on the argument



```
> Number.isFinite("42");
< false
> let answerToTheUltimteQuestionOfLife = 42;
  answerToTheUltimteQuestionOfLife.toExponential();
< "4.2e+1"
```

… or operate on the object, itself

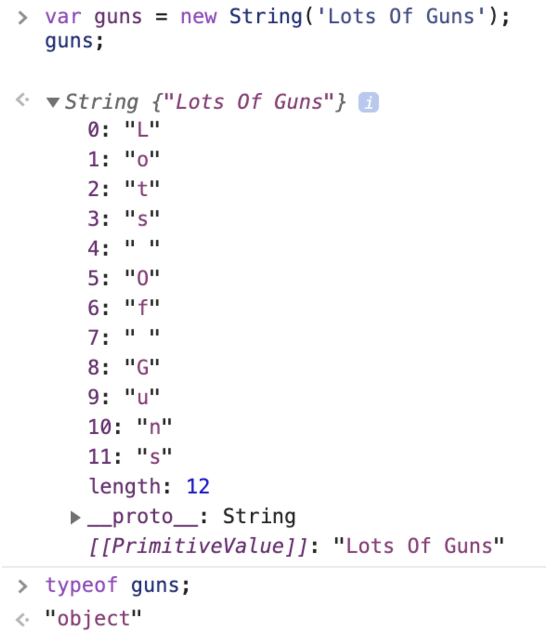
*…in long identifier names, it is always easy to spot typos. (or **not**)*

descriptive !== long

Build-in objects and new

So we can safely call the constructor of the string, number and boolean built-in objects without new. But can we call them with new?

Yes, you can, but there is little point in that.



```
> var guns = new String('Lots Of Guns');
  guns;

<· ▼String {"Lots Of Guns"} i
      0: "L"
      1: "o"
      2: "t"
      3: "s"
      4: " "
      5: "O"
      6: "f"
      7: " "
      8: "G"
      9: "u"
      10: "n"
      11: "s"
      length: 12
     ▶__proto__: String
      [[PrimitiveValue]]: "Lots Of Guns"
> typeof guns;
<· "object"
```

These objects and their primitives behave similarly most of the time, yet please don't declare these variables this way.

Searching for a substring

`.indexOf(searchValue [, fromIndex])` returns the position where the match was found or -1 if nothing can be found. Also, we have `.lastIndexOf()`.

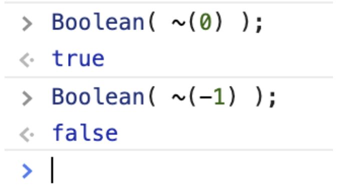
```
let str = '*** Widget ***';
if (~str.indexOf('Widget')) {
    alert('Found it!');
}
```

We just show you this, so that now you know, what not to use.

It is not explicit.

```
let str = '*** Widget ***';
if (str.indexOf('Widget') !== -1) {
    alert('Found it');
}
```

It is the *bitwise not* and it works this way:
it returns 0 only for -1



```
> Boolean( ~(0) );
<· true
> Boolean( ~(-1) );
<· false
> |
```

A jó vadász legjobb barátja a ~~sötétség~~ az explicit kód.

The good hunter's best friend is the ~~darkness~~ explicit code.

Your code always have to be expressive

A bitwise operator is explicit only in one case:
when the goal it do a bitwise operation on a value.

It tells me: *“I want a bitwise operation on a value,
because I am interested about bits.”*

But what it really is: ***checking an occurrence in a string***.

It may be a surprise but using the longer version
is less complex, than the shorter.

Checking a substring existence: includes, startsWith, endsWith

`.includes(searchString[, position])`

The method determines whether one string may be found within another string, returning true or false as appropriate.

```
var str = 'To be, or not to be, that is the question.';
str.includes('To be');                  // true
str.includes('question');               // true
str.includes('nonexistent');            // false
str.includes('To be', 1);               // false
```

The methods `.startsWith()` and `.endsWith()` do exactly what they say:

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

Getting a substring

MDN is using the term: confusing 590 times. This is one of them.

method	selects...	negatives
slice(start, end)	from start to 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

Getting a substring - continued

```
//*** length vs indices:
"string".substring(2, 4);     // "ri" (start, end) indices / second value is NOT inclusive
"string".substr(2, 4);        // "ring" (start, length) length is the maximum length to return
"string".slice(2, 4);         // "ri" (start, end) indices / second value is NOT inclusive

//*** watch out for substring swap:
"string".substring(3, 2);     // "r" (swaps the larger and the smaller number)
"string".substr(3, 2);        // "in"
"string".slice(3, 2);         // "" (just returns "")

//*** negative second argument:
"string".substring(2, -4);    // "st" (converts negative numbers to 0, then swaps first and second position)
"string".substr(2, -4);       // ""
"string".slice(2, -4);        // ""

//*** negative first argument:
"string".substring(-3);       // "string"
"string".substr(-3);          // "ing" (read from end of string)
"string".slice(-3);           // "ing"
```

Brought to you by Stack Overflow

Correct comparisons

The `LocaleCompare()` method returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.

```
referenceStr.localeCompare(compareString[, locales[, options]])
```

```
// The letter "a" is before "c" => negative value
'a'.localeCompare('c');                               // -2 or -1 (or some other negative value)

// Alphabetically the word "check" comes after "against" => positive value
'check'.localeCompare('against');                     // 2 or 1 (or some other positive value)
'a'.localeCompare('a');                               // 0: "a" and "a" are equivalent

let items = ['réservé', 'Premier', 'Cliché', 'communiqué', 'café', 'Adieu'];
items.sort((a, b) => a.localeCompare(b, 'fr', { ignorePunctuation: true }));
// ['Adieu', 'café', 'Cliché', 'communiqué', 'Premier', 'réservé'

console.log('ä'.localeCompare('z', 'de'));       // a negative value: in German, ä sorts before z
console.log('ä'.localeCompare('z', 'sv'));       // a positive value: in Swedish, ä sorts after z

console.log("2".localeCompare("10"));                                // 1: by default, "2" > "10"
console.log("2".localeCompare("10", undefined, { numeric: true }));   // -1: numeric using options
```

MDN: .localCompare()

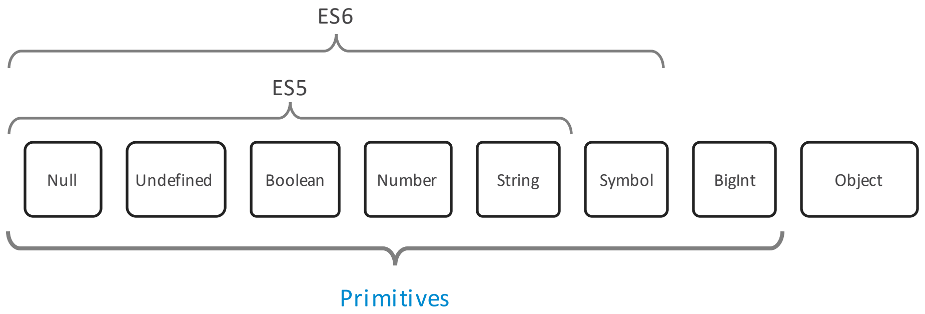
Summary

- Strings in JavaScript are encoded using UTF-16.
- 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.

ARRAY

In the last episode …

In the *JavaScript Intro* lecture we already introduced the data types:



"Where are thou, Array?"

We also mentioned, that Function is not a JavaScript type, right? Function
Let us add Array to this list: Array so Array is not a type, too.

Then what is it?

Array is an object.

Data Types vs Structures

Array is an object.

Why is it important?

Because in almost every aspect it looks and behaves as an object.
It is much easier to think and reason about arrays as objects.

Also, you could avoid mysterious bugs, if you think about it.

⁂

That being said, many articles, blog posts, even books refers it as *array type.*

Why?

Because they refer “type” as a data structure (such as a collection of data).
In this series, we try to keep the terminology unambiguous. Array is an object, and that’s it.

“God, observing their city and tower, confounds their speech so that they can no longer understand each other, and scatters them around the world.”

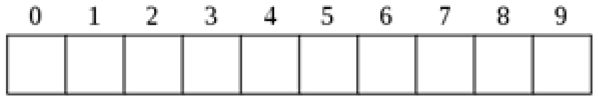
↑

also, it is an array of stories. kind of.

Array

Array is an ordered collection of indexed elements of any type.

The first element's index is zero.



```
let arr = [];
let arr = [1, true, ..., "Hello"];
```

You'll almost always initialize an array as an array literal...

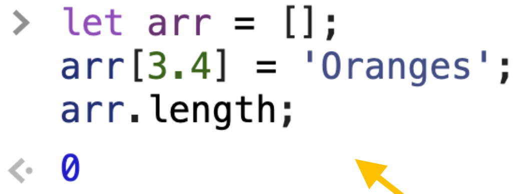
```
let arr = Array.from();
```

... or from an array-like or iterable object.

```
let arr = Array.of;
let arr = Array();
let arr = Array(42);
let arr = Array(1, true, ..., "Hello");
let arr = new Array();
let arr = new Array(arrayLength);
let arr = new Array(1, true, ..., "Hello");

arr.length = arrayLength;
```

How *not to add* an element to an array



```
> let arr = [];
  arr[3.4] = 'Oranges';
  arr.length;
<· 0
```

This is not what it looks like.
It adds a simple property to the array object.

“Is that noooormal?”

Array - accessing elements

Array length

```
const fruits = ['Apple', 'Banana'];
console.log(fruits.length); // 2
```

Accessing array elements:

```
const first = fruits[0]; // Apple
const last = fruits[fruits.length - 1]; // Banana
```

.length

`[].length`

The length property sets or returns the number of elements in that array.

```
var cats = ['Dusty', 'Misty', 'Twiggy'];
console.log(cats.length);
cats.length = 2;
console.log(cats);
cats.length = 0;
console.log(cats);
cats.length = 3;
console.log(cats);
```

Array.isArray

now we are familiar with this construct: it operates on the argument

`Array.isArray()`

A static method determines whether the passed value is an Array.

~~`[].isArray()`~~

```
Array.isArray([
    1,
    2,
    3,
    'John'
]); // true

Array.isArray({
    prop: 10
}); // false
```

basically, this is the only proper way to check an array

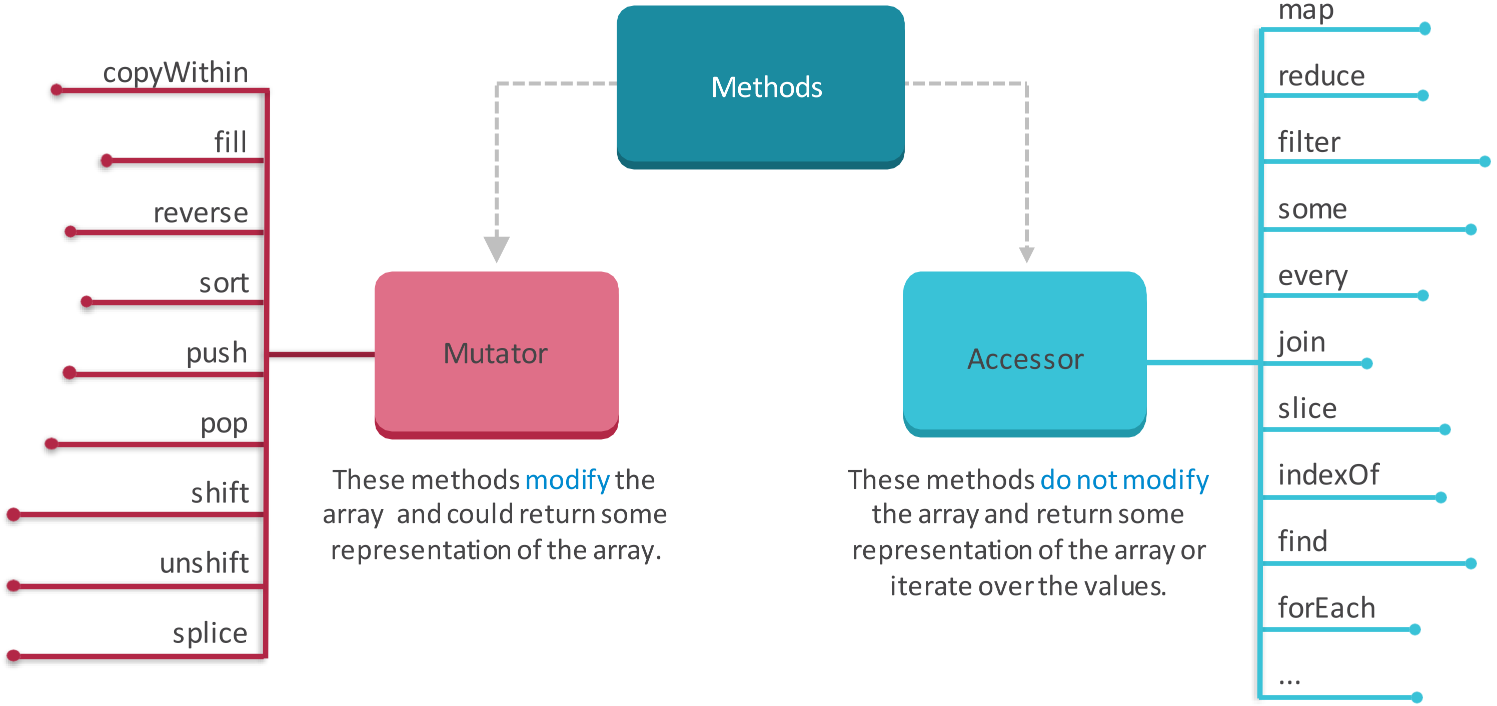
alternatively - if you like it:

```
Object.prototype.toString.call([]) === '[object Array]'
```

other solutions could lead to false answers in some cases, such as:

```
[] instanceof Array;
```

Array prototype methods



These methods modify the array and could return some representation of the array.

These methods do not modify the array and return some representation of the array or iterate over the values.

.fill

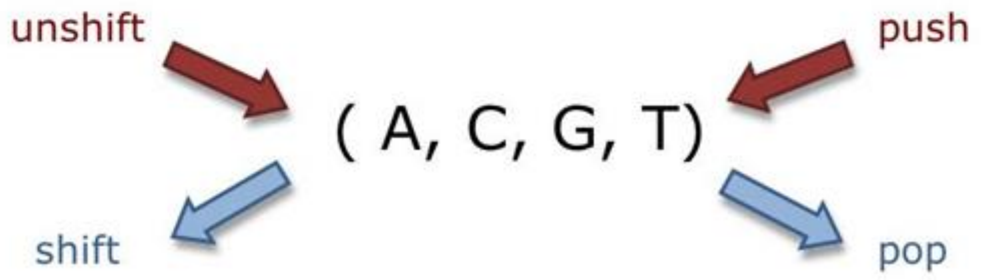
```
.fill(value)
```

The fill() method fills all the elements of an array from a start index to an end index with a static value.

```
arr.fill(value[, start = 0[, end = this.length]])
```

```
[1, 2, 3].fill(4);
[1, 2, 3].fill(4, 1);
[1, 2, 3].fill(4, 1, 2);
```

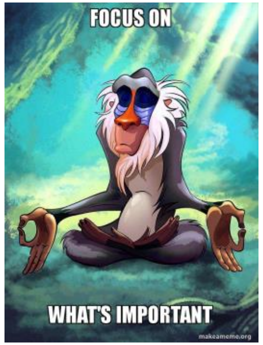
pop, push, shift, unshift



`.pop`	extracts the last element of the array and returns it;
`.push`	appends the element to the end of the array;
`.shift`	extracts the first element of the array and returns it;
`.unshift`	adds the element to the beginning of the array;

← *push/pop methods run fast, while shift/unshift are slow.*

.pop, .push



these are!

.pop

Removes the last element from an array and returns that element. This method changes the length of the array.

```
let arr = ['one', 'two', 'three'];
arr.pop(); // 'three'
console.log(arr);
// ['one', 'two']
```

.push

Adds one or more elements to the end of an array and returns the new length of the array.

```
let arr = ['one', 'two'];
arr.push('three’);
console.log(arr);
// ['one', 'two', 'three']
```

return value is arr.length!

just sayin’

.splice, .fill

.splice

Changes the contents of an array by removing existing elements and/or adding new elements.

```
let arr = [
    'one',
    'two',
    'bad_value',
    'four'
];
arr.splice(2, 1, 'three');
console.log(arr);
// ['one', 'two', 'three', 'four']
```

.fill

Fills all the elements of an array from a start index to an end index with a static value.

```
let arr = ['c', 'b', 'a'];
arr.fill('same');
console.log(arr);
// ['same', 'same', 'same']
```

.find

.find (ES6)

Method returns the value of the first element in the array that satisfies the provided testing function. Otherwise, undefined is returned.

```
let firstDivisibleNumBy14 = [1, 2, 28].find(
    (value) => {
        return value % 14 === 0;
    }
); // 28
```

Loops

for

```
let arr = ['Apple', 'Orange', 'Pear'];
for (let i = 0; i < arr.length; i++) {
    alert(arr[i]);
}
```

for...of

```
for (let fruit of ['Apple', 'Orange', 'Plum']) {
  alert(fruit);
}
```

for...in

```
let arr = ['Apple', 'Orange', 'Pear'];
for (let key in arr) {
    alert(arr[key]);
}
```

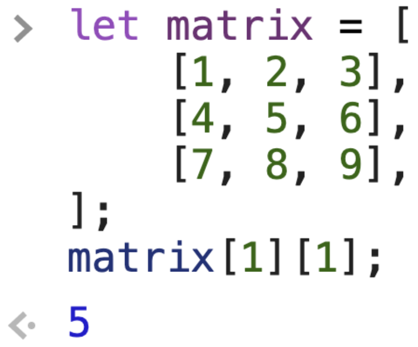
Cons

The loop for ... in iterates over all properties, not only the numeric ones.

The for ... in loop is optimized for generic objects, not arrays, and thus is 10-100 times slower.

Multidimensional arrays

Arrays can have items that are also arrays.
We can use it for multidimensional arrays,
to store matrices.



```
> let matrix = [
      [1, 2, 3],
      [4, 5, 6],
      [7, 8, 9],
  ];
  matrix[1][1];
<· 5
```

Summary

The length property is the array length or, to be precise, its last numeric index plus one. It is auto-adjusted by array methods.

`push(...items)`	adds items to the end;
`pop()`	removes the element from the end and returns it;
`shift()`	removes the element from the beginning and returns it;
`unshift(...items)`	adds items to the beginning;

To loop over the elements of the array:

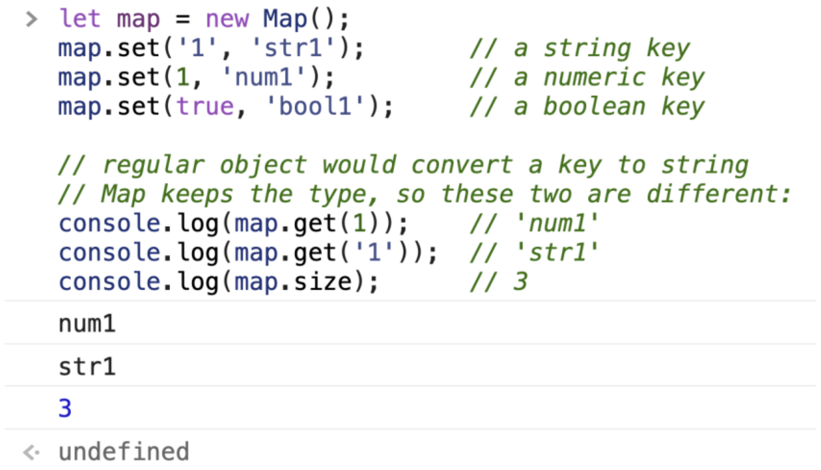
`for (let i=0; i<arr.length; i++)`	works fastest, old-browser-compatible;
`for (let item of arr)`	the modern syntax for items only;
`for (let i in arr)`	never use;

MAP AND SET

Map

The Map object holds key-value pairs. Any value (both objects and primitive values) may be used as either a key or a value.

The iteration goes in the same order as the values were inserted. Map preserves this order, unlike a regular Object.



```
> let map = new Map();
  map.set('1', 'str1');         // a string key
  map.set(1, 'num1');           // a numeric key
  map.set(true, 'bool1');       // a boolean key

  // regular object would convert a key to string
  // Map keeps the type, so these two are different:
  console.log(map.get(1));      // 'num1'
  console.log(map.get('1'));    // 'str1'
  console.log(map.size);        // 3
  num1
  str1
  3
<· undefined
```

Map methods and properties

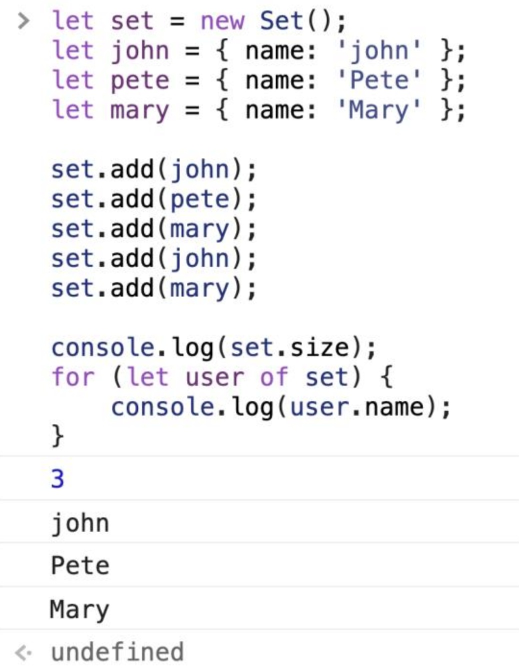
`new Map()`	creates the map;
`.set(key, value)`	stores the value by the key;
`.get(key)`	returns the value by the key, undefined if key doesn’t exist in map;
`.has(key)`	returns true if the key exists, false otherwise;
`.delete(key)`	removes the value by the key;
`.clear()`	removes everything from the map;
`.size`	returns the current element count;
`.keys()`	returns an iterable for keys;
`.values()`	returns an iterable for values;
`.entries ()`	returns an iterable for entries [key, value], it’s used by default in for … of;

Set

Set objects are collections of values.

A set object lets you store unique values of any type (whether primitive values or object references).

You can iterate through the elements of a *set* in insertion order. A value in the *set* may only occur once; it is unique in the collection.



```
> let set = new Set();
  let john = { name: 'john' };
  let pete = { name: 'Pete' };
  let mary = { name: 'Mary' };

  set.add(john);
  set.add(pete);
  set.add(mary);
  set.add(john);
  set.add(mary);

  console.log(set.size);
  for (let user of set) {
      console.log(user.name);
  }
  3
  john
  Pete
  Mary
<· undefined
```

Set methods and properties

`new Set(iterable)`	creates the set, and if an iterable object is provided (usually an array), copies values from it into the set;
`.add(value)`	adds a value, returns the set itself;
`.delete(value)`	removes the value, returns true if value existed at the moment of the call, otherwise false;
`.has(value)`	returns true if the value exists in the set, otherwise false;
`.clear()`	removes everything from the set;
`.size`	is the elements count;
`.keys()`	returns an iterable object for values
`.values()`	same as set.keys(), for compatibility with Map
`.entries()`	returns an iterable object for entries [value, value], exists for compatibility with Map

Q&A

edu_hu@epam.com