SUBJECT. You don't Know JS Yet (Get started) YEAR. MONTH. DAY.

* preface: is written by the author and tells readers how and why the book come into bung.
* foreward: is written by someone other than the author and tells the readers why they should read the book.
* You don't know JS Yet (YDKJSY)
this book is written for anyone who has at least 6-12 months experience.
* The parts: Every parts of JS are useful, some parts are more useful than others. You should go about learning all parts of javascript and where appropriate use them. Don't buy the lie that you should only learn and use a small collection of good parts while avoiding all the bad stuff. Don't listen when sb says your code isn't "modern".
* The mission with YDKJSY is to empower every single JS developer to fully own the code they write, to understand it and to write with intention and clarity.
* The path: Take one chapter, read it completely through start to finish, and then go back and re-read it section by section. Stop between each section and practice the code or ideas from that section. Spend a week or two for each chapter and a month or two on each book.
* chapter1: what is javascript?
* patience and persistance are best as you take these first few steps
* three main pillars in JS: 1) Scope/closure
2) prototype/object
3) Types/coercion

NOTEBOOK

SUBJECT.

YEAR. MONTH. DAY.

* A good start always depends on a solid first step.
* The name Javascript is an artifact of marketing shenanigans. Brendan Eich, he code-named it Mocha.
Internally at Netscape, called LiveScript.
When to publicity, Javascript was the vote.
* There are some superficial resemblances between Javascript and Java code. both Java and Javascript syntax from C (and to an extent, C++)
* Oracle (via Sun) the company that still owns and runs Java, also owns the official trademark for the name "Javascript" (via Netscape). This trademark is almost never enforced, and likely couldn't be at this point.
* For these reason some have suggested use JS instead.
The offical name by TC39 and formalized by the ECMA standards body is ECMAScript. since 2016 add suffix to that. e.g ECMAScript2019 or ES2019
* Don't use terms JS5 or ES8 to refer to the language. some do but those terms only serve to perpetuate confusion. "ES20XX" or just JS are what you should stick to.
* Java is to Javascript as ham is to hamster → Jeremy Keith 2009
* TC39 the technical steering committee that manages JS. They meet regularly to vote on any agreed changes.

NOTEBOOK

SUBJECT.

YEAR. MONTH. DAY.

* TC39 members between 30 to 100, from broad section of web industrial companies (Mozilla, Google, Apple) device maker (Samsung)
* There is just one JS, the official standard as maintained by TC39 and ECMA
* Appendix B, An additional ECMAScript Features for web browsers. The JS specification includes this Appendix to detail out any known mismatches between official JS specification and the reality of JS on the web. In other word there are exceptions that allowed only for web JS. other JS environment must stick to the letter of the law.
* Various JS environment (like browser JS engines, NodeJS, etc) add APIs into Global scope of your JS programs that give you environment-specific capabilities, like being able to pop an alert-style box in the user's browser.
fetch(), getCurrentLocation(), fs.writeFile(), console.log(), alert()
they are APIs → They are not defined by JS. But they look like JS. they are functions and object methods and they obey JS syntax rules. The behavior behind them are controlled by the environment running the JS engine, but on the surface they officially have to abide by JS to be able to play in the JS playground.
So an Alert() call is JS, but alert itself is really just a guest, not part of the official JS specification.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* Dev tools are tools for developers. Their primary purpose is to make life easier for developers. They prioritize DX (Developer experience). Since such tools vary in behaviour from browser to browser.

Don't trust what behaviour you see in a dev tools console as representing exactly the latter JS semantics. For that read the specification. Instead, think of the console as a "JS-friendly" environment that's useful in its own right.

* Paradigm: mindset and approach to structure code.

3 typical paradigm code strategies:

1) Procedural: organized code in a top-down, linear progression through a pre-determined set of operations, usually collected together in related units called procedures.

2) OO (object oriented): style code by collecting logic and data together into units called classes.

3) FP (functional): organized code into functions (pure computation as opposed to procedures), and the adaptation of those functions as value.

C → procedural
Java and C++ → OOP
Haskel → FP
} single paradigm language

JS → multi paradigm language → we can use all paradigm in a single application.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* Forward and Backward compatibility.

Backward: the ability of a new version of JS to understand and work with code written in an older version.

Forward: the ability of older versions of JS to understand and work with code written in a newer version.

JS isn't, and can't be, forwards-compatible, it's critical to recognize JS is backwards compatibility.

HTML and CSS are forward compatible but to some certain extent are also backward compatible ???

* Since JS is not forward-compatible, it means that there is always a gap between your code and oldest engine.

The solution is transpiling like using babel.

it is very important to our code is used new version of JS because of clean, and communicates ideas most effectively. cause of that we have to use transpiler.

* If the forward-compatibility issue is not related to new syntax, but rather to a missing API method that was only recently added, the most common solution is to provide a definition for that missing API. This pattern called a polyfill (aka "shim").

→ Transpilation and polyfilling are two highly effective techniques for addressing the Gap between code and environment.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

e.g.

```
if(str){                                  var x$0, x$1;
    int x = 3          transpile          if(str){
    clg(x)            ---------->           x$0 = 3
}                        babel              clg(x$0)
else {                                    } else { x$1 = 4
    int x = 4                               clg(x$1)
    clg(x)                                }
}
```

Promise, fetch —polyfill / "shim"→ کدهای جدید رفتارهای / اینه سیستم‌های قبلی ?

* compiling: The process of translating human-readable source code into machine-executable instructions. The compiled program can then be run on a computer without the need for a compiler to be present.

* interpretation: The process of executing code directly without any compilation step.

- Interpreted languages → python, Ruby, JS
- Compiled languages → C, C++, Java, Go

* The compiled code is specific to the target hardware, it typically can not be easily ported to other platforms. so if we want to run on another platform we need to re-compile its source code.

Compiled		Interpreted	
pro	Cons	pros	Cons
ready to run	not cross platform	cross-platform	interpreter required
often faster	inflexible	simpler to test	often slower
source code is private	extra step	easier to debug	source code is public

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* programming and scripting language.

Basically all scripting languages are programming languages.

programming → is an organized way of communication with computer using a set of commands and instruction. the term programming languages usually refers to high-level languages such as C++, Java, Ada, Pascal, Fortran.

scripting: supports script and is capable of being executed without being compiled ahead of time.

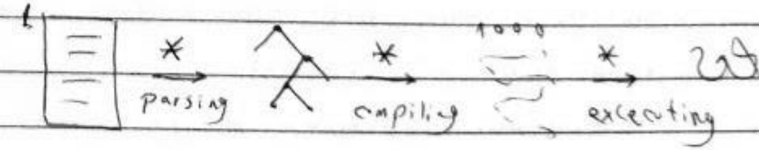
the process of analyzing a string of code or text to determine its structure and meaning. ↑	programming languages: C++ C# Java script languages: php python JS

* parsing: perform code generation before execution.

once any source code has been fully parsed, it is very common that its execution will include parsed form of the program usually called an Abstract Syntax Tree (AST) to that executable form → JS is parsed language.

JSVM → JS virtual machine.

post parsing → pass of JIT (Just-In-Time)



NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

1) After a program leaves a developer's editor, it gets transpiled by Babel, then packed by webpack or others then it gets delivered in that very different form to a JS engine.

2) JS engine parsed the code to an AST.

3) Then the engine converts that AST to kind of byte code, a binary intermediate representation (IR) which is then refined/converted even further by the optimizing JIT compiler.

4) Finally, the JS VM execute the program.

* It is clear that in spirit, if not in practice, JS is a compiled language.

*Polyfill and shim are often used interchangeably, but there is a subtle(-ish) difference between them.

Polyfill is specially designed to fill gap where a new functionality is unavailable in the browsers, whereas a shim is more general and can be used to modify and extend an existing API.

*Web Assembly (WASM)

before WASM at first in 2013 engineers from Mozilla 3 game from C to JS → it name is ASM.JS

Several years after ASM.js, another group of engineers (also initially from Mozilla) released Web Assembly (WASM)

WASM and ASM.js both provide a path for non-JS programs

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

(C, etc) to be converted to a form that could run in the JS engine WASM format similar to Assembly, that can skip the parsing/compilation that the JS engine normally does.

The parsing/compilation of a WASM-targeted program happen ahead of time (AOT), what's distributed is a binary packed program ready for the JS engine to execute with very minimal processing.

advantages: ① performance improvement ② bring more parity for non-JS language to web platform.

WASM is evolving (develop gradually) to become a cross-platform virtual machine (VM) of sorts, where programs can be compiled once and run in a variety of different system environment.

So WASM is not only for the web and is not JS

WASM will not replace JS. Some people will use it as an escape hatch ("") from having to write JS.

* Strictly speaking

In 2009, JS added strict mode as an opt-in mechanism for encouraging better JS programs.

* The benefits of strict mode far outweigh the costs, but sadly after 10 years, strict mode / so many useful things optionality means that it's still not necessarily the default for JS programmers because old habits die hard.

restrict mode is a guide to the best way to do things

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* use strict mode. Don't fight it and arguing with it.
* ES6 assumes strict mode, so all code in such files modules / assume strict mode Automatically.

* Defined (Summary)

1) JS is an implementation of the ECMAScript standard (version ES2019 as of this writing), which is guided by the TC39 committee and hosted by ECMA. It runs in browser and other JS environment such as NodeJS.

2) JS is a multi-paradigm language, meaning the syntax and capabilities allow a developer to mix and match (and bend and reshape) concepts from various major paradigms, such as procedural, Object-oriented (OO/classes) and functional (FP)

3) JS is a compiled language, meaning the tools (including the JS engine) process and verify a program (reporting any errors!) before it executes.

* Difference between functional and procedural programming paradigm

without side Effect	with side Effect
e.g: function addTwo(x) { return x + 2 }	e.g: let counter = 0 function increment() { return ++counter }
clg(addTwo(2)) // 4	clg(increment()) clg(counter) // 1
focus on "what to do" not "How to do"	designed to perform specific task emphasizes special linear approach.
writing readable, reuseable and modular code that easy to understand, test, debug.	execute commands line by line.

SUBJECT. YEAR. MONTH. DAY.

Procedural side effects can cause errors and make debugging more difficult. In functional, side effect are minimized through the use of pure functions that don't modify external state without side Effect.

* chapter 2 : surveying JS

The best way to learn JS is to start writing JS.

our goal is to get a better Feel For JS.

Each file is a program, each standalone file is its own separate program.

The reason this matters is mainly (primarily) around error handling. one file may fail (during parse/compile or execution) and that will not necessarily prevent the next file from being processed.

Consider separate .js files as separate JS programs From the perspective of usage of an application, it sure seems like one big program.

Many projects use build process tools to combine all separate files into a single file. when this happen, JS treats this seperate file as the entire program.

Regardless of which code organization pattern (and loading mechanism) is used for a file (standalone or module) you should still think of each files as its own (mini) program, which may then cooperate with other mini programs to perform the functions of your overall application.

global scope: use global (node) or use window (browser)

import-export file-based

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* values → values are data. → (1) primitive (2) object

values are embedded in programs using literal: [illegible]

In js, a literal refers to a notation for representing a fixed value in source code. In other words, it's a way of writing a constant value directly in code without storing in variable or calculating it on the fly.

[illegible]

e.g: greeting("my name is masoud") → literal - directly add value in program

literal make code more concise and easier to read.

keep in mind that literals are fixed values and can't changed at runtime.

- primitive string literal → "x" - 'x' `x`
  - "x" ↓ double quote, 'x' ↓ single quote → use them most of the time (stick to one)
  - `x` ↓ back-tick → use it in include interpolated expressions

clg(`my name is ${firstName}`) ← interpolation

- primitive boolean literal → true - false
- bigInt primitive type → const e = 1234 ...n (important → show bigInt)
- number primitive → [illegible] math [illegible]

* JS array indicates are 0-based (0 is the first position)

- null and undefined primitive → show emptiness or absence of a value. They are diffent (some differences historic and some of them contemporary)

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

It's safest and best to use only undefined as the single empty value.

- symbol primitive value → hidden unguessable value. symbols are almost exclusively used as special keys on object

hichhikersguide[symbol("meaning of life")] // 42

we won't encounter direct usage of symbols very often in typical JS programs. They're mostly used in low-level code such as in libraries and frameworks.

* Arrays and objects

arrays are a special type of object.

function like array are special kind (aka sub-type) of object

objects are more General.

object → key / properties → name.first or name["first"] (two forms to access values; square bracket)

array → numeric position.

key or properties and its values.

* typeof operator

typeof 42	// number	
" "abc"	// string	
" true	// boolean	
" undefined	// undefined	
" null	// object -- oop bug	exception
" {"a": 1}	// object	
" [1,2,3]	// object	exception
" function hello(){}	// function	exception

warning

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* variable → containers of values.
variable and literal
&
identifier.
* declared variable (create variable)
* several various syntax forms that declare variable (aka identifier)
e.g.
→ var - let - const → identifier
function scope — block scope — we cannot re-assigned
* It's ill-advised to use const with object values. because we can change values but we can't re-assigned it. avoid this
e.g. const actors = ["a", "b", "c"] ← situation
actors[2] = "d" // ok. :(
actors = [] // Error!
* The best semantic use of const is when you have a simple primitive value. This makes programs easier to read.
↳ if we do that, no confusion of re-assignment (not allowed) and mutation (allowed); that's the safest and best way to use const.
* other identifier
① function hello(name) {
clg(name)
}
identifier behaves like var
② try {
someError()
}
catch(err) {
clg(err)
}
identifier
behaves like let

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* function declaration * function expression
function hello() {
return
}
(parameters)
const, var or let = myFunction() {
return
}
arrow function
* functions are values. can passed, assigned.
* not all language treats functions as values. but JS does.
* only return single value. if we have more value we can return array and object
* function can be assigned as properties of an objects:
var whatToSay = {
greeting() {...}
question() {...}
answer() {...}
}
whatToSay.greeting()
// Hello
statement / expression + expression e.g. const x = [1,2]
x[5] = 5
clg(x[3]) // undefined
clg(x) // [1,2,0,0,0,5]
0 1 2 3 4 5
undefined
* Comparison → equality
we must be aware of the nuanced differences between an equality comparison and an equivalence comparison
== ===
check value check behaviour or function

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* Triple equals === or strict equality

3 === 3.0 // true
"Jss" === "Jss" // true
null === null // true
false === false // true

42 === "42" // false
"hello" === "Hello" // false
True === 1 // false
0 === null // false
"" === null // false
null === undefined // false

== → loose equality

* All value comparisons in JS consider the type of the values being compared, not just === operator. Specifally, === disallows any sort of type conversion (aka, "coercion") in its comparison, where other JS comparisons do allow coercion. كذا بزن

Coercion
- Implicit → clg("1" + 2); // "12"
  clg(1 == "1") // True
- Explicit → clg(Number("42"); // 42

The process of converting values

The === operator is designed to fix in two cases

① NaN === NaN // false
② 0 === -0 // True

} avoid using === in these two situations

NaN → is a special value that represents an undefined or unrepresentable value resulting from a mathematical operation. It indicates that a value is not a legal number.

① 5/0 → Infinity -5/0 → -Infinity
② 0/0 → NaN → بزن

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

③ clg(Math.sqrt(-1) → NaN
clg(Math.log(-1) → NaN

④ clg(Number("Hello") → NaN
Number("") → NaN
Number(true) → 1
Number(false) → 0

⑤ NaN + 5 → NaN
NaN * 5 → NaN

⑥ isNaN("Hello") → True
isNaN(5) → False

clg(Object.is('1', 1)) // false
clg(Object.is(NaN, NaN)) // True
clg(Object.is(-0, 0)) // false
const obj = {} // false
clg(Object.is(obj, {}))

* If we want not to get lie for { NaN → use Number.isNaN(...)
-0.0 → use Object.is(...)

You could think of Object.is() as the "quadruple-equals" ====, the really-really-strict comparison.

* The story gets even more complicated when use (non-primitive)

[1,2] === [1,2] // false
{a:42} === {a:42} // false
(x => x*2) === (x => x*2) // false
→ refer to structral equality.

* In JS all object values held by reference are assigned and passed by reference-copy, and for our current discussion are compared by reference (identity) equality:

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

var x = [1,2,3]

* assignment is by reference copy. So y references the same array as x, not another copy of it.

var y = x;

y === x // True → both hold a reference to the same array

y === [1,2,3] // false } compare only reference identity.

x === [1,2,3] // false }

* JS does not provide a mechanism for structural equality comparison of object values, only reference identity comparison. To do structural equality comparison, you'll need to implement the checks yourself. Brendan Eich

* == loose operator → big mistake → dangerous / bug-ridden

in fact two operator == and === consider the type of the values being compared. And if the comparison is between the same value type, both do exactly the same thing, no difference whatsoever → anyway later

The == differs from === in that it allows coercion before the comparison. In other words, they both want to compare values of like types, but == allows type conversions first and once the types have been converted to be the same on both sides, then == does the same things as ===.

Instead of loose equality → coercive equality.

NOTEBOOK

SUBJECT. if one's idea vs another's side story YEAR. MONTH. DAY.

e.g. 10 == "10" // True } try to convert to number if

"Hello" == 0 // false } can't return false.

e.g. var arr = ["1", "10", "100", "1000"]

for (let i = 0; i < arr.length && arr[i] < 500; i++) {

// will run 3 times

}

e.g. var x = "10" } both string → use alphabetical (dictionary-like) comparison.

var y = "9"

x < y // True, → important.

* The wiser approach is not avoid coercive comparisons, but to embrace (learn) and learn their ins and outs. detailed or complicated facts of somethings.

* How we organize in JS

Few major patterns for organizing code (data and behaviour) → value → modules / classes → method

we can do use both. In some respects, these patterns are very different, but in other ways, they're just different sides of the same coin. Being proficient in JS requires understanding both patterns and when they are appropriate (and not!)

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* classes

The terms "object-oriented", "class-oriented" and "classes"

→ They are not universal in definition.

→ we use common the one used in C++ and Java.

* A class in program is a definition of a "type" of custom data structure that includes both data and behaviour.

* A class must be instantiated (with new keyword) one or more times.

e.g: class Page() {
  constructor(text) {
    this.text = text
  }
  print() {
    clg(this.text)
  }
}

class Notebook {
  constructor() {
    this.pages = []
  }
  addPage(text) {
    const newPage = new Page(text)
    this.pages.push(newPage)
  }
  print() {
    for(let page of this.pages) {
      page.print()
    }
  }
}

define classes

const myNotebook = new Notebook()
myNotebook.addPage('first page')
myNotebook.addPage('second page')
myNotebook.print()
// first page
// second page

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* class Inheritance and polymorphism

A bit less used in JS.

Inheritance is a powerful tool for organizing data/behaviour in seperate logical units (classes), but allowing the child class to cooperate with the parent by accessing/using its behaviour and data.

both the inherited and overridden methods can have the same name and co-exist is called polymorphism.

e.g: class publication {
  constructor(title, author) {
    this.title = title
    this.author = author
  }
  print() {
    clg(this.title + " " this.author)
  }
}

class Book extends publication {
  constructor(title, author, ISBN) {
    super(title, author)
    this.ISBN = ISBN
  }
  print() { super.print()
    clg(this.ISBN)
  }
}

Used.

const newBook = new Book('hello', 'masoud', '123456')
newBook.print()
// hello
masoud
123456

polymorphism
inheritance

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* Modules: → classic modules
→ ES Modules (ESM)

The same goal as the class pattern, which is to group data and behaviour together into logical units.

- classic modules → we don't use this and new keyword

e.g.

```
function Publication(title, author) {
    var publicAPI = {
        print() { clg(
            title + author
        ) }
    }
    return publicAPI
}

function Book(bookdetails) {
    var pub = Publication(bookdetail.title, bookdetail.author)
    var publicAPI = {
        print() {
            pub.print()
            clg(bookdetail.ISBN)
        }
    }
    return publicAPI
}

var YDNJSY = Book('hello', 'masoud', '12345')
YDNJSY.print()  // hello
                masoud
                12345
```

The usage aka instantiation

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* There are other variations to this factory function from that are quite common across JS, even in 2020

- AMD (Asynchronous Module definition)
- UMD (Universal module definition)
- Commonjs (classic Nodejs-style modules)

All these rely on the same basic principles.

* ES Modules (ESM)

(1) introduced in ES6 → no wrapping function to define a module. The wrapping context is a file. ESM are always file-based. one file = one module.

(2) use export keyword to add a variable or method to its public API definition. if sth defined in module but not exported then it stayed hidden (just as with classic modules)

(3) you don't instantiate an ES module, you just import it to use its single instance.

e.g: publication.js file

```
function printDetails(title, author) {
    clg(title + author)
}
export function create(title, author) {
    var publicAPI = {
        print() { printDetails(title, author) }
    }
    return publicAPI
}
```

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

blogpost.js file:

```
import {create as createPub} from "publication.js"
function printDetails(pub, URL) {
    pub.print();
    console.log(URL)
}
export function create(title, author, URL) {
    var pub = createPub(title, author)
    var publicAPI = {
        print() {
            printDetails(pub, URL)
        }
    }
    return publicAPI
}
```

main.js file:

```
import {create as newBlogPost} from "blogpost.js"
var fooAgainst1 = newBlogPost("Hello", "marcus", "https")
fooAgainst1.print()   // Hello
                      marcus
                      https
```

* We do assume multiple instantiation, so these following snippets will mix both ESM and classic modules.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* As shown, ES modules can use classic modules internally if they need to support multiple instantiation. Alternatively we could have exposed a class from our module instead.
* Digging to the Roots of JS goal → how JS works → time-varying
* Iteration: (Repeat)

The iteration pattern has been around for decades, and suggests a "standardized" approach to consuming data from a source one chunk at a time. The idea is that it's more common and helpful to iterate the data source, to progressively handle the collection of data by processing the first part, then next, and so on, rather than handling the entire set all at once.

If we have many sets of data, we'll need iterative processing to deal with this data (typically, a loop).

The iterator pattern defines a data structure called an "iterator" that has a reference to an underlying data source (like the query result rows), which exposed method like next(). Calling next() return the next piece of data. We don't always know how many pieces of data that you will need to iterate through, so the pattern completes it by some special value or exception.

the importance of the iterator pattern is in adhering to a standard way of processing data iteratively, which

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

creates cleaner and easier way to understand code.
ES6 standardized a specific protocol for the iterator pattern directly in the language. The protocol defines a next() method whose return is an object called an "iterator result"; the object has "value" and "done" properties, where done is a boolean that is false until iteration over the underlying data source is complete.

* Consuming iterators → emit manual loop equivalent → less readable
- for ... of loop

e.g: var it = /* ... */;
for (let val of it) { clg(val) } // ...

- ... operator → two symmetrical forms (spread, Rest) ① ②
The spread for is an iterator-consumer.

① An array spread
var vals = [...it]

② a function call spread
e.g: doSomething(...it)

in both case → iterator-consumption protocol
(the same as for ... of loop)

* Iterables
The iterator-consumption protocol defined technically for consuming iterables. An iterable is a value that can be iterated over.
The protocol Automatically creates an iterator instance

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

from an iterable, and consumes just that iterator instance to its completion. This means single iterable could be consumed more than once, each time, a new iterator instance would be created and used.
ES6 defines the basic data structure/collection types in JS as iterables. Strings, arrays, maps, sets and ... ① ② ③ ④

eg1: var arr = [1,2,3]
for (let val of arr) { clg(val) } // 1,2,3

e.g 2: var arrCopy = [...arr]; → shallow copy an array

e.g 3: var greeting = 'Hello'
var chars = [...greeting]
chars; // ["H","E","L","l","o"]

e.g 4: var btnNames = new Map()
btnNames.set(btn1, 'btn1')
btnNames.set(btn2, 'btn2')
for (let [btn, btnName] of btnNames) {
btn.addEventListener('click', function onClick() {
clg(btnName);
});
}
→ array destructuring

e.g 5: for (let btnName of btnNames.values()) {
clg(btnName)
}

NOTEBOOK // Btn1
// Btn2

SUBJECT. YEAR. MONTH. DAY.

e.g 6. var arr = [10, 20, 30]

for (let [idx, val] of arr.entries()) {

cl(`[${idx}]: ${val}`) }

[0]: 10

[1]: 20

[2]: 30

available funcs: keys(), values(), entries()

↑ object

↓ array

iteration protocol

"standardizing" on this protocol means code that is overall more readily recognizable and readable.

* closure → it might even be as important to understand as variables or loops, that's how fundamental it is. we need to be able to recognize where closure is used in programs, as presence or lack of closure is sometimes the cause of bugs (or even the cause of performance issues).

* closure is when a function remembers and continues to access variables from outside its scope, even when the function is executed in a different scope.

we see two definitional characteristics:

① closure is part of the nature of a function. objects don't get closures, functions do.

② to observe a closure you must execute a function in a

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

different scope than where that function was originally defined.

e.g. function greeting(msg) {

return function who(name) {

cl(msg + name) }}

var hello = greeting('Hello')

var howdy = greeting('Howdy')

hello("Kyle"); // Hello, Kyle!

hello('Sarah'); // Hello, Sarah!

howdy("Grant") // Howdy Grant

First, The greeting(..) outer function is executed, creating an instance of the inner function who(..); that function closes over the variable msg, which is the parameter from the outer scope of greeting(..). When that inner function is returned, its reference is assigned to the hello variable in the outer scope. Then we call greeting(..) a second time, creating a new inner function instance, with new closure over a new msg, and return that reference to be assigned to howdy. When the greeting(..) function finishes running, normally we would expect all of its variables to be garbage collected (removed from memory). We'd expect each msg to go away, but they don't. The reason is closure. Since the inner function instances are still alive (assigned to hello and howdy). Their closures are still preserving the msg variables.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

These closures are not a snapshot of msg variable's value. They are direct link and preservation of the variable itself. That means closure can actually observe (or make) updates to these variables over time.

e.g. function counter(step = 1) {
    var count = 0;
    return function increaseCount() {
        count = count + step
        return count }}
var incBy1 = counter(1)
var incBy3 = counter(3)
incBy1() // 1
incBy1() // 2
incBy3() // 3
incBy3() // 6
incBy3() // 9

each instance of inner increaseCount() function is closed over both the 'count' and 'step' variable from its outer counter(..) function's scope. step remains the same over time but count is updated on each invocation of that inner function since closure is over the variables and not just snapshots of the values. these updates are preserved.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

closure is most common when working with asynchronous code:
e.g. function getSomeData(url) {
    ajax(url, function onResponse(resp) {
        clg(url + resp)})};
    getSomeData("https:...")
The inner function onResponse(..) is closed over url, and thus preserves and remembers it until the Ajax call returns and executes onResponse(..). Even though getSomeData(..) finishes right away, the url parameter variable is kept alive in the closure for as long as needed.
It's not necessary that the outer scope be a function, it usually is, but not always - just that there be at least one variable in an outer scope accessed from an inner function.
e.g. for (let [idx, btn] of buttons.entries()) {
    btn.addEventListener("click", function onclick() {
        clg(idx)})};
because this loop is using let declarations, each iteration gets new block-scoped (aka, local) idx and btn variables, the loop also creates a new inner onclick(..) function each time. That inner function closes over idx, preserving it for as long as the click handler is set on the btn. So when each button is clicked, its handler can print its associated idx value, because the handler remembers its respective idx variable. Remember this closure is not over the value (like 1, 2, 3), but over variable idx itself.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

*this keyword → one of most mechanism and most misunderstood.

① function's 'this' resolves to the function itself.

② 'this' points the instance that a method belong to.

→ both incorrect.

when a function is defined, it is attached to its enclosing scope via closure.

scope is the set of rules that controls how references to variables are resolved.

functions also have another characteristic besides their scope that influences what they can access. This is best described as an "execution context" and it's exposed to the function via it 'this' keyword.

"scope": static, contains fixed set of variables available at the moment and location you define a function.

"function's execution context": dynamic, dependent on how it is called (regardless where defined or called from)

"this": Dynamic, determined each time the function is called

"execution context": object whose properties are made available to a function while it executes.

"scope": object, the scope is hidden inside the JS engine it is always the same for that function, and its properties take the form of identifier variables available inside the function

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

① e.g. function classroom (teacher) {

return function study() { → this-aware function

cl(teacher + this.topic)}

var assignment = classroom("kyle")

it is a function that is dependent on its execution context.

② assignment(); // kyle, undefined → ops :(

→ call as a plain, normal function, without providing it any execution context.

since we are not in strict mode, context-aware function that are called "without any context specified" Default the context to the global object (window object in browser). As there is no global variable has topic property we get undefined in output

③ var homework = {

topic: "JS",

assignment: assignment

} this refer to

homework.assignment() // Kyle, JS

④ var homework2 = { topic: "Math" }

assignment.call(homework2) // kyle, Math

* third way to invoke a function with call(...) method which takes an object (homework2) to use for setting the this reference for the function call.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* The benefits of this-aware function and their dynamic context:

① more flexibility re-use a single function with data from different objects.

② A function that closes over a scope can never reference a different scope or set of variables but a function that has dynamic this context awareness can be quite helpful for certain tasks.

* prototypes

This → For function execution

prototype → for an object and specifically resolution of a property access. → linkage between two objects

The linkage is hidden behind the scenes, though there are ways to expose and observe it.

This prototype linkage occurs when an object is created it's linked to another object that already exists.

"prototype chain" → A series of objects linked together via prototypes.

The purpose (ie., from an object B to another object A) is so that accesses against B for properties/methods that B does not have, are delegated to A to handle.

Delegation of property/ properties method access allows two or more objects to cooperate with each other to perform a task.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

Consider normal literal object

var homework = { topic: "JS" }

homework object has a single property on it: topic. However, its default prototype linkage connects to the Object.prototype object, which has common built-in methods like toString() and valueOf(), among others.

e.g: homework.toString() → it works even though homework doesn't have toString() method defined. The delegation invoked object.prototype.toString() instead.

* Object linkage

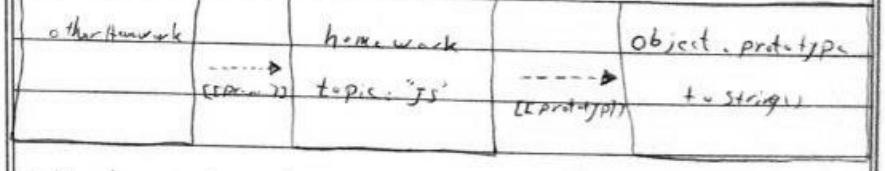
To define linkage, you can create the object using Object.create() utility:

e.g: var homework = { topic: "JS" }

var otherHomework = Object.create(homework)

otherHomework.topic // "JS"

returns the newly created (and linked) object

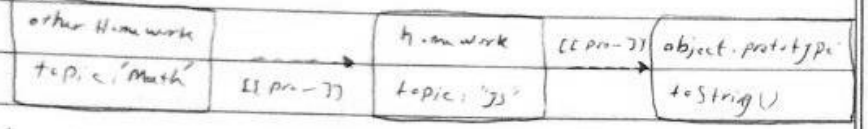


Object.create(null) → create object that is not prototype linked anywhere. purely standalone object that may be PK freeable.

NOTEBOOK

homework.topic // "JS"
otherHomework.topic // "JS"
otherHomework.topic = "math"
otherHomework.topic // "math"
homework.topic // "JS" -- not "math"

Delegation through the prototype chain only applies for accessing/lookup the value in a property. If you assign to a property of an object, that will apply directly to the object regardless of where that object is prototype linked to.



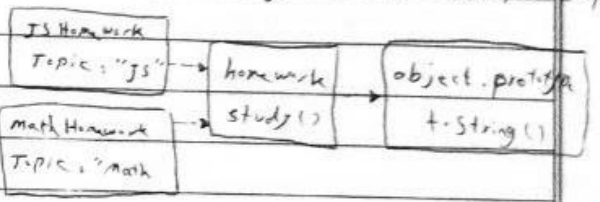
The topic on otherHomework is "shadowing" the property of the same name on the homework object in the chain.

* Another way of creating an object with prototype linkage is using "prototypal class" pattern. was added in ES6
* one of the main reasons 'this' supports dynamic context based on how the function is called is so that methods calls on objects which delegate through the prototype chain still maintain the expected 'this'.

NOTEBOOK

e.g. var homework = {
study() { cl(this.topic); }
var JSHomework = Object.create(homework)
JSHomework.topic = "JS"
JSHomework.study() // "JS"
var mathHomework = Object.create(homework)
mathHomework.topic = "Math"
mathHomework.study() // "math"

The two object JSHomework and mathHomework each prototype link to the single homework object, which has the study() function. JSHomework and mathHomework are each given their own 'topic' property.



JSHomework.study() delegates to homework.study, but its this (this.topic) for that execution resolves to JSHomework because how the function is called. similarity for mathHomework.study()
The preceding code snippet would be far less useful if 'this' was resolved to 'homework'. Yet in many other languages, it would seem 'this' would be homework because the study() method is indeed defined on homework.

* Asking (why?) Asking critical question becomes you better developer the right

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

Javascript under the hood - Traversy media.

* A memory leak is an unintentional form of memory consumption whereby the developer fails to free an allocated block of memory when no longer needed.
* Appendix A : Exploring further
* values vs. References

We have two types of values : primitives and objects

primitive values are always assigned/passed as value copies

e.g. var myName = "kyle"
var yourName = myName
myName = "Frank"
clg(myNam) // kyle
clg(yourNam) // Frank

improve that primitive values are always assigned/passed as value copies. ↘ DRY :D

Each variable holds its own copy of the value.

References are the idea that two or more variables are pointing at the same value, such that modifying this shared value would be reflected by an access via any of those references.

In JS only objects value (array, object, function, etc) are treated as references.

e.g. var myAddress = { street: '123 JS Blvd' }
var yourAddress = myAddress;
myAddress.street = "456 JS Blvd"
clg(yourAddress.street) // 456 JS Blvd

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

An update to one is an update to both.

Because the value assigned to myAddress is an object, it's held/assigned by reference, thus the assignment to the yourAddress variable is a copy of the reference, not the object value itself.

conclusion : JS chooses the value-copy vs. reference copy behaviour based on the value type. primitives are held by value, objects are held by reference. There's no way to override this in JS, in either direction.

* Back to chapter 4: The bigger picture
* Pillar 1 : scope and closure

Scope (functions, blocks) → The organization of variables into unit of scope → one of the most fundational characteristics of any language. perhaps no other has a greater impact on how programs behave.

scope like bucket
variables like marble
scope model → the rules that help determine which color marbles go in which matching-color buckets.

scope nest inside each other
for any given expression or statement
higher/outer scopes are accessible
lower/inner scopes are hidden and inaccessible (only variables at that level of scope nesting)
This is how scopes behave in most languages, which is called lexical scope.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

JS is lexically scoped, though many claim it isn't because of two things that are not present in other lexically scoped languages: 1) if declared at the beginning of a script, all variable declarations and function declarations in a script are treated as if they were declared at the beginning of the script. 2) Variables declared with var are function scoped, even inside a block. These are not sufficient reasons that tell us the JS is not lexical scope.

Let/const declarations have Peculiar error behaviour called TDZ (Temporal dead zone) which results in observable but unusable. It is the area of a block where a variable is inaccessible until the moment the computer completely initializes it with a value. eg:

```
// TDZ starts here
console.log(food) // returns Error, TDZ continues
// TDZ continues
let food = 'pizza' // TDZ ends here
// TDZ does not exist here
```

closure is natural results of lexical scope

* pillar 2 : Prototypes.

JS is one of very few languages where you have the option to create objects directly and explicitly, without first defining their structure in a class.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

For many years, people implemented the class design pattern on top of prototypes - so called "prototypal inheritance" and then with the advent of ES6's class keyword, the language doubled down on its unique approach to OOP programming. Author thinks, this hidden the beauty and power of the prototype system: the ability for two objects to simply connect with each other and co-operate dynamically (during function/method execution) through chaining this keyword. He thinks, behavior delegation (connecting objects as objects let objects connect through the prototype chain) is more powerful than class inheritance. But class inheritance gets almost all the attention. And the rest goes to functional programming (FP). This got author sad, because destroy any chances for explanation of delegation as a workable alternative → anti-class way.

Pillar 3: Types and coercion

A tidal wave of interest in the broader JS community has begun to shift to "static typing" approaches, using type-aware tooling like TypeScript or Flow.

This pillar is more important than the other two.

* with the Grain (highlight sb doing sth difficult)

Some advice to share:

① Consider the grain (How most people approach and use JS)

NOTEBOOK

Console-ninja / Quokka .js / in your face

SUBJECT. Tahid Ebnalia / shabnam mahbub YEAR. MONTH. DAY.

If you ever want to break out from the crowd, you're going to have to break from how the crowd does it.

② Don't just parrot what I say. own your opinions, defend them.

③ Don't be afraid to go against the grain.

④ Don't read this books and then try to change all that grain in your existing project over night. That approach will always fail. It's better to shift those things little by little over time.

⑤ always keep looking for better way to use what JS gives us to author more readable code. Everyone who works on your code, including your future self, will thank you.

* Continue Appendix A:
* So many function form

```
var awesomeFunction = function(coolthings) {
    // ..
    return sthAmazing;
};
```

Function expression

→ Its anonymous function expression, since it has no name.

JS Perform a "name inference" on an anonymous function

```
awesomeFunction.name;   // awesomeFunction
```

→ This property of a function reveal either its directly given name (in the case of declaration) or its inferred name in the case of anonymous function. This value generally used

NOTEBOOK by developer tools when inspecting a function value or when reporting an error stack trace.

SUBJECT. YEAR. MONTH. DAY.

name inference only happens in limited case. If you pass a function expression as an argument to a function call, the name property will be an empty string, and the developer console will usually report "anonymous function"

e.g.
```
function dosth(callback) {
    clg(callback.name)   // empty string ' '
}
```

but Chrome console we get the name of anonymous function. It's important to remember that this behaviour is specific to certain JS environments and may not be consistent across all browsers or platforms. The ECMAScript specification doesn't define a name for anonymous function, so different engines are free to handle it differently. Don't rely on "name" property in production code.

Even if a name is inferred, it's still anonymous function. This function doesn't have an identifier to use to refer itself (recursion) event unbinding, etc. → remove eventListener of an HTML Element

```
var myFunction = function SomeName(coolthings) {
    return AmazingStuff;
}
```

→ named function expression. (most priority)

```
myFunction.name   // SomeName
```

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* Here some more declaration forms:

```
// Generator function declaration
function *two() {..}
// Async function declaration
async function three() {..}
// async generator function
async function *four() {..}
// named function export declaration (ES6 modules)
export function five() {..}
// IIFE
(function() {..})();
(function namedIIFE() {..})();
// asynchronous IIFE
(async function() {..})();
(async function namedAIIFE() {..})();
// arrow function
var f;
f = () => 42;
f = x => x * 2
f = (x) => x * 2
f = (x, y) => x * y
f = x => ({ x: x * 2 })
f = x => { return x * 2 }

f = async x => {
    var y = await doSth(x);
    return y * 2;
}
someoperation( x => x * 2)
```

IIFE: ایمنی → capsulation ← [illegible]; scope محدود کردن

arrow function → this [illegible]

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

Keep in mind that arrow function expressions are syntactically anonymous, meaning the syntax doesn't provide a way to provide a direct name identifier for the function.

He thinks, arrow function or anonymous function use frequently in our program, is not good enough. Use the most appropriate tool for each job.

Functions use in class definition

```
class myClass {
    // class methods
    coolMethods() {..}    // no commas!
}
var object1 = {
    // object method
    coolMethod() {..},    // commas!
    boringMethod() {..}
    oldSchool: function() {..}
}
                // anonymous function expression property
```

study them closely and practice.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* Coercive Conditional Comparison

```
var x = 1;
if (x) {
    // will run
}
while (x) {
    // will run once
    x = false
}
```

like this ⟹

```
var y = 1;
if (x == true) {
    // will run
}
while (x == true) {
    // will run, once
    x = false
}
```

```
var x = "Hello"
if (x) {
    // will run
}
if (x == true) {
    // won't run :(
}
```

like it ⟹

```
var x = "Hello"
if (Boolean(x) == true) {
    // will run
}
// which is the same as:
if (Boolean(x) === true) {
    // will run
}
```

* Boolean always return boolean. The important part of is to see that before the comparison, a coercion occurs, from whatever type x currently is, to boolean.
You just can't get

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

* prototypal "classes" → quite uncommon in JS these days!

(prototypal)

```
var classroom = {
    welcome() {
        cl('welcome')
    }
}

var mathClass = Object.create(classroom)
mathClass.welcome(); // welcome
```

(prototypal Class)

```
function Classroom() {
    // ...
}
Classroom.prototype.welcome = function hello() {
    cl('welcom')
}
var mathClass = new Classroom();
mathClass.welcome() // welcom
```

* Though mathClass does not have a welcome property/function it delegates to the function classroom.prototype.welcome()
this prototypal class pattern is now strongly discouraged, in favor of using ES6's class mechanism.

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

```
class Classroom {
    constructor() {
        ...
    }
    welcome() {
        alg("welcome")
    }
}
var mathClass = new Classroom()
mathClass.welcome(), // welcome
```

* under the covers, the same prototype linkage is wired up, but this class syntax fits the class-oriented design pattern much more cleaner than "prototypal classes"

NOTEBOOK

SUBJECT. YEAR. MONTH. DAY.

NOTEBOOK