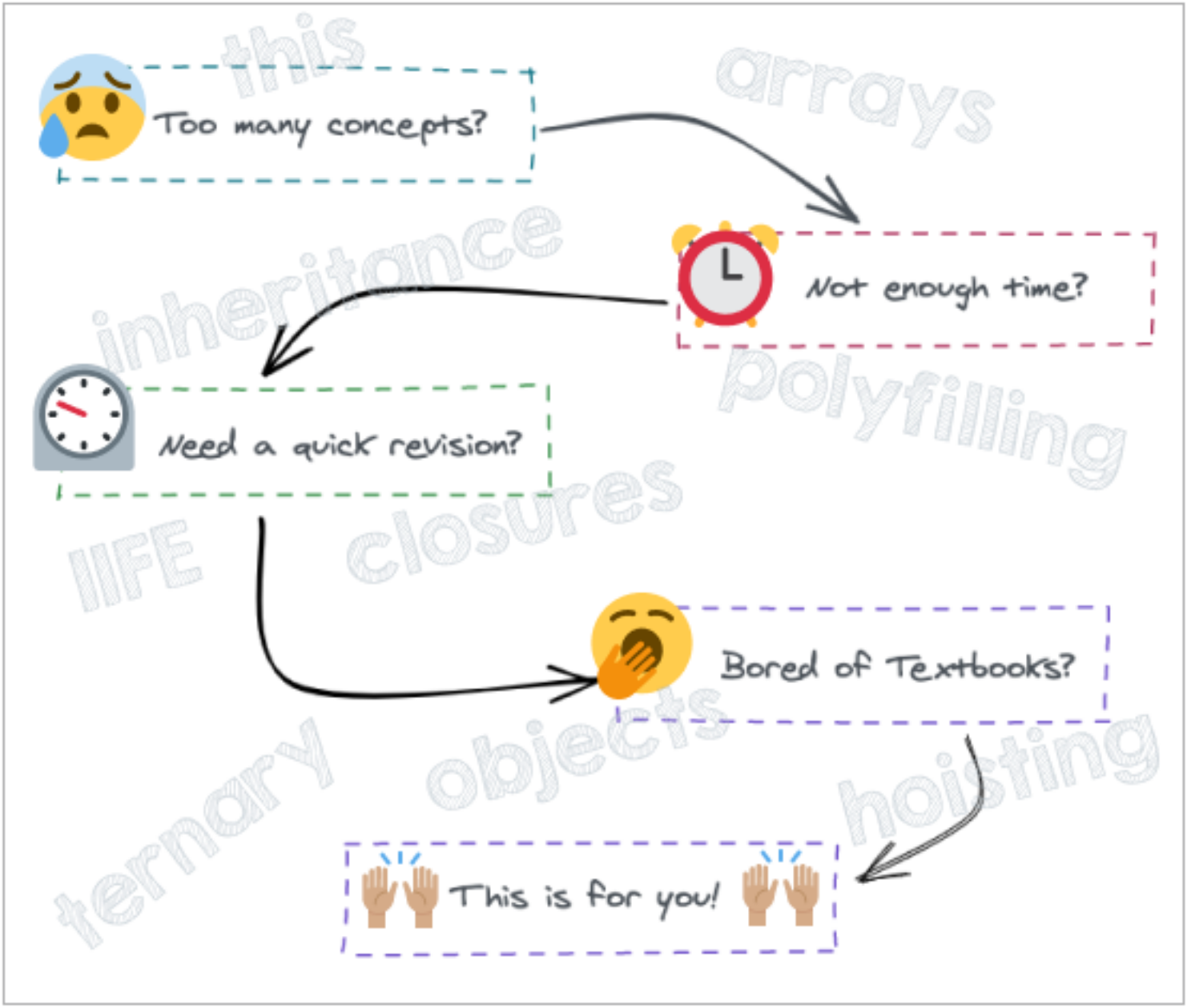
JAVASCRIPT ILLUSTRATED

A sketchnotes compilation of important JS concepts!



Created with ♥ by @kokaneka

JavaScript Illustrated

A handbook created with 🖤 by @kokaneka

Index

#SKETCHNOTES

You don't know JS

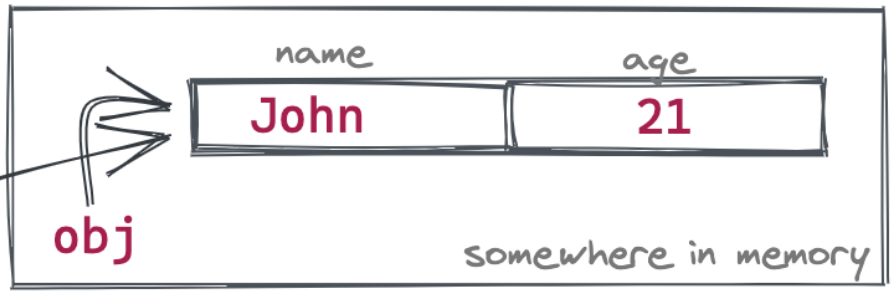
JAVASCRIPT OBJECTS

Javascript objects are called by reference

What does that mean?

The variable storing the object does not store the value of the object but a reference to the memory where the object stays.

```
let obj = {
  name: "John",
  age: 21
}
```




```
let newObj = obj;
```

Doing this assigns the same reference to newObj

```
newObj.name = "Jane"
```


```
console.log(obj.name)
```

logs: Jane

@kokaneka

sketchnotes YDKJS

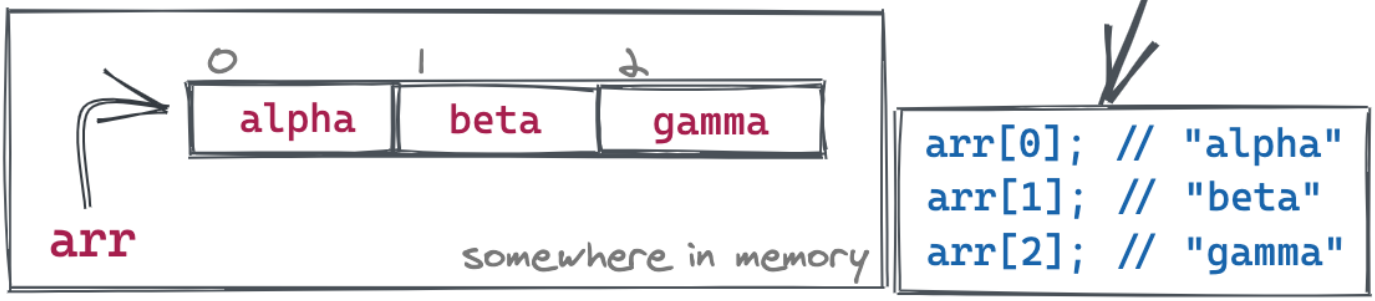
#SKETCHNOTES

You don't know JS

JAVASCRIPT ARRAYS

Javascript arrays can be roughly considered as objects where the keys are numbers, instead of properties.

```
var arr = ['alpha','beta','gamma'];
```



The similarity does not ends there, arrays are recognized as having the type of objects

```
typeof arr; //"object"
```

Also, arrays have the auto updated "length" property.

```
arr.length; // 3
```

Fun fact: The "length" property returns 1 number greater than the highest index that is populated in the array!

```
arr[8] = 'kappa'
```

arr now holds 'kappa' in the 9th index

```
console.log(arr.length)
```

Logs 9

even though the array "has" just 4 items

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

`===` Versus `==`

=== is strict comparison

Does not allow the values being compared to be coerced

== is loose comparison

Allows the values being compared to be coerced

`===`

`42 === "42"`

Hey Engine, are these 2 values strictly equal?

The types of those two values are different. There is no parallel universe where these two values could be equal. Please don't ask again, I'm busy.

It's a hard false bro.

`==`

`42 == "42"`

Hey Engine, are these 2 values loosely equal?

Umm... They are not of same type. Let me see what I can do here. I'll convert the string to a number and get back to you.

after coercion...

Well well.. Look what we have there Turns out, they converge to the same value afterall!

Here, it's a true!

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

TRUTHY Versus FALSY

What are truthy values?

Values, that when coerced, convert to true

x → coercion → true

What are falsy values?

Values, that when coerced, convert to false

y → coercion → false

FALSY

TRUTHY

FALSY	TRUTHY
""	"hello", 22
0, -0, NaN	[], {}
null, undefined	[1, 2], {a: 'yo'}
false	true

A thing worth noting that an empty array and object are truthy!

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

ARRAY to STRING coercion

When arrays are coerced to strings, the numbers are joined with a comma in between!

`[4, 5, 6, 7]` becomes `"4,5,6,7"`

Coming to comparison,

```
let a = [4, 5, 6, 7]
let b = [4, 5, 6, 7]
let c = "4,5,6,7"
```

`a == c` **true** — a gets coerced to this string and matches to c

`b == c` **true** — b gets coerced to this string and matches to c

`a == b` **false** — arrays are not primitive types in JS Hence the variables store references and not values

@kokaneka

sketchnotes YDkJS

#SKETCHNOTES

You don't know JS

var V/s let SCOPE

var and let behave quite differently when it comes to scope and hoisting. Both allow variable access in all inner scopes. But outer scope behavior differs.

```
function foo() {
  var a = 1;
  let b = 2;

  function bar() {
    var c = 2;
    console.log(d);
    let d = 2;

    if (d == 2) {
      var e = 1;
      let f = 2;
    }

    console.log( a, b);
    console.log(e);
    console.log(f);
  }

  bar();
  console.log( a );
}
```

Variables declared at the topmost scope available everywhere inside.

Newly defined inside c

error

Logging before declaring gives error because of 'let'

Available from the other scope

This is fine, as 'e' gets hoisted to the top of function bcoz of 'var'

error

This gives error cause 'let' scope is limited to the if block

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

HOISTING

hoisting is the phenomenon where a var declaration is conceptually "moved" to the top of its enclosing scope

```
function foo() {
  a = 3;

  console.log( a );

  var a;
}
```

assigned a value before being declared

console logged before being declared, logs 3

```
function foo() {
  var a;
  console.log( a );

  a = 3;

  console.log( a );

  var a;
}
```

Behaves as if declared at the top of the scope

Logging it here logs undefined

Pro Tip: Only the variables declared with the 'var' keyword get hoisted and not the ones declared using 'let'

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

Ternary Operator

The ternary operator, also known as the conditional operator acts as a replace for if, else-if and also saves space.

```
var b;
if (a > 41) {
    b = "hello";
}
else {
    b = "world";
}
```

USE CASE

Assigning a value to the variable 'b' based on the value of 'a'

ADVANTAGES

- Saves Lines of code
- More readable

same result using ternary operator:

```
var b = (a > 41) ? "hello" : "world";
```

false

true

check if 'a' is greater than 41

If the expression is true, return this value.

Else, return this value.

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

IIFE in Javascript

IIFE is the short form for Immediately invoked function expression.

declare a function

```
(function IIFE(){
        console.log( "Hello!" );
})();
```

Invoke it immediately after declaration

The brackets just tell the compiler that this is not a normal JS function

What can IIFEs be useful for?

"Hiding" a variable from the outer scope.

'a' in the outer scope

```
var a = 42;

(function IIFE(){
  var a = 10;
  console.log( a ); // 10
})();

console.log( a );
```

'a' created here does not "pollute" the overall scope!

USE CASE? Some libraries are exposed as IIFEs to prevent collision of variables!

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

CLOSURE IN JAVASCRIPT

Closures can be thought of as a way for functions to "remember" their scope variables even after having finished running.

```
function addFiveFactory() {
  let five = 5;

  function addFive(x) {
    return x+5;
  };

  return addFive;
}

const addFive = addFiveFactory();
```

A function factory that is returning us the addFive() function.

The variable, five is declared inside the scope of the factory.

Runs the factory function and returns us the addFive function.

Notice that by this time, the addFiveFactory has stopped running and is out of the function stack, but still,

```
addFive(3); // returns 8 i.e. (3 + 5)
```

So, how did the addFive function get a reference to the variable 'five' which was outside of its scope and even though the factory function exited long back?

'five' was stored in the 'closure scope' of addFive()!

@kokaneka

sketchnotes YDkJS

#SKETCHNOTES

You don't know JS

'this' in JAVASCRIPT

Contrary to what most beginner Javascript developers believe, the 'this' keyword in javascript does not point to 'itself' (a common answer I received)

Here are 4 common conditions that the value depends on :

```
var ctx = "global";

function logCtx() {
  console.log( this.ctx );
}

logCtx(); global
```

Inside a function sitting in the global context, 'this' points to global.

```
var obj1 = {
  ctx: "obj1",
  logCtx: logCtx
};

obj1.logCtx(); obj1
```

Adding the function as an object attribute and later calling the function on that object sets that object (the caller) as 'this'.

```
var obj2 = {
  ctx: "obj2"
};

logCtx.call( obj2 ); obj2
```

Setting the 'this' explicitly with the 'call' method sets the value of 'this' with the supplied argument.

```
new logCtx(); undefined
```

Prefixing the function call with the 'new' keyword sets the 'this' value for the function to be a brand new empty object.

@kokaneka

#SKETCHNOTES

You don't know JS

Prototypal Inheritance in JS

In JS, every object has a prototype reference & if a property is not found on an object then a lookup happens on the prototype reference & up the chain.

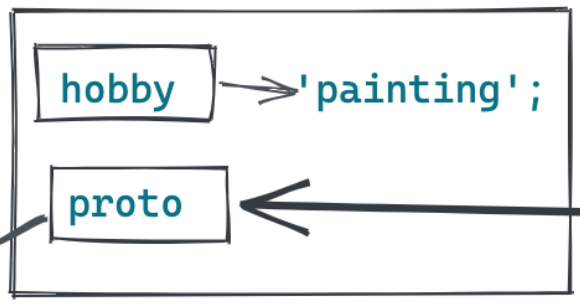
Here's how that works:

```
var parent = {
  skill: 'cooking'
};
```

An object with a key 'skill' having the value 'cooking'.

Creating an object named 'child' with its prototype ref to parent

```
var child = Object.create( parent );
child.hobby = 'painting';
```



'child' object having the 'parent' object as prototype

This is not found on the object so looked up the prototype chain.

This is just a key access.

```
console.log(child.hobby)
```

painting

```
console.log(child.skill)
```

cooking

This behavior of the language is used in order to emulate inheritance

@kokaneka

sketchnotes YDKJS

#SKETCHNOTES

You don't know JS

Polyfilling in JavaScript

Polyfill is that piece of code which runs on an older environment (that does not support a certain feature) but provides the equivalent code behavior as if the feature was supported natively on that environment.

Let's say that the ECMA standard comes up with a new utility to detect whether an Array has the number 13 in it and calls it isUnluckyArray for what-so-ever reason!

Expected behavior

```
let arr1 = [10, 12, 13, 15]; let arr2 = [13];
let arr3 = [1, 2, 3, 4];
arr1.isUnluckyArray()  true
arr2.isUnluckyArray()  true
arr3.isUnluckyArray()  false
```

These are unlucky as they contain the number 13

This is not an 'unlucky' array

This behavior can be polyfilled using this piece of code:

```
if(!Array.prototype.isUnluckyArray) {
  Array.prototype.isUnluckyArray = function () {
    return this.includes(13);
  }
}
```

First check whether the functionality exists

Notice how we are adding the functionality to the Array.prototype so that it is available on all arrays.

@kokaneka

#SKETCHNOTES

You don't know JS

WEIRD STUFF JS

Accessing a variable throws a ReferenceError if the variable is not available in the scope, but setting it's value generally won't throw!

```
function foo() {
    console.log(a);
}
```

Trying to access 'a' which does not exist.

Throws ReferenceError

```
function foo() {
    a = 20;
}
```

Trying to set 'a' which does not exist.

Silently goes through and creates an 'a' on the global scope whose value is 20.

```
console.log(window.a)  // Logs 20
```

How can this be avoided?

STRICT MODE!

This same code throws an error when strict mode is enabled

```
function foo() {
    a = 20;
}
```

@kokaneka

sketchnotes YDKJS

Connect with me on Twitter for more frequent updates ?