Behind the scenes personal guide

JavaScript is a high level, prototype-based object-oriented, multi-paradigm, interpreted or just-in-time compiled, dynamic, single- threaded, garbage-collected programming language with first-class functions and a non-blocking event loop concurrency model.

High-level: Developer does NOT have to worry about managing resources because JavaScript has abstractions but programs will never be as fast or as optimized in comparison to C programs.

Garbage-collected: An algorithm inside the JavaScript engine which automatically removes old unused objects from the computer memory.

Interpreted or just-in-time compiled: The computers processor only understands 0's and 1's. The JavaScript engine converts our abstraction to machine code then executed immediately.

Multi-paradigm: Paradigm in programming is an approach or mindset of structuring code, which will direct your coding style and technique. Paradigm can be classified as imperative or declarative. JavaScript does all the below:

1. Procedural programming
2. Object Oriented programming
3. Functional programming

Prototype-based object-oriented: Almost everything in JavaScript is an object, Except for primitive values. You can use the push method on an array because of prototypal inheritance. We create arrays from an array blueprint which is like a template which is called the prototype. This protype contains all the array methods and the array we create in our code then inherit the methods from the blueprint.

First-class functions: Functions are treated just as regular variables. So, we can pass them into other functions, and return them from functions.

Dynamic: Dynamically typed. We don't assign data types to variables, types become known at runtime. Data type of variable is automatically changed. This is good and bad, its controversial. Look into Typescript if you want strongly typed JavaScript.

Single-threaded & Non-blocking event loop concurrency model: **Concurrency model** is just how the JavaScript engine handles multiple tasks happening at the same time. We need that because JavaScript runs in one **single thread** so it can only do one thing at a time. (in computing a thread is like a set of instructions that is executed in the computers CPU, the thread is where our code is actually executed in the machines processor) So what about a long running task like fetching data from a remote server? Using an **event loop** takes long running tasks, executes them in the "background", and puts them in the main thread once they are finished.

this keyword / variable

*A special variable that is created for every execution context. It takes the value of (points to) the "owner" of the function in which the ***this*** keyword is used.

****this*** is NOT static. It depends on how the function is called, and its value is only assigned when the function **is actually called**.

Four ways which functions can be called using *this* keyword : not included, new, call, apply, or bind.

Method (a function attached to an object) → ***this*** = < Object that is calling the method >

```
const kelsy = {
    name: 'Kelsy',
    year: 1991,
    calcAge: function () {     // calcAge is the method
        return 2022 - this.year; {   // object kelsy is value of this keyword
    }
};
console.log(kelsy.calcAge());
```

Simple function call (not attached to any object) → ***this*** = undefined

> **only undefined in strict mode otherwise: points to global object which is the window object in the browser.

Arrow functions → ***this*** = < ***this*** of surrounding function or parent function (lexical ***this keyword***) >

> ** Arrow functions do not get their own ***this*** keyword. It gets picked up from the outer lexical scope of the arrow function.

Event listener → ***this*** = < DOM element that the handler is attached to >

****this*** does **NOT** point to the function itself, and also **NOT** the variable environment of the function.

Primitive vs Objects

Primitives: number, string, Boolean, undefined, null, symbol, bigint

Objects: object literal, arrays, functions, etc…

Primitives -> 'primitive types' when referring to memory

Objects -> 'reference types' when referring to memory

REMEMBER: Js Engine consists of Call Stack (where functions are executed) and the Heap (where objects are stored in memory)

Objects can be const and mutable because their memory address in the heap is stored in the stack and is never actually changed, any changed to the object happens in the heap which is allowed

Primitive types → stored in → Call Stack Heap ← stored in ← Objects

PRIMITIVE VALUES EXAMPLE

```
let age = 30; //line1
let oldAge = age; //line2
age = 31;         //line3
console.log(age);    //31
console.log(oldAge); //30
```

Identifier points to the address and not the value itself

So, age variable is equal to memory address 0001 which holds the value of 30

oldAge points to same memory address as age variable

--the value at a certain memory address cannot be changed, a new piece of memory is allocated

When setting age to 31 memory address 0002 is created and age identifier points to that

CALLSTACK

Line	Identifier	Address	Value
Line1 – Line 3	age	0001	30
Line2	oldAge	0002	31

REFERENCE VALUE EXAMPLE

1. When a new object is created it is stored in the heap with memory address then value itself
2. Me identifier points to new piece of memory in the stack
3. Memory in the stack points to the object in the heap by using memory address as its value

The piece of memory in the call stack has a reference to the piece of memory in the heap which hold(me) object which is why their called reference types! We do this because an object may be to large to be stored in the stack.

Friend can be const because you're not changing the value in the call stack your changing the value in the heap. Primitive const values cannot be changed but reference values can be.

```
const me = {          //line1
    name: 'Kelsy',
    age: 30,
}

const friend = me;    //line6
friend.age = 27;      //line7

console.log('friend:', friend); //27
console.log('me:', me);     //27
```

CALLSTACK

Line	Identifier	Address	Value
Line1-4	me	0003	D30F
Line6	friend		

HEAP

Line	Address	Value

Line1-4
 	
Line 6
	D30F	{
		name: 'Kelsy';
		age: 30;
		}

Line 7		{
		name: 'Kelsy';
		age: ~~30~~; 27;
		}

```
const kelsy2 = {        //Line1
    firstName: 'kelsy',
    lastName: 'watkins',
    age: 30,
    family: ['Matt', 'Bella'],
};
const kelsyCopy = Object.assign({}, kelsy2);  //line7
kelsyCopy.lastName = 'Auretta';  //line8

console.log('before marriage: ', kelsy2);
console.log('After marriage: ', kelsyCopy);

kelsyCopy.family.push('Smella'); //line13
kelsyCopy.family.push('Stinkus'); //lin14

console.log('family: ', kelsy2); //family size 4
console.log('family: ', kelsyCopy); //family size 4
```

Object.assign did not copy smella and stinkus to the new object because its depply nested

HEAP

Line	Address	Value

Object kelsy2	D30F	{
		firstName: 'kelsy',
		lastName: 'watkins',
		age: 30,
		};

Array is object and gets nested address on heap	D30S
	THIS BELONGS TO BOTH
		{
		family: ['Matt', 'Bella'], + ‘smella’, ‘stinkus’
		}

Change watkins to kelsy works – top level	D30F-copy	{
		firstName: 'kelsy',
		lastName: 'watkins', ‘auretta’
		age: 30,
		};

Line	Identifier	Address	Value
Line1-6	kelsy	0003	D30F
Line 7 and 8	kelsyCopy		D30F-Copy
			D30S