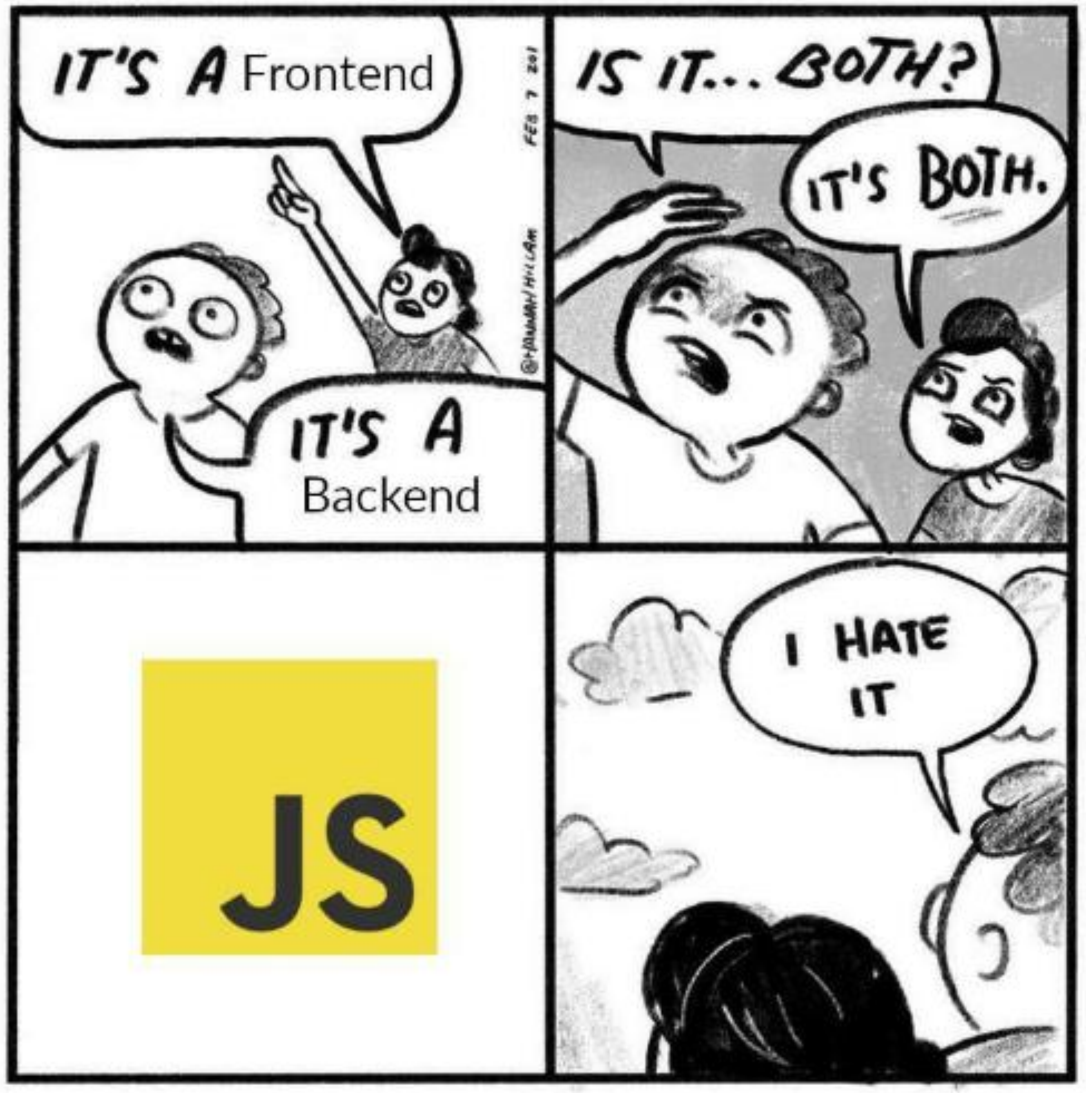


It's a Meteor !

Variables

- Variables are containers for storing data values in a program.
- They **provide named references to memory locations**, allowing you to manipulate data.
- In JavaScript, variables are declared using the keywords **var, let, or const**.

Function-level Scope vs. Block-level Scope:

Function-level Scope:

- Variables declared with var have function-level scope.
- They are accessible throughout the entire function, regardless of block structures.

Block-level Scope:

- Variables declared with let and const have block-level scope.
- They are limited to the block or statement in which they are declared.

Variable Declaration

var - Function-level Scope:

Variables declared with var are function-scoped, meaning they are accessible throughout the entire function in which they are declared.

```
function exampleFunction() {
    if (true) {
      var functionScopedVar = "I am function-scoped";
    }
    console.log(functionScopedVar); // "I am function-scoped"
  }
  exampleFunction();

  console.log(functionScopedVar); // Error: functionScopedVar is not defined
```

let - Block-level Scope:

- Variables declared with let are block-scoped, meaning they are accessible only within the block in which they are defined.
- let declarations are hoisted, but there is a temporal dead zone where they are not accessible before the declaration.

```
if (true) {
    let blockScopedVar = "I am block-scoped";
    console.log(blockScopedVar); // "I am block-scoped"
  }

  console.log(blockScopedVar); // Error: blockScopedVar is
not defined
```

const - Block-level Scope with Constant Value:

- Variables declared with const are block-scoped like let, but they cannot be reassigned.
- They must be initialized when declared, and their value cannot be changed.

```
const constantVar = "I am constant";

//constantVar = "New value"; // Error: Assignment to a
constant variable

if (true) {
  console.log(constantVar); // "I am constant"
}
```

Try this program

```
function exampleFunction() {
    if (true) {
      var functionScopedVar = "I am function-scoped";
      let blockScopedLet = "I am block-scoped";
      const blockScopedConst = "I am constant and block-scoped";
    }

    console.log(functionScopedVar); // "I am function-scoped"
    console.log(blockScopedLet); // Error: blockScopedLet is not defined
    console.log(blockScopedConst); // Error: blockScopedConst is not 
defined
  }
  exampleFunction();
```

Shadowing:

- Local variables can "shadow" global variables with the same name.
- The local variable takes precedence within its scope.

```
var shadowedVar = "I am global";

function exampleFunction() {
  var shadowedVar = "I am local";
  console.log(shadowedVar); // "I am local"
}

console.log(shadowedVar); // "I am global"
```

Hoisting

Hoisting is a JavaScript behavior where variable and function declarations are moved to the top of their scope during the compilation phase before the code is executed. **This behavior allows you to use variables and functions before they are declared in the code.**

However, it's important to note that **only the declarations are hoisted, not the initializations**.

Hoisting with ‘Var’

```
console.log(x); // undefined
var x = 5;
console.log(x); // 5
```

The first console.log(x) outputs undefined because the declaration is hoisted, but the assignment (x = 5) happens later in the code.

Hoisting with let and const:

```
// console.log(y); // Error: Cannot access 'y' before
initialization
let y = 10;
console.log(y); // 10
```

With let and const, there is also hoisting, but unlike var, the variables are not initialized at the top. Accessing them before the declaration results in a ReferenceError because they are in the "temporal dead zone" until they are actually declared in the code.

Thing to note:

- Variables declared with let or const are hoisted WITHOUT a **default initialization**. So, accessing them before the line they were declared throws ReferenceError: Cannot access 'variable' before initialization.

- But variables declared with var are hoisted WITH a default initialization of undefined. So, accessing them before the line they were declared returns undefined.

When I’m coding HTML
When I’m coding CSS
When I’m coding Javascript

Hoisting with Functions

```
hello(); // "Hello, world!"

function hello() {
  console.log("Hello, world!");
}
```

Function declarations are also hoisted, so you can call a function before it appears in the code. This is why the hello() function can be invoked before its declaration.

Hoisting Caveats:

- While the declarations are hoisted, the initializations are not, so trying to access the value of a variable before its declaration and assignment will result in undefined.
- **It's generally considered good practice to declare variables at the top of their scope to avoid confusion.**
- Understanding hoisting is crucial for writing predictable and error-free JavaScript code.

Understanding JavaScript Data Types:

1. Primitive Data Types:

a. String:

Represents textual data.

Enclosed in single (') or double (") quotes.

```
let name = "John";
```

b. Number:

Represents numeric values.

Can be integers or floating-point numbers.

```
let age = 25;
let pi = 3.14;
```

c. Boolean:

Represents true or false values.

Used for logical operations.

```
let isStudent = true;
```

d. Undefined:

Represents the absence of a defined value.

Variables that are declared but not assigned a value are undefined.

```
let undefinedVar;
```

e. Null:

Represents the intentional absence of any object value.

It's a special value that indicates a variable has no value or no object assigned.

```
let nullVar = null;
```

2. Object Data Type:

a. Object:

A collection of key-value pairs.

Keys are strings, and values can be of any data type, including other objects.

```
let person = {
    name: "Alice",
    age: 30,
    isStudent: false
  };

console.log(person.name);
```

b. Array:

An ordered list of values.

Accessed by index, starting from 0.

```
let fruits = [1, "banana", "orange"];
console.log(fruits[0]);
```

c. Function:

A reusable block of code.

Can take parameters and return a value.

```
function add(a, b) {
    return a + b;
  }

var a=add(2,3);
console.log(a);
```

3. Dynamic Typing:

JavaScript is dynamically typed, meaning variable data types are determined at runtime.

The same variable can hold different types of values.

```
let dynamicVar = 10; // dynamicVar is a number
dynamicVar = "Hello"; // dynamicVar is now a string
```

4. Typeof Operator:

The typeof operator is used to determine the data type of a variable.

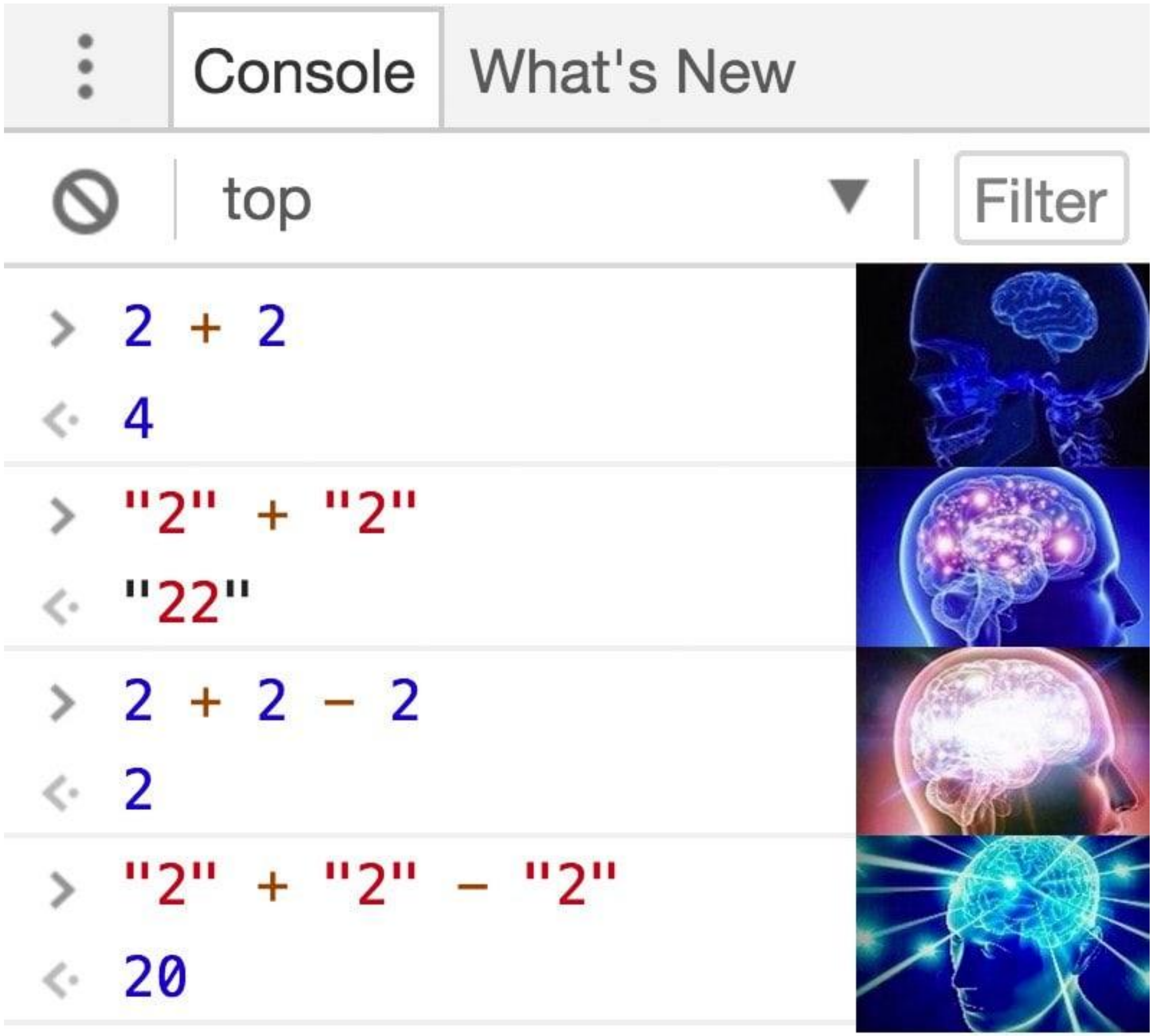
```
let exampleVar = "Hello";
console.log(typeof(exampleVar)); // "string"
```

5. Type Coercion:

JavaScript performs type coercion when operators are used with different data types.

Values are automatically converted to a common type during an operation.

```
let result = 5 + "5"; // Result is the string "55"
console.log(result);
```



```
> 2 + 2
<· 4
> "2" + "2"
<· "22"
> 2 + 2 - 2
<· 2
> "2" + "2" - "2"
<· 20
```

Working with Operators:

1. Arithmetic Operators:

Perform basic mathematical operations.

```
let x = 5;
let y = 2;

console.log(x + y); // Addition: 7
console.log(x - y); // Subtraction: 3
console.log(x * y); // Multiplication: 10
console.log(x / y); // Division: 2.5
console.log(x % y); // Modulus: 1
```

2. Comparison Operators:

Compare values and return a Boolean result.

```javascript
let a = 10;
let b = 5;

console.log(a > b);  // Greater than: true
console.log(a < b);  // Less than: false
console.log(a >= b); // Greater than or equal to: true
console.log(a <= b); // Less than or equal to: false
console.log(a === b); // Equal to: false
console.log(a !== b); // Not equal to: true
```

Math Class

1 = 1

1 ≠ 2

Normal Coding Languages

```
1 == 1
1 != 2
```

Javascript

```
1 === 1
1 !== 2
```

3. Logical Operators:

Combine or manipulate Boolean values.

```
let p = true;
let q = false;

console.log(p && q); // Logical AND: false
console.log(p || q); // Logical OR: true
console.log(!p);     // Logical NOT: false
```

4. Assignment Operators:

Assign values to variables.

```
let num = 10;
num += 5; // Equivalent to: num = num + 5
num -= 3; // Equivalent to: num = num - 3
num *= 2; // Equivalent to: num = num * 2
num /= 4; // Equivalent to: num = num / 4
```

Working with Loops:

1. for Loop:

Executes a block of code a specified number of times.

```
for (let i = 0; i < 5; i++) {
    console.log(i);
  }
```

2. while Loop:

Repeats a block of code while a specified condition is true.

```
let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}
```

3. do-while Loop:

Similar to a while loop, but it ensures that the code block is executed at least once.

```
let count = 0;
do {
  console.log(count);
  count++;
} while (count < 5);
```

•

4. for...in Loop:

Iterates over the properties of an object.

```
let person = { name: "John", age: 30, occupation:
"Developer" };
for (let key in person) {
  console.log(key + ": " + person[key]);
}
```

Understanding JavaScript Objects:

- Objects in JavaScript are collections of key-value pairs.
- Keys are strings or symbols, and values can be of any data type, including other objects.
- Objects can contain properties (data) and methods (functions).
- Object literal notation is a concise way to create objects.
- Object constructors are used to create instances of objects with shared properties and methods.
- Objects can be iterated over to access their properties.

How to define objects?

```
// Object literal notation
let car = {
    make: "Toyota",
    model: "Camry",
    year: 2022,
    isElectric: false,
    start: function() {
      console.log("Engine started!");
    }
  };
```

```javascript
// Accessing properties
console.log(car.make);     // Outputs: Toyota
console.log(car.year);     // Outputs: 2022

// Calling a method
car.start(); // Outputs: Engine started!
```

Object Constructor

```
// Object constructor function
function Book(title, author, year) {
    this.title = title;
    this.author = author;
    this.year = year;
  }

  // Creating instances of the object
  let book1 = new Book("The Catcher in the Rye", "J.D. Salinger", 1951);
  let book2 = new Book("To Kill a Mockingbird", "Harper Lee", 1960);
```

Working with Arrays:

```javascript
// Creating an array
let fruits = ["apple", "banana", "orange", "grape"];

// Accessing elements
console.log(fruits[0]); // Outputs: apple
console.log(fruits[2]); // Outputs: orange
```

```javascript
// Adding elements to the end
fruits.push("kiwi");
console.log(fruits);
// Adding elements to the beginning
fruits.unshift("mango");
console.log(fruits);
// Removing the last element
let removedLast = fruits.pop();

// Removing the first element
let removedFirst = fruits.shift();
```

```
// Finding index of an element
let indexOfOrange = fruits.indexOf("orange");

// Iterating using a for loop
for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
  }

// Slicing an array
let citrus = fruits.slice(1, 3);
console.log(citrus);
```

```javascript
// Splicing an array (modifying original array)
let removed = fruits.splice(1, 2, "pear", "melon");
// Concatenating arrays
let moreFruits = ["grapefruit", "pineapple"];
let allFruits = fruits.concat(moreFruits);
```

Summary on Arrays

- Arrays in JavaScript are ordered lists of values.
- Array elements can be of any data type.
- Arrays have zero-based indices, meaning the first element is at index 0.
- Various methods are available for adding, removing, and manipulating array elements.
- Iterating through arrays can be done using traditional for loops or array methods like forEach.
- Slicing extracts a portion of an array without modifying the original, while splicing allows modification by adding or removing elements.
- Concatenation combines multiple arrays into a new array.

Simple function

```
function add(a, b) {
  return a + b;
}

// Invocation
const sum = add(5, 3);
console.log('Function Declaration - Sum:', sum);
```

Function Expression

```
const multiply = function(a, b) {
  return a * b;
};

// Invocation
const product = multiply(4, 6);
console.log('Function Expression - Product:', 
product);
```

Arrow Function

```
const subtract = (a, b) => a - b;

// Invocation
const difference = subtract(8, 3);
console.log('Arrow Function - Difference:',
difference);
```

Error Handling

- Error handling is essential for gracefully managing unexpected issues in your code.
- It **prevents your program from crashing** and provides a way to handle errors in a controlled manner.

Example Program

```javascript
try {
  // Code that may throw an error
  throw new Error("An error");
} catch (error) {
  // Code to handle the error
  console.error(error.message);
} finally {
  // Code that will run regardless of whether there was an error
  console.log("Cleanup code");
}
```

- **try Block:**

The try block contains the code that might throw an error. In this case, it explicitly throws a new Error object with the message "An error.“

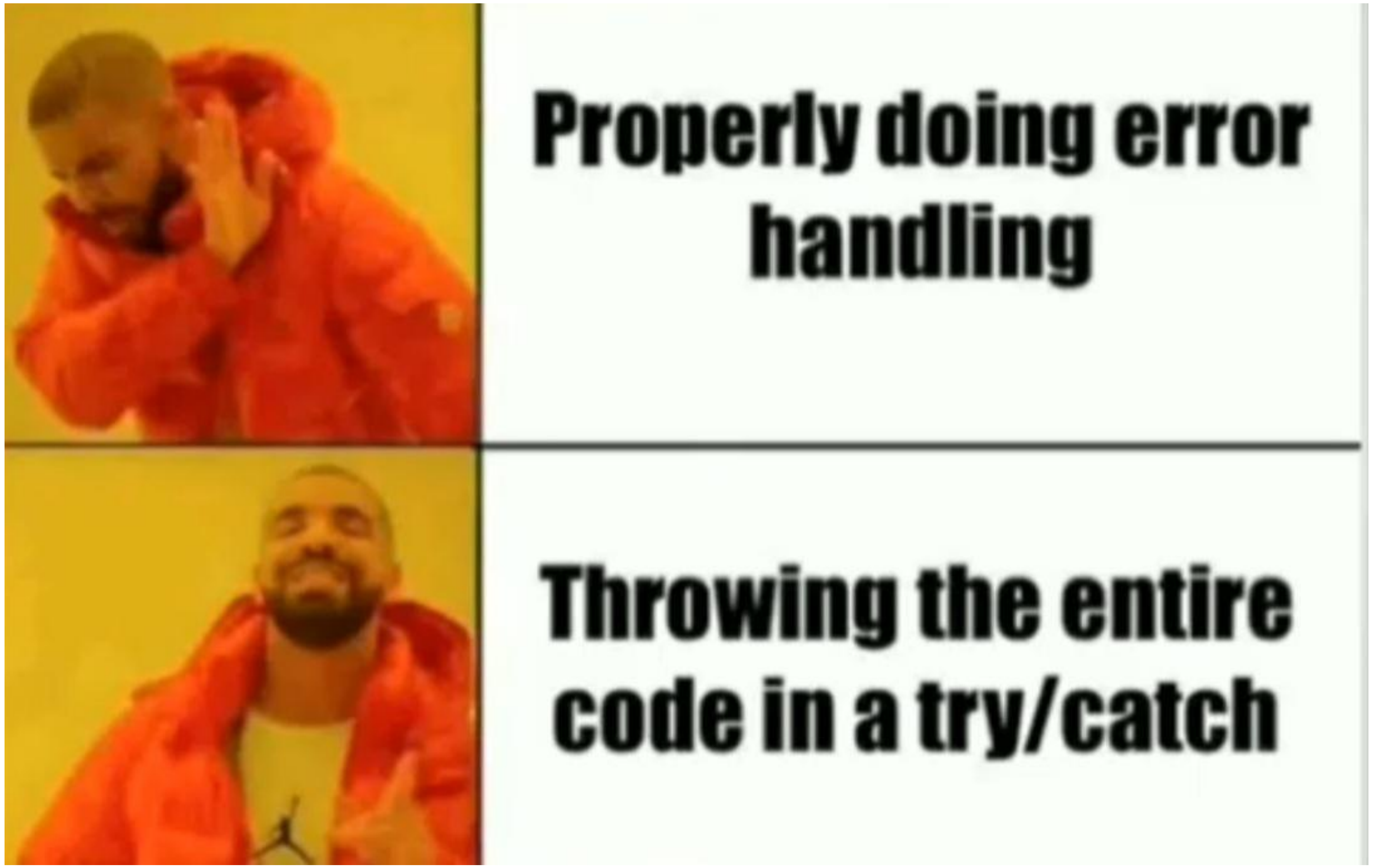
- **catch Block:**

If an error occurs within the try block, the control is transferred to the corresponding catch block.

The catch block takes an error parameter, which represents the caught error.

- **finally Block:**

The finally block contains code that will execute regardless of whether an error occurred or not. This block is useful for cleanup operations or tasks that must be performed regardless of the program's state.


Properly doing error handling
Throwing the entire code in a try/catch

Timers

- Timers in JavaScript are mechanisms that allow you to execute code after a specified delay or at regular intervals.
- There are three main timer functions in JavaScript: setTimeout, setInterval, and clearInterval.

setTimeout Function

- Executes a function or a piece of code once after a specified delay.

```
const delayedFunction = () => {
  console.log('Delayed function executed!');
};

setTimeout(delayedFunction,2000); // Executes after
2 seconds
```

-

setInterval Function

- Executes a function or a piece of code repeatedly at a specified interval.

```
const repeatedFunction = () => {
  console.log('Repeated function executed!');
};

const intervalId = setInterval(repeatedFunction, 
1000); // Executes every 1 second
```

-

In case you want it to stop after some time

```
// Function to be executed at intervals
function repeatedFunction() {
  console.log('Executing repeated function...');
}

// Set an interval (execute repeatedFunction every 1000 milliseconds)
const intervalId = setInterval(repeatedFunction, 1000);

// After some time (e.g., 5000 milliseconds), stop the interval
setTimeout(() => {
  clearInterval(intervalId); // Stop the interval
  console.log('Interval stopped.');
}, 5000);
```

Callbacks

- Callbacks are functions that are **passed as arguments to other functions and are executed after the completion of an asynchronous operation or a certain event**.
- Commonly used in scenarios like handling asynchronous tasks, event handling, and timer functions.
- **Reference:**https://builtin.com/software-engineering-perspectives/callback-function

Synchronous vs Asynchronous Callback Functions

- **Synchronous callback functions execute instantly, but asynchronous callback functions execute at a later time.**
- The sequence in which synchronous callbacks are defined is followed when they are executed. This implies that the first callback will be processed ahead of the second callback if a function calls two synchronous callbacks.
- The **order in which asynchronous callbacks are executed is random**. You can't predict when the asynchronous callback will be invoked as a result.

Synchronous Callback Function Example

```
function parentFunction(name, callback){
    callback();
    console.log("Hey "+name);
}

function randomFunction(){
    console.log("Hey I am callbackfunction");
}

parentFunction("Random String",randomFunction);
```

Decode this code

- In this example, we have two functions: **parentFunction** and **randomFunction**. The **parentFunction** takes two arguments: **name (a string)** and **callback (a function).** It executes the callback function and then logs a greeting message using the provided name.
- The **randomFunction** simply logs a “Hey I am a callback function” message.

Asynchronous Callback Function Example

```
function parentFunction(name, callback){
    setTimeout(callback,1000);
    console.log("Hey "+name);
}

function randomFunction(){
    console.log("Hey I am callbackfunction");
}

parentFunction("Random String",randomFunction);
```

Same code can also be written in this way with anonymous function

```
function parentFunction(name, callback){
    setTimeout(callback,1000);
    console.log("Hey "+name);
}

parentFunction("Random String",function(){
    console.log("Hey I am a callback Function");
});
```

We can make it more shorter…using arrow functions.

```
const parentFunction=(name, callback)=>{
    setTimeout(callback,1000);
    console.log("Hey "+name);
}

parentFunction("Random String",function(){
    console.log("Hey I am a callback Function");
});
```

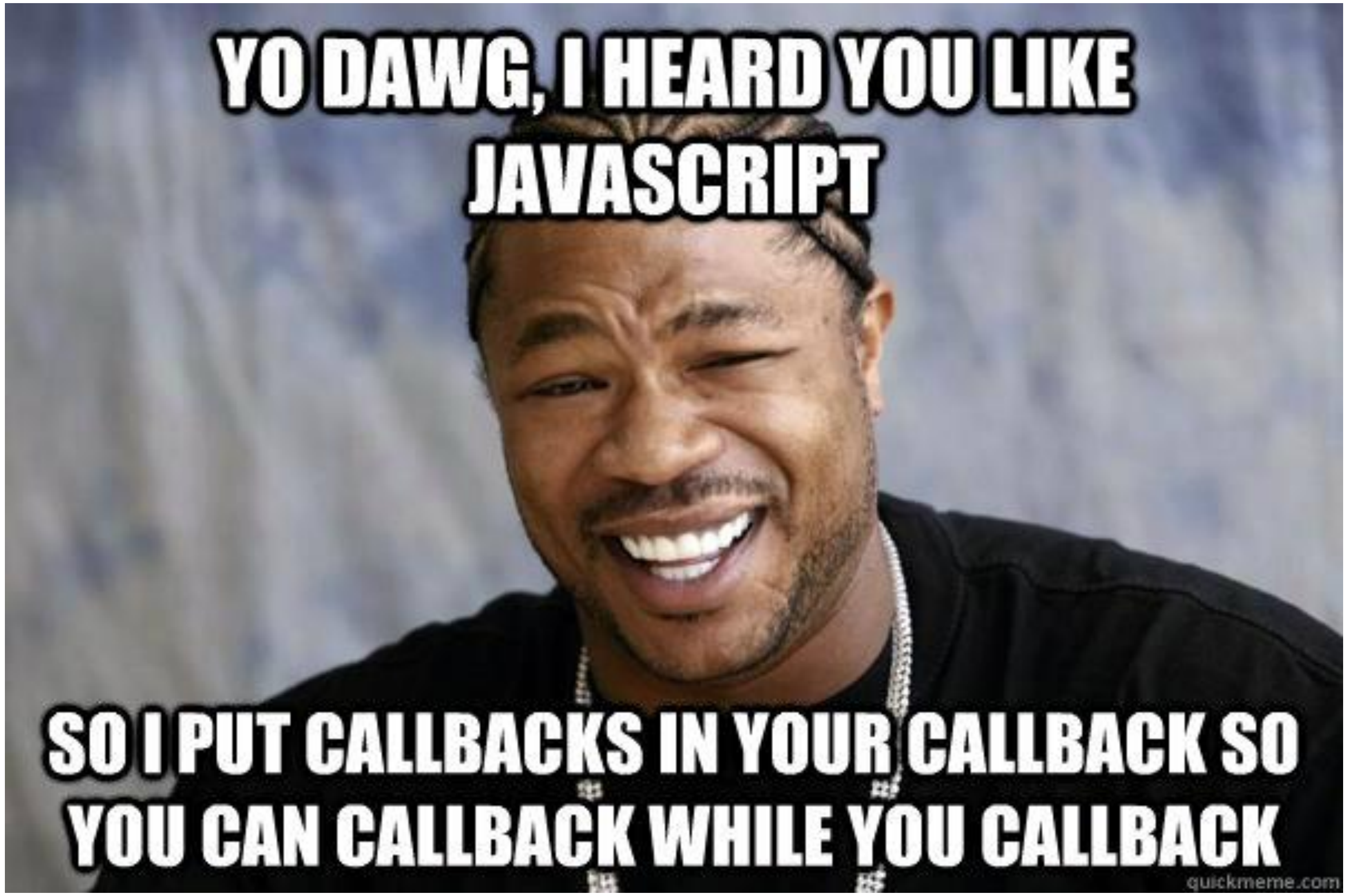
More shorter…..

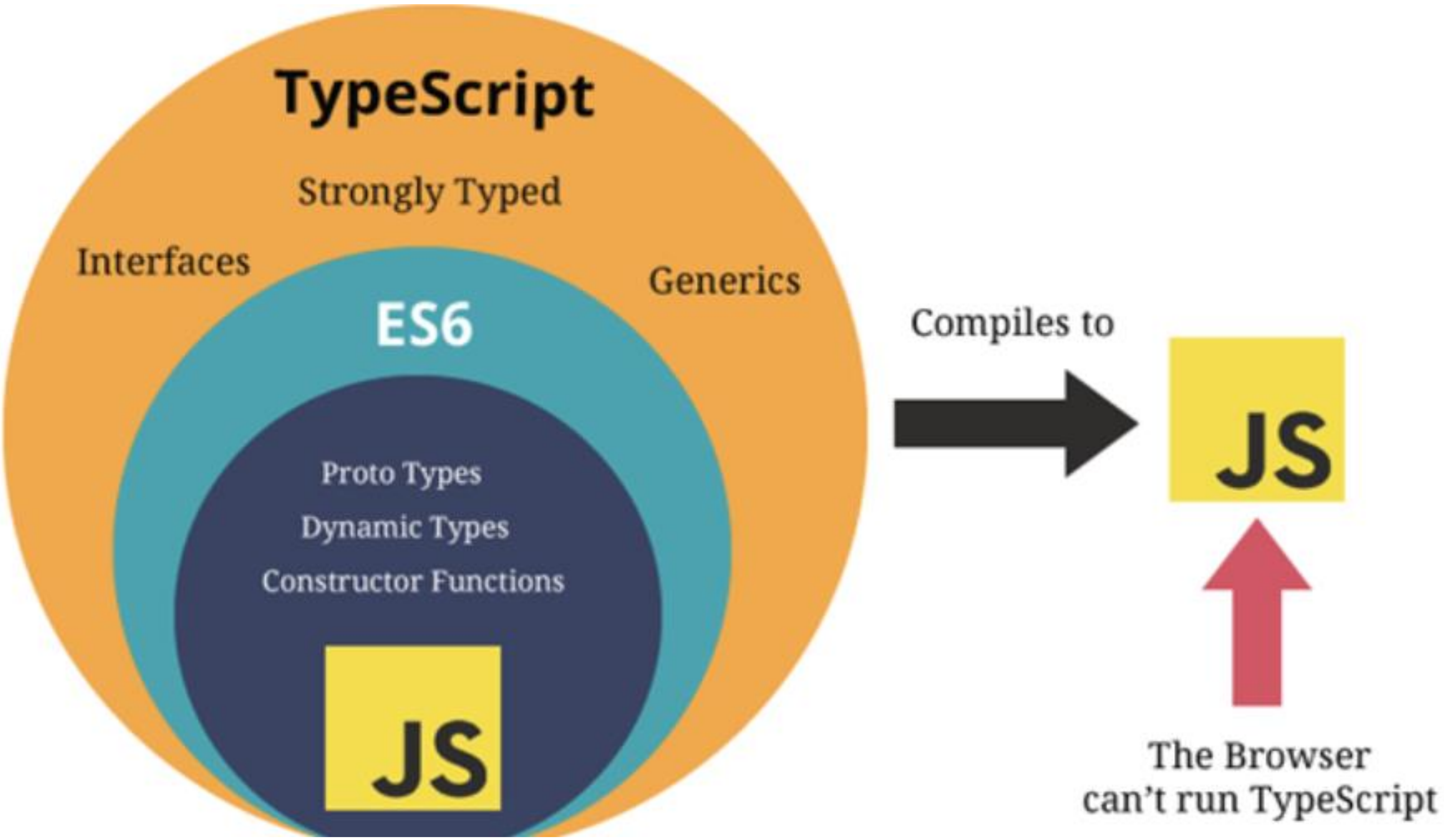
```
const parentFunction = (name, callback) =>
(setTimeout(callback, 1000), console.log("Hey "
+name));

parentFunction("Random String", () =>
console.log("Hey I am a callback Function"));
```

I'M A CALLBACK
I'M ALSO, BUT ANONYMOUS
imgflip.com

Importance of Callback

- **You can use them to send asynchronous API calls**. Many APIs are asynchronous, which means they don't immediately return a value. They instead give back a promise or a callback method. When the API call is finished, the callback function is invoked.
- **You can use them to enhance your code's performance**. Callbacks enable you to multitask while an asynchronous action is running, which can help your code execute faster.
- **Callback functions can be used to manage the flow of asynchronous operations, preventing the infamous "callback hell**." This situation is where deeply nested callbacks make code hard to read and maintain.
- **Higher-order functions**, or functions that can take other functions as inputs or return other functions as values, are based on the concept of callbacks. This functional programming pattern is effective.


YO DAWG, I HEARD YOU LIKE JAVASCRIPT
SO I PUT CALLBACKS IN YOUR CALLBACK SO YOU CAN CALLBACK WHILE YOU CALLBACK
quickmeme.com


TypeScript
Strongly Typed
Interfaces
Generics
ES6
Proto Types
Dynamic Types
Constructor Functions
JS
Compiles to
JS
The Browser
can't run TypeScript