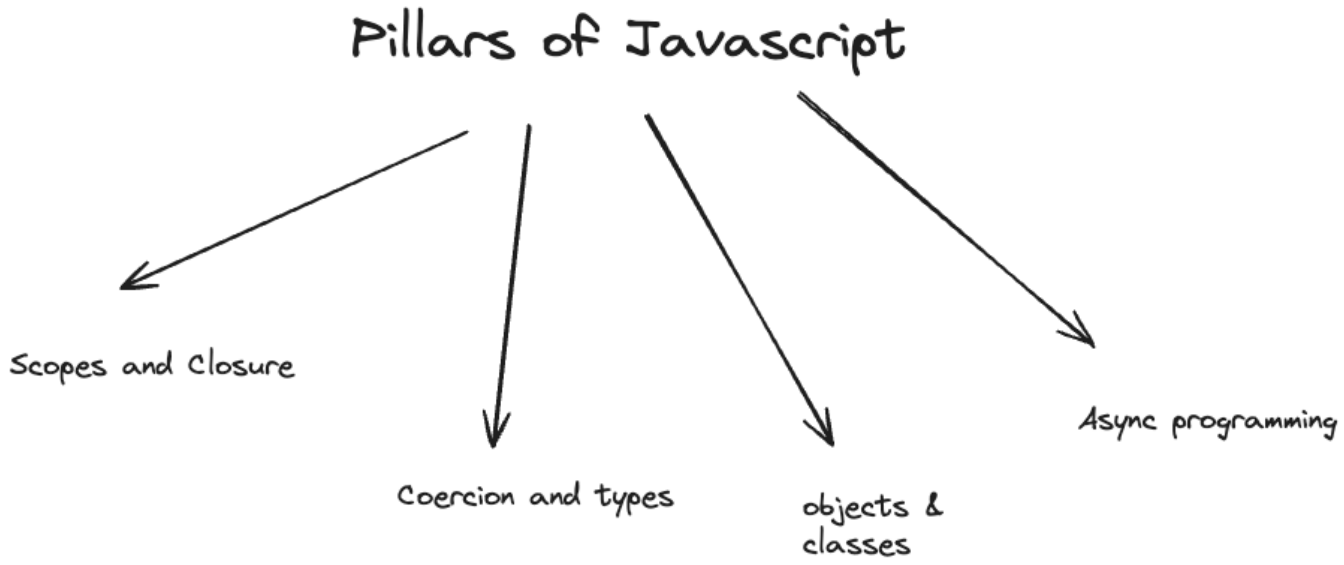
Javascript Pillars

Mainly we have 4 pillars

- Scopes and Closures
- Coercion and types
- Async Programming
- Objects and classes



Scopes

Scopes as a word is closely related to vision i.e. what particular part you can see and what you cannot see.

In Javascript, we use the concept of scopes to figure out where a variable or function is accessible / visible.

But how exactly the scoping mechanism in JS works ?

Note:

Scoping mechanism in JS is very very different than other languages (Java , C++, python etc). So don't mix the concepts of JS with other languages.

To understand the whole scoping mechanism, we need to understand the concept of compilation and interpretation in programming languages.

Most of the languages are divided into two categories:

- Compiled language
- Interpreted language

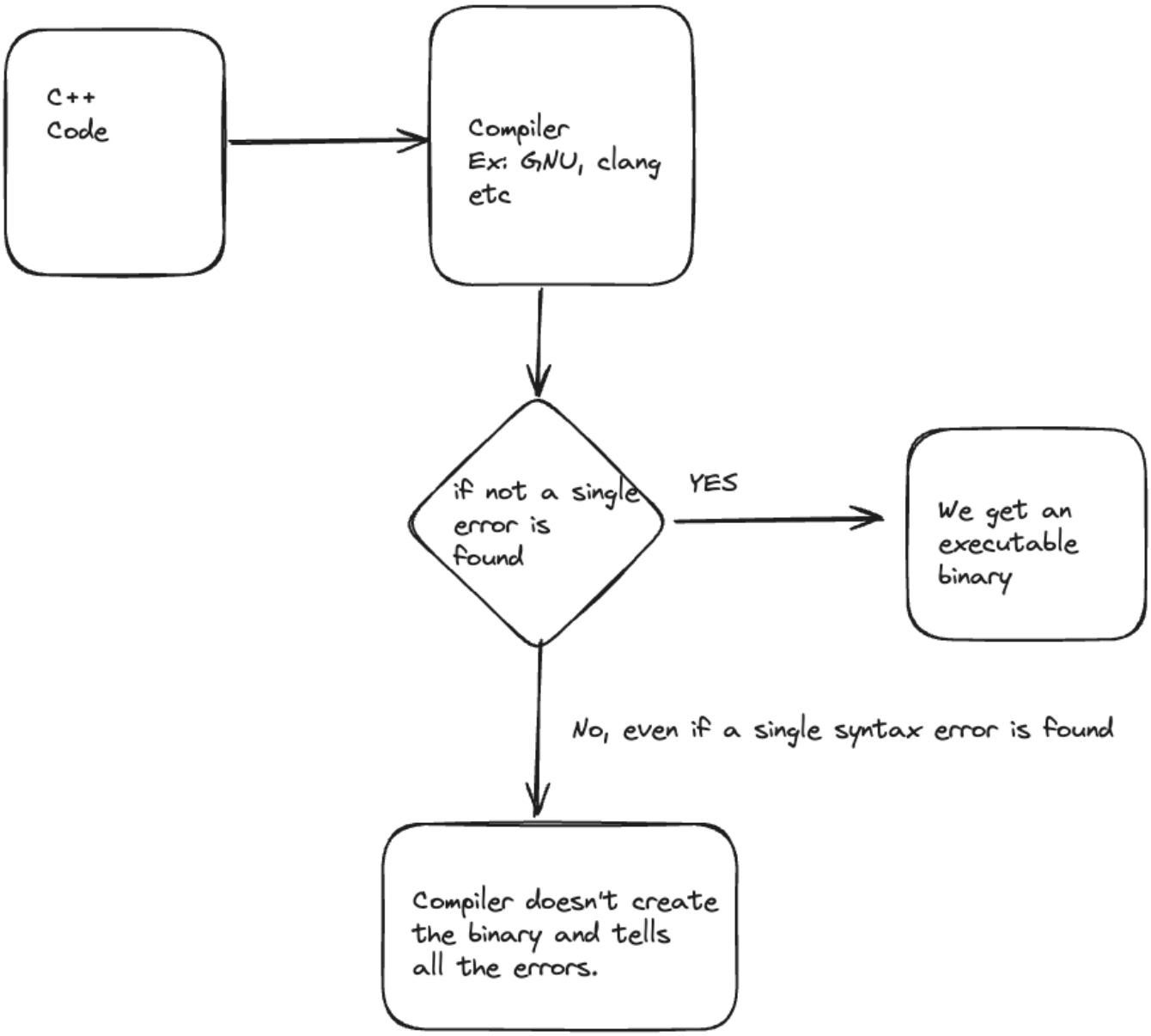
The question that should come to your mind is, whether JS is compiled or interpreted ?
To understand this let's take a deeper dive into compilation and interpretation.

Compiled Languages

Example of compiled languages are: **C** , **C++** etc.
To run the code of any compiled programming language we need to use another software called as Compiler.

Compiler takes the whole code, analyses it for errors, if there are no errors then it will give us an executable binary, but if there is any single error, then nothing will be added to the executable, in fact no executable will be made and compiler will throw the errors which are present in the code.
Interesting fact is that all the errors are told at once.
If let's say the first 100 lines of code are correct and the 101th line has an issue, still nothing will

be executed.



Interpreted Languages

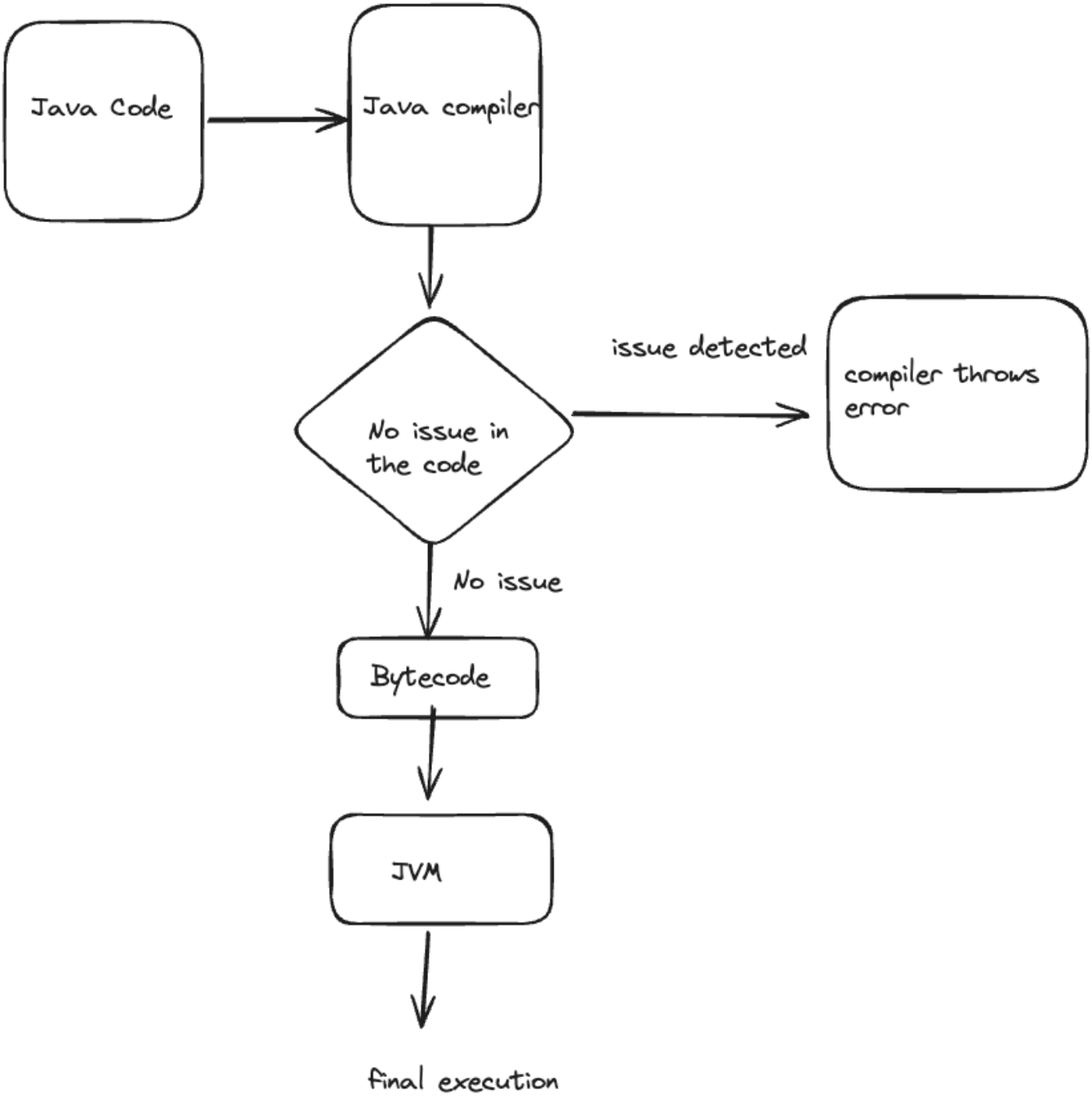
Pure interpreted languages exist for example: Bash
These are those languages, which execute our code directly, without reading / analysing the whole code prior. They execute code line by line, if any line has an error, then everything before it gets properly executed and the moment we detect the first error, execution stops.
If there is more code, after the error, nothing is executed post we encounter the error.

```
echo "hello world"
ls -l
excho 'hello world"";
echo "hello world"
```

```
[Running] bash "/Users/sanketsingh/Developer/Javascript-Intermediate/Scopes/test.sh"
hello world
total 0
-rw-r--r--  1 sanketsingh  staff   0 11 Jul 20:37 README.md
drwxr-xr-x  3 sanketsingh  staff  96 11 Jul 21:32 Scopes
/Users/sanketsingh/Developer/Javascript-Intermediate/Scopes/test.sh: line 5: unexpected EOF while
looking for matching `''
/Users/sanketsingh/Developer/Javascript-Intermediate/Scopes/test.sh: line 8: syntax error: unexpected
end of file
```

Hybrid Languages

There is kind of like a third category of languages which uses both compilation and interpretation for the final execution of code. Ex: **Java**, **JS**, **Python**, etc



Nature of JS

JS combines the process of compilation and interpretation. A lot of people think it is interpreted only but that's not true. How ?

```
console.log("wow wow");
function fun() {
    le r = uy;
}
```

The above piece of throws an error without executing `console.log` and prints nothing on the console and just says the error altogether.
This shows that JS is somehow analysing the whole prior and then only starting the execution.

Actually every JS code is executed in 2 phases:

- **Phase 1** : Compilation and Scope Resolution phase
- **Phase 2**: Interpretation or execution phase

What do we mean by scope resolution ?

When we say scope resolution, we mean that we need to allocate the scope / visibility to each and every variable altogether.

Types of scopes in JS

In JS, there are multiple type of scopes available:

- Global Scope
- Function Scope
- Block Scope

Global Scope

As the name suggest, global scope refers to the scope where the variable is gonna be accessible from anywhere in the program.
Global scoped variables are accessible throughout your program, be it in a function, for loop, while loop, if-else anything.

```
let x = 10;

function fun() {

    console.log(x);
}

function gun() {
    for(let i = 0; i < x; i++) {}
}
```

```
fun();
gun();

console.log("value of x is", x);
```

In the above piece of code x has been defined outside any function, it's available in the common area or global area of the code, hence it has global scope.

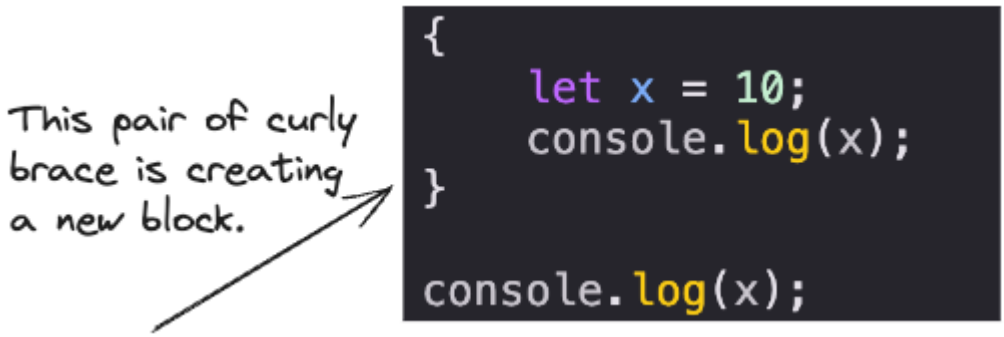
If we would have defined x inside the function fun, then we would not have been able to access x outside the function. If we define x inside the function fun, then it will be having a new scope called as `function scope`.

Function scope

Function scope means, visibility of a variable is only inside the function where variable has been defined, you cannot access it outside the function.

```
function fun() {

    let x = 10;

    console.log(x);
}

function gun() {
    for(let i = 0; i < x; i++) {} // x is not visible here
}

fun();
gun();

console.log("value of x is", x); // x is not visible here also
```

Block scope



Wherever we define a pair of curly braces, may be with if-else, while loop, for loop etc or may be without anything, it creates something called a block.
What is a block ? **Block is a collection of valid JS instructions enclosed in a pair of curly braces.**

```
if(10 > 20) {
        // This is the block of if
    // any code written here is meant to be inside the block of if
} else {
        // This is the block of else
    // any code written here is meant to be inside the block of else
}
```

Any variable defined inside a block has a block scope, which is the third kind of scope.
In the below code, we won't be able to access x, because it is defined inside a block, and due to that it has a block scope, hence it is not visible outside the block.

```
{
    let x = 10;
    console.log(x); // here x is visble because we try to access it within
the block
}

console.log(x); // here x is not visible, hence it throws an error
```

So is the scope of the variable defined only by where is it declared ?

No, scope of the variable is also decided by how it is declared. So, how a variable is declared ? We have three ways:

- var -> var helps us to declare function scoped or global scoped variables.
- let -> let helps us to initialise block scoped variables

- const -> const also helps us to initialise block scoped variables
  These 3 keyword helps us to declare variables. And they also help us to define the scope of the variables.

You might be thinking that I gave all the examples with let only, so those were only for demo purpose to show you the placement of global, function and block scoped variables.

Any variable is used only in two ways:

- RHS -> i.e. when we consume the variable
- LHS -> i.e. when we assign value or declare the variable
  For example:

```
var x = 99; // LHS - we are assigning it a value
console.log(x); // RHS - we are consuming the value of the variable
```

Lexical Scoping / Lexical Parsing

JS does scope resolution using lexical scoping mechanism. It is also called as static scoping. In lexical scoping we allocate scopes to the variables during compile time. SO in JS, values to the variables are allocated in phase two i.e execution phase but scope of the variable is decided during phase 1.

var

var helps us to define global and function scoped variables. We cannot make a block scoped variable with var.
Now a question can be, how is function scope different than block scope ?
A variable having function scope has a special property, that it can be defined anywhere in the function but will be still accessible through the whole function.

```
function fun() {
    console.log("the value of x here is",x);
    var x = 10;
    console.log("the value of x here is",x);


}

fun();
```

Here x is being used before declaration, and this code works, because x has function scope due to var hence x is visible through the code.

let

Both let and const work in a similar way when we talk about the scope perspective. Whenever we declare any variable with `let` then the variable will get the scope of the nearest block. If the nearest block is a `for` loop block then variable gets scope of the for loop, if the nearest block is an `else` block then variable gets the block scope of else or if the nearest block is a `function` block then variable gets a block scope inside function.

What is the meaning of block scope inside function ? How is it different from var's function scope ?

When we say function scope then it means that no matter where we declare the variable in the function it's available throughout the function.

But that's not the case with a block scope inside function. In a block scope variables are only accessible after they are declared, we cannot access them before their declaration.

```
function fun() {

    console.log(x);
    var x = 10;
    console.log(x);
}
fun();
```

In the above piece of code, x will be printing undefined and then 10, because x is initialised by `var` hence it gets a function scope so it will be available everywhere in the function, no matter when we declare it.

```
function fun() {

    console.log(x); // this is a tdz area
    let x = 10;
    console.log(x);
}
fun();
```

The above code throws the following error:

```
ReferenceError: Cannot access 'x' before initialization
```

This is because in a block scoped variable we cannot access the variable before the initialisation. This makes block scoped variable special. And the region before the declaration of the block scoped variable is called as `Temporal Dead Zone`.

Temporal Dead Zone (TDZ)

This is the region before the declaration of a block scoped variable. A variable declared with let, const, or class is said to be in a "temporal dead zone" (TDZ) from the start of the block until code execution reaches the place where the variable is declared and initialized.

More facts on let

`let` doesn't allow redeclaration of a variable, and this will be called out in the phase 1.

```
let x = 10; // block scope in the global area
console.log(x);
let x = 20; // error in the phase 1
console.log(x);
```

```
var x = 10; // global
console.log(x);
let x = 20; // error
console.log(x);
```

Hoisting

Hoisting is a consequence of the scoping mechanism of JS. Because of the fact that JS executed your code in 2 phases, a lot of variables are already known during the phase 1, and then they are accessed in phase 2. So it looks like to a lot of people that JS knows about a few variables before their declaration. And indeed it is true, because during phase 1 all the formal declarations are read, so JS indeed knows about the variables before execution phase. And this mechanism of knowing variable before their declaration line is termed as **Hoisting**. This word is

not an actual word present in the official ecmascript docs. The JS community came up with this one word definition for this consequence.