**JavaScript Notes**

Everything in JavaScript happens inside an **Execution Context.**

**Execution Context** like a big box and has 2 components :-

1. Memory – All the variables and functions are stored (also known as variable environment)
2. Code – Code is executed 1 line at a time (also known as thread of execution)

JS is Synchronous single-threaded language.

Single-threaded – one command at a time

Synchronous single-threaded – one command at a time and in a specific order

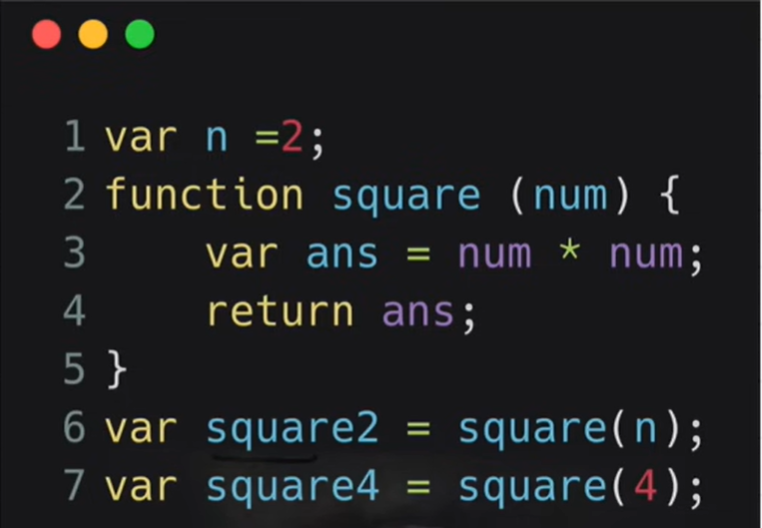
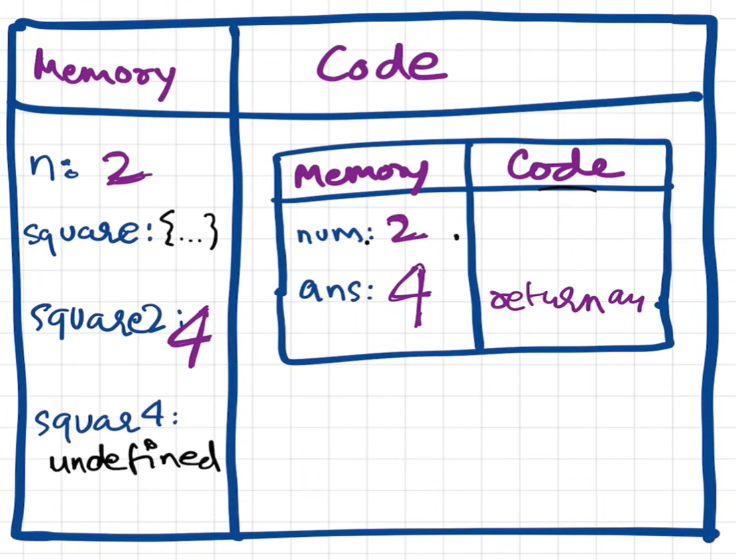
* **What happens when you run JS code?**
* When we run JS code and execution context is created.
* Execution context created in 2 phases:

1. Memory Creation Phase

* JS will allocate memory to all the variables and functions
* For Variables - allocates a special value ‘Undefined’
* For functions – whole code copied will be there in memory

1. Code Execution Phases

* Again here also JS runs through the whole JS program line by line and execute the code and started assigning actual values to the memory variables and functions.
* Whenever new function invokes, a brand-new execution context is created
* Then again two phases happen for that particular function here for square function and return value where was the function invoked and execution context will be deleted
* Whatever calculation are there in function will be done in **Code** component



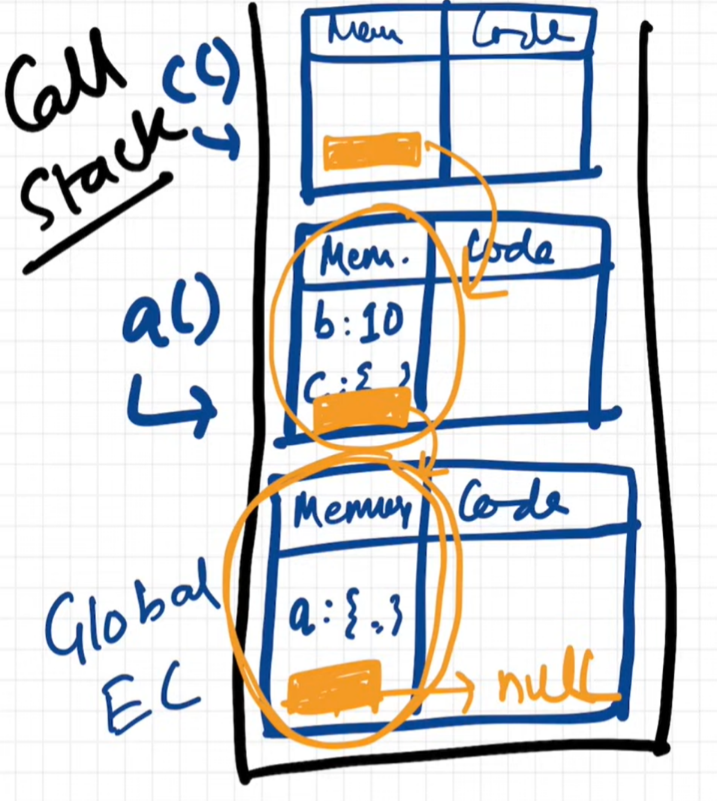
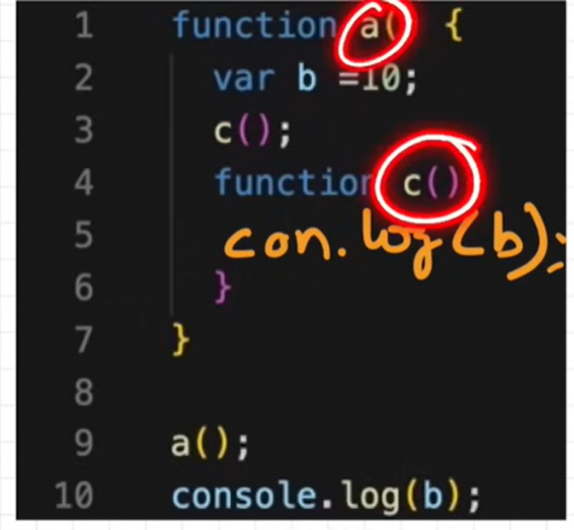
* When there are lot functions inside function then it will create context inside context
* JS handles this very beautifully by using **Call Sack.** For managing the execution contexts.
* JS has it’s own call stack, In the bottom of the stack we have Global Execution Context (GEC)
* Whenever EC created it pushed into the stack and whenever deleted it will popped out of the stack.
* Call stack maintains the order of execution of execution contexts:
* Call Stack is also known as
* Execution Context Stack
* Program Stack
* Control Stack
* Runtime Stack
* Machine Stack
* **Hoisting in JS & Execution Context :**
* Go to Sources for debugging the code.
* In JavaScript, **hoisting** refers to the built-in behaviour of the language through which declarations of functions, variables, and classes are moved to the top of their scope – all before code execution. In turn, this allows us to use functions, variables, and classes before they are declared.

- In the global level **this** points to **window** object.

- **Window** is global object created along with the GEC.

- this === window

* JavaScript is a **loosely typed** language, meaning you don't have to specify what type of information will be stored in a variable in advance. JavaScript automatically types a variable based on what kind of information you assign to it
* **Scope Chain – Directly related to lexical environment.**
* Scope in JS determines the accessibility of variables and functions at various parts of one’s code or program.
* JS engine uses scopes to find out the exact location or accessibility of variables and that particular process is known as Scope Chain.



**What is Temporal Dead Zone?**

* The **Temporal Dead Zone** refers to the period in the execution of a JavaScript program where a variable exists but cannot be accessed. This occurs during the time between the creation of a variable and the point where it is declared.

**Are let & const declarations hoisted?** **Yes**

* let and const are hoisted but not in global scope they allocated in different memory space.
* We can not access these variables before assigning value to them.

Syntax Error vs Reference Error vs Type Error

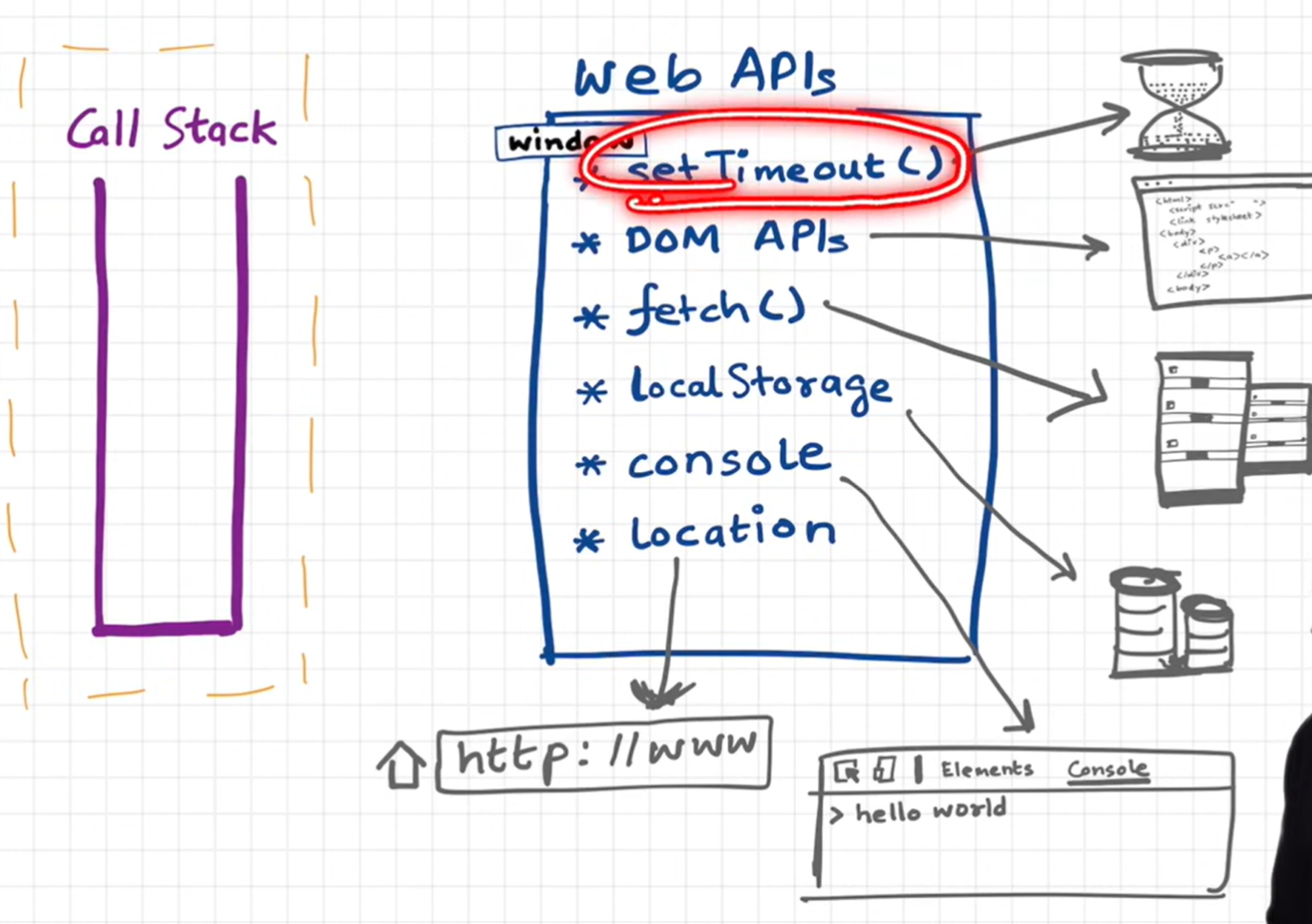
* **Closures**
* A closure is the combination of a function bundled toge ther (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives a function access to its outer scope.
* Uses:
* Module Design Pattern
* Currying
* Functions like once
* Memoize
* Maintaining state in async world
* setTimeouts
* Iterators
* **First-class Function**

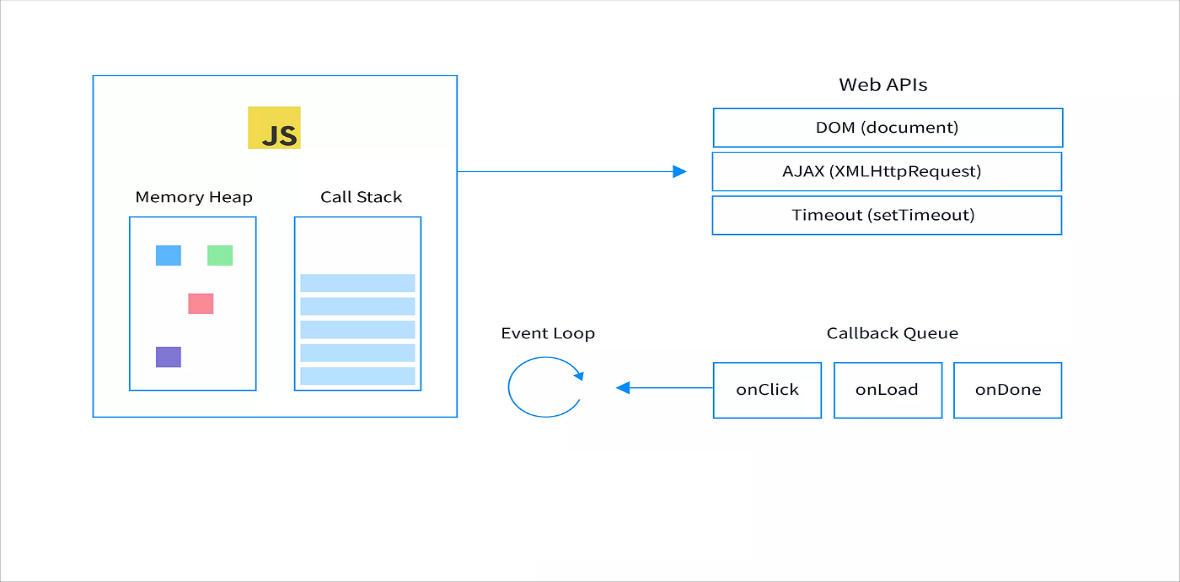
A function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.

* **Callbacks:**
* A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
* **Event Loop**

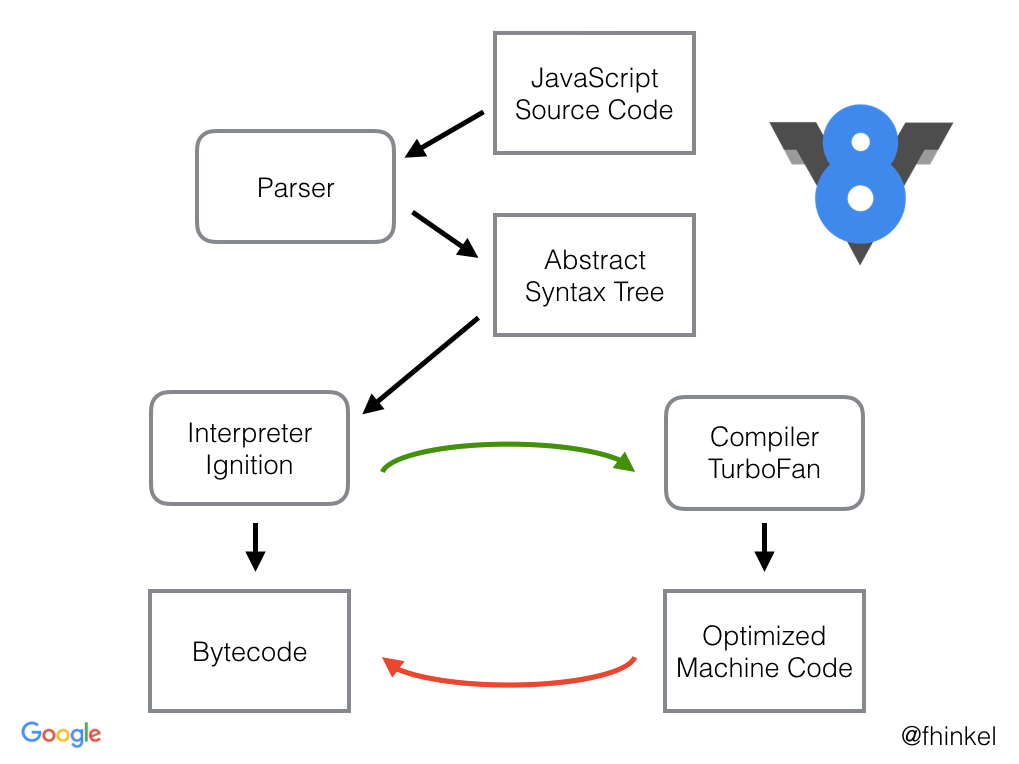
This diagram illustrates the event loop’s role in JavaScript’s execution model.

1. **Memory Heap and Call Stack**: JavaScript’s memory heap stores variables and objects, while the call stack keeps track of function calls. The call stack executes tasks sequentially, one by one.
2. **Web APIs**: When JavaScript encounters asynchronous tasks (like setTimeout, AJAX requests, or DOM events), they’re offloaded to Web APIs. These APIs handle the tasks independently without blocking the call stack.
3. **Callback Queue**: Once an asynchronous task completes, its callback function (like onClick, onLoad, or onDone) is moved to the callback queue. The callback queue stores all callbacks waiting to be executed.
4. **Event Loop**: The event loop constantly checks if the call stack is empty. If it is, the event loop takes the first task from the callback queue and pushes it onto the call stack.
5. **Execution Order**: This process ensures that synchronous code runs first, and asynchronous callbacks are only executed when the main stack is clear, allowing JavaScript to handle multiple tasks smoothly without blocking.





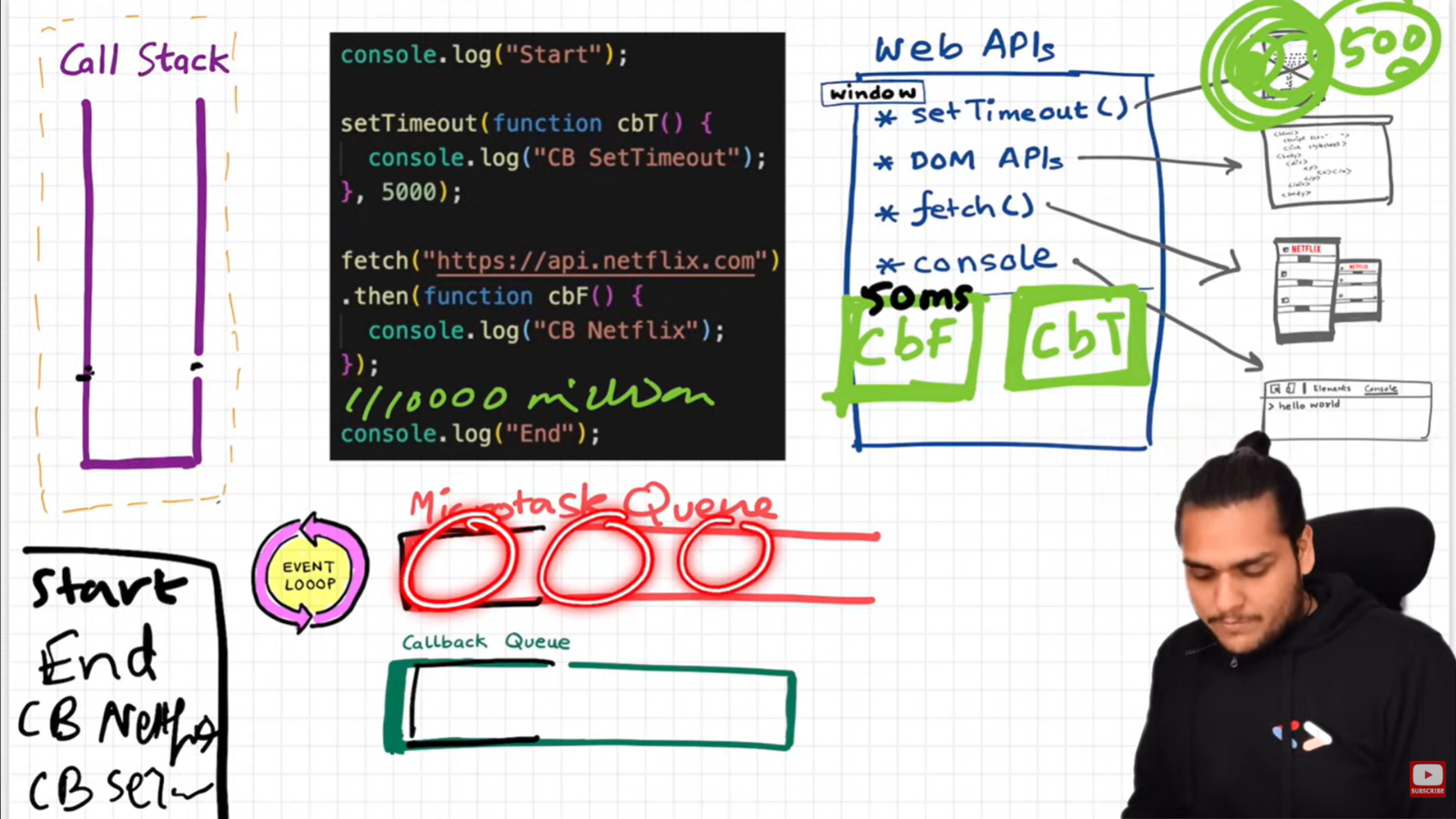
* **JavaScript Engine**



This diagram shows how the **V8 JavaScript engine** (used in Chrome and Node.js) processes and executes JavaScript code.

1. **JavaScript Source Code**: First, the raw JavaScript code you write is fed into the V8 engine.
2. **Parser**: The V8 engine starts by using a **parser** that reads the JavaScript code and converts it into an **Abstract Syntax Tree (AST)**. The AST is a structured representation of the code that the engine can understand and work with. [ astexplorer.net ]
3. **Interpreter (Ignition)**: The AST is then sent to the **Ignition interpreter**, which translates it into **bytecode**. Bytecode is a low-level representation of code, but it’s not fully optimized machine code yet. Ignition interprets and executes this bytecode immediately, allowing the code to run quickly without waiting for full optimization.
4. **Compiler (TurboFan)**: While the interpreter is running, the V8 engine’s **TurboFan compiler** identifies frequently used parts of the code (often called "hot code"). TurboFan compiles these hot parts into **optimized machine code**. This machine code is faster and more efficient because it’s tailored for the specific tasks the code performs.
5. **Optimization Cycle**: The V8 engine can switch between interpreting and optimizing as it gathers more information. If an optimized function starts running in ways the engine didn’t expect, it may “deoptimize” back to bytecode and recompile as needed.

* **Higher Order Functions: How to Use Higher Order Functions**
* A higher order function is a function that takes one or more functions as arguments, or returns a function as its result.
* There are several different types of higher order functions like map and reduce.
* When working with arrays, you can use the map(), reduce(), filter(), and sort() functions to manipulate and transform data in an array.
* When working with objects, you can use the Object.entries() function to create a new array from an object.
* When working with functions, you can use the compose() function to create complex functions from simpler ones.

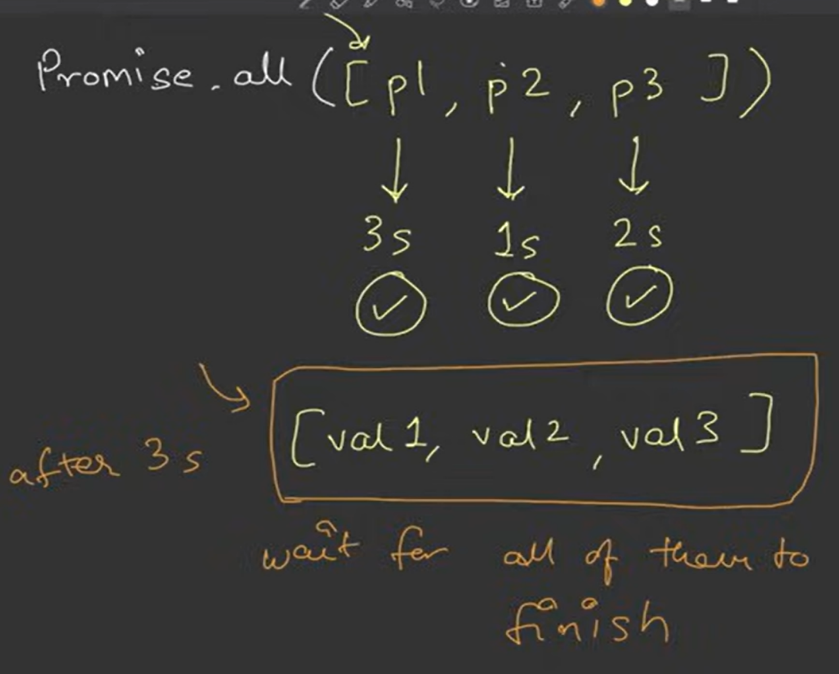
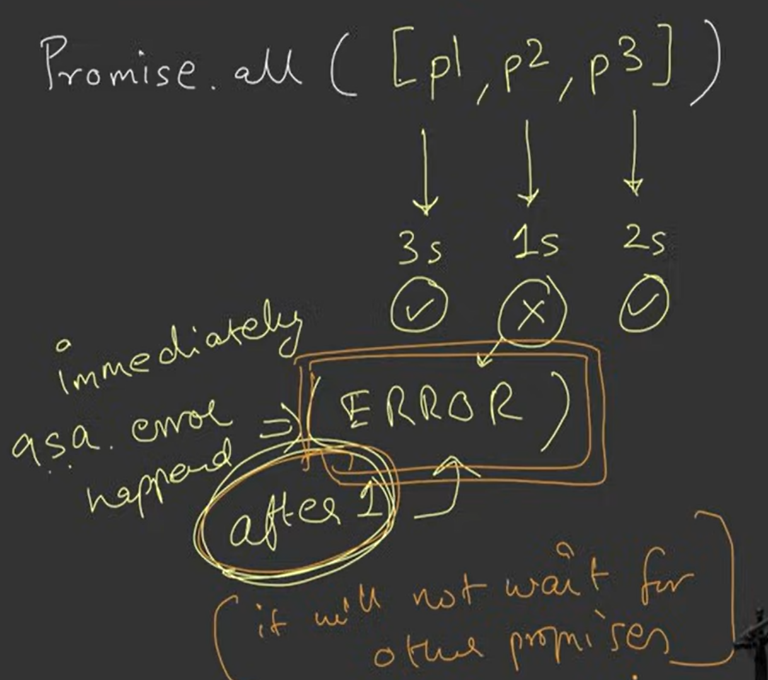


* **Callback Hell**
* Callback Hell is when multiple callbacks (functions that execute after another function completes) are nested within each other. This leads to code that’s hard to read and maintain because each callback relies on the previous one finishing. It often happens in asynchronous programming (like JavaScript), where you need to wait for one task to complete before starting the next.
* **Inversion of Control**
* Inversion of Control (IoC) is when a program hands control over part of its flow to a different entity (like a framework or library). Instead of dictating every step, you “invert” control and let the other entity decide when and how certain tasks should happen.

s

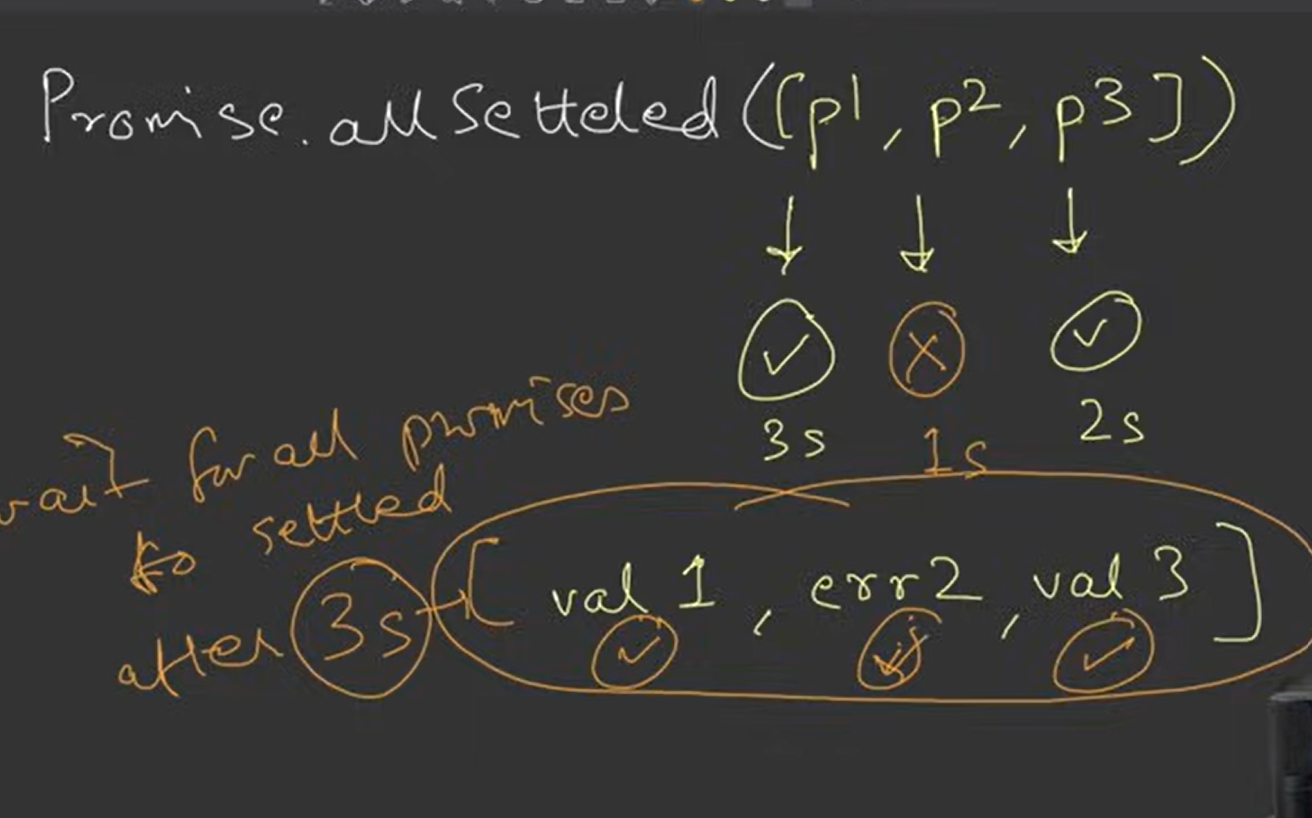
* **Promises**
* A Promise in JavaScript is an object representing the eventual result of an asynchronous operation. It can be either fulfilled (operation completed successfully), rejected (operation failed), or still pending (operation has not yet completed). Promises help avoid *callback hell* by chaining operations, making asynchronous code easier to read and manage.
* **Promise API**

1. **Promise.all()** –
   * The **Promise.all()** static method takes an iterable of promises as input and returns a single [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). This returned promise fulfills when all of the input's promises fulfill (including when an empty iterable is passed), with an array of the fulfillment values. It rejects when any of the input's promisesrejects, with this first rejection reason.



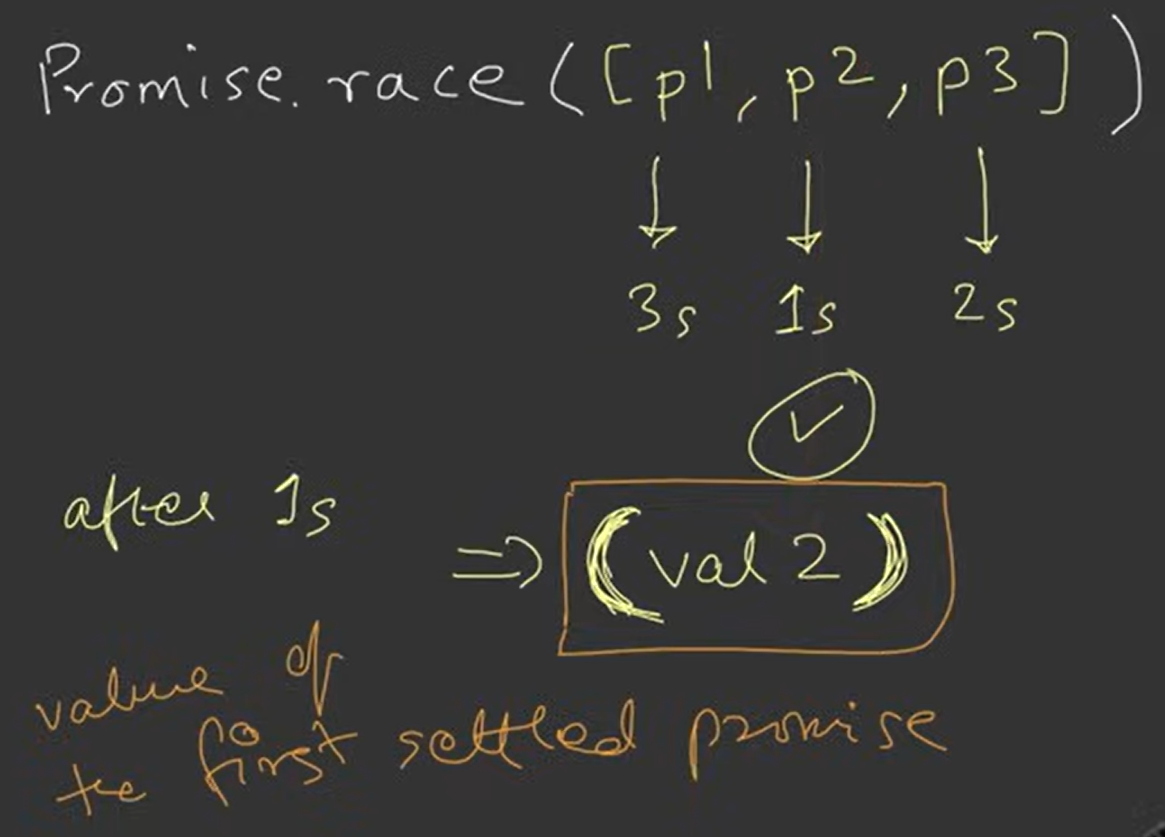
1. **Promise.allSettled()**

* The **Promise.allSettled()** static method takes an iterable of promises as input and returns a single [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). This returned promise fulfills when all of the input's promises settle (including when an empty iterable is passed), with an array of objects that describe the outcome of each promise.



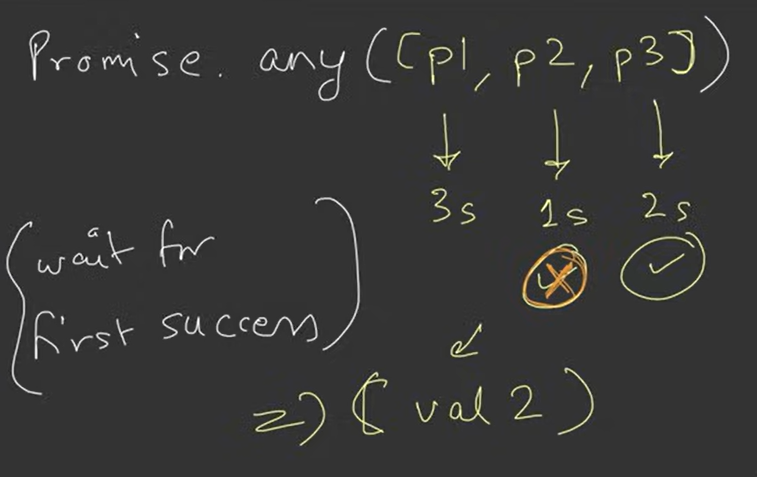
1. Promise.race()

* The **Promise.race()** static method takes an iterable of promises as input and returns a single [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). This returned promise settles with the eventual state of the first promise that settles.
* If First settled promise throw an error then it will show an error.



1. Promise.any()

* The **Promise.any()** static method takes an iterable of promises as input and returns a single [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). This returned promise fulfills when any of the input's promises fulfills, with this first fulfillment value. It rejects when all of the input's promises reject (including when an empty iterable is passed), with an [AggregateError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AggregateError) containing an array of rejection reasons.
* It is same as Promise.race but the only difference is it will wait for first promise to settle.
* If all promises get failed then it will show aggregateError (array of all the errors).



**Settled means –**

* + Resolve , Reject
  + Succes, Failure
  + Fulfilled, Rejected

**Async & Await :**

1. **async Function**:
   * When you add async before a function, it means this function will always return a promise. Even if you don’t explicitly return a promise, JavaScript automatically wraps the result in a promise for you.
2. **await Keyword**:
   * Inside an async function, you can use await to pause the execution until a promise is resolved.
   * It helps you write code that looks like it’s running step-by-step, even though it’s still asynchronous under the hood.