① Execution Content:

A JS Execution Content is a critical concept that defines the enviornment in which JS code is executed.

② Phases of Execution Content

There are total of 2 phases which program goes through, they are:

(1) Creation phase (2) Execution phase

(1) Creation phase:

In the Creation phase, the JS engine setup memory for variables & functions, assigning variables the initial value "undefined" & storing function defination as they are.

setup memory & environment

(2) Execution phase:

In the execution phase, the engine executes the code line by line, replacing "undefined" with actual values for variables, evaluating expressions, & running function calls by creating new execution content for each invoked function.

run the code & assign value

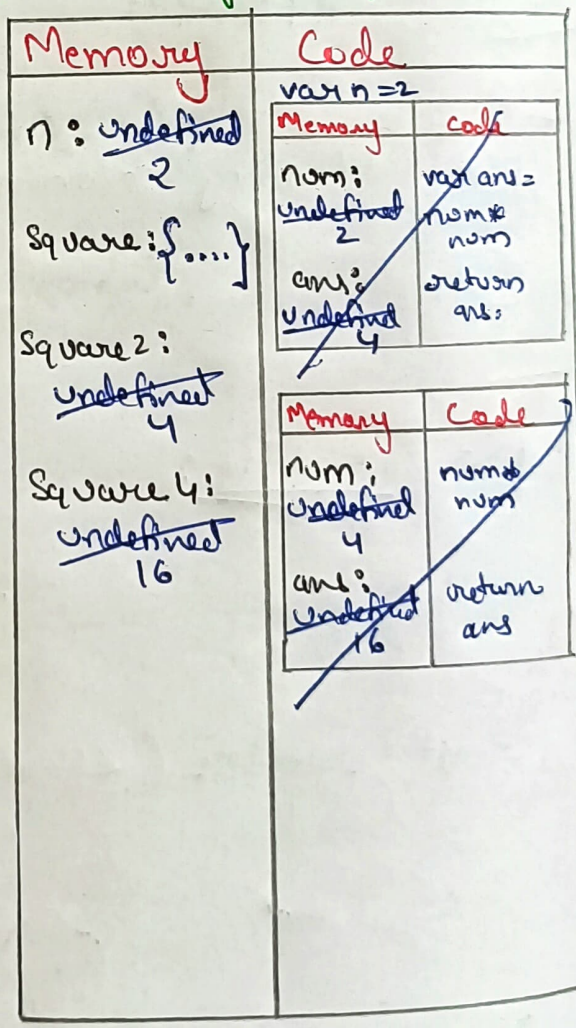
③ Execution Stack (Call Stack)

- JS uses stack to keep tracks of all execution content.
- Global content goes first; when a function is called, its content is placed on top.
- when the function finishes, that content is also poped out.

Example:

```
var n = 2;
function square(num) {
        var ans = num * num;
        return ans;
}
var square2 = square(n);
var square4 = square(4);
```

Call stack of Example code:



④ Hoisting

Hoisting in JS is a behaviour that allows variables & functions to be used before they are declared in the code.

Note: only declarations are hoisted, not initialization or assignments.

Hoisting works differently for different type of declarations:

1. "var" declaration
   variable declared with var are hoisted to the top of their scope & initialized with undefined.
2. "let" & "const" declaration
   variables declared with let & const are also hoisted, but they are not initialized.
   Accessing them before their declaration results in ReferenceError. They exist in the Temporal Dead zone (TDZ) until their declaration is evaluated.

How does Hoisting Work?

Hoisting in Javascript happens because during the compilation phase, before the code is executed, JS engine scans through your code & allocates memory for variable & function declarations.

⑤ Undefined V/s Not Defined

★ undefined

Meaning: The variable is declared but not yet assigned any value.

- In the memory creation phase (during hoisting), JS sets up space for the variable & assigns it a placeholder value: undefined. This acts like a temporary stand in until the variable is initialized in the code.
- So, undefined is basically JS way of saying, I know this variable exists, but no real value has been given yet. I'm just holding a placeholder for you.

★ not defined

Meaning: The variable was never declared in the scope.

- No placeholder, No memory slot: just doesn't exist in the environment record.
- Accessing it throws a Reference Error.

⑥ Scope

Scope defines where in your code you can access a variable.

Scope Chain

If JS doesn't find a variable in the current scope, it looks one level up, keeps going until it either finds it or reaches global scope. So, the chain of looking up is called the scope chain.

Eg.]

```
var a = 10;
function outer(){
    var b = 20;
    function inner(){
        var c = 30;
        console.log(a,b,c);
    }
    inner();
}
outer();
```

// Here, finds a in global
b in outer.
c in inner.

⑦ Lexical Environment

A Lexical Environment is basically:

- the local memory (variables (+) functions declared in that scope)
- A reference to the lexical environment of its parent.

⑧ Closures

A closure in JS is the combination of a function & the lexical environment within which that function was declared.

Closures allow a function to access variables from its outer scope even after that scope has finished executing.

Eg.]

```
function outer(){
    let x = 5;
    function inner(){
        console.log(x)
    }
    inner();
}
outer();
```

output: 5

Closures allows the inner function to remember & use the variable "x" from its outer function, even when outer function has already finished executing & no longer exists in the Call stack.

Why are closures useful? →

1) Data Privacy & Encapsulation
2) functions that 'remember' state (like counters)

(9) First-Class functions

In JS, functions are considered "first-class citizen", meaning they can be treated like any other variable.

- Functions can be assigned to variables.
- Functions can be passed as an arguments to other functions.
- Functions can be returned from other functions.

this ~~is~~ is very useful because it allows Callbacks, higher-order function, & writing more flexible & dynamic code.

(10) Callback function

A callback is just function passed as an argument to another function, & then called (invoked) inside that function after something happens.

Used for asynchronous operations (like reading files, API calls, timeout, etc).

Eg.)
```
Console.log("start");
setTimeout(function() {
    Console.log("This runs after 2 sec");
}, 2000);
Console.log("End");
```
CB fn.
HOF

output:
Start
End
This runs after 2 seconds.

(11) Event loop

The event loop is a crucial mechanism in JS that enables asynchronous, non-blocking operations within a single-threaded environment.

It continuously monitors the Call Stack & the callback queue along with Microtask queue to ensure that the tasks are being executed in specific order.

Asynchronous operations, such as setT... Timeout, network requests, or user interactions, dont block the main thread. Instead, they are handled by background WEBAPI's (in browsers) or the system kernel (in Node.js) once these operations are complete, their callbacks are placed in the Event Queue.

In short:
The Traffic Controller that makes sure your Synchronous code finishes first & then takes care of all the queued asyne tasks.

{ Microtask queue has higher priority than normal Callback queue }

(12) Higher order function

A function that takes another function as an argument (or) returns another function.

Higher-order function makes code shorter, smarter & more reusable.

(13) HOF V/s Callback function

```
SetTimeout ( function() {
        Console.log ("This shows");
    }, 2000);
```

- SetTimeout → Higher order function
- function() { ... } → Callback function

Think of a callback as a guest who arrives at the party & the Higher order function as the host who invites that guest in.

Note:

- Every callback function is passed to a higher order function
- But not every higher-order function necessarily takes only one callback: it can also return new functions.

(14) Array Methods

The array methods are super useful for transforming & processing data in a clean, and a functional way.

⇒ map()

- used to transform each element of an array & returns a new array.
- Does not change the original array.

```
Const arr = [1,2,3,4];
Const double = arr.map (
        function (num) {
            return num * 2;
        });
Console.log (doubled); // [2,4,6,8]
```

⇒ filter()

- used to filter elements based on a condition & return a new array with only those elements that pass the test.

```
Const arr = [1,2,3,4,5,6];
Const even = arr.filter ( function(num){
        return num % 2 === 0;
    });
Console.log(evens); // [2,4,6]
```

⇒ reduce()

- used to reduce an array to a single value (sum, products, max value) by applying a function repeatedly on each elem. & accumulating the result.

```
Const arr = [1,2,3,4];
Const sum = arr.reduce (
        function(acc, curr) {
            return acc + curr;
        }, 0);
Console.log(sum); // 10
```

(acc: index (0); curr: current value)

(18) Promise

History:

* Problems with callbacks:

1) Callback Hell:
When multiple callbacks are nested inside each other, it becomes messy & hard to read & maintain.

2) Inversion of code:
You lose control over the flow, which can lead to bugs or unexpected behavior if that code doesn't call back properly.

3) Error handling Complexity:
Each callback needs seperate error handling, making code messy.

Solution

* Promise

A promise is a javascript object representing the eventual completion or failure of an asynchronous operation.

Think of a promise as a placeholder for a value that will be available for future.

It has 3 states:
→ Pending
→ Fulfilled
→ Rejected

* How promises solve the problem that occurs while using callbacks.

1) Avoid Callback Hell:
Promises use chaining ".then()" instead of deep nesting.

2) Control Flow (fix inversion of code).
Promises return an object you can control & chain.
You decide what happens next instead of relying on external callback executions.

3) Centralized Error Handling

".catch()" handles errors in one place.

Example: Consuming promises.

```
doTask1()
    .then(function(result1) {
        return doTask2(result1);
    })
    .then(function(result2) {
        return doTask3(result2);
    })
    .catch(function(err) {
        console.error('Error:', err);
    });
```

Example: Creating Promises.

resolve — Reject

```
Const myPromise = new Promise(function(res, rej){
    if (//op. success_fol) {
        resolve('success');
    }
    else {
        reject('Error')
    }
});
```

(16) Async / Await

★ Async

- The "async" keyword is used to define a function that always returns a promise.
- This holds true irrespective of the actual return value of the function.

★ Await

- The "await" keyword can only be used inside "async" function.
- It effectively pauses the execution of async function & waits for the promise to settle (either resolve or reject).

★ Why is async/await is better than .then.catch for handling promises.

- Async/await is often considered better than ".then()" for writing asynchronous code because it provides a more readable & maintainable syntax that resembles synchronous code, & it simplifies error handling using "try/catch" blocks.

★ How does async/await works behind the scenes?

⇒ Behind the scenes, async/await works on top of promises.

⇒ When the function encounters await, its execution is Suspended, allowing other code to run.

⇒ The JS engine waits for the promise to resolve or reject.

⇒ once resolved, the function resumes execution from the next line after "await", using the resolved value.

Eg]:

```
async function getData() {
    const response = await fetch(...)
    const data = await response.json();
    console.log(data);
}
getData();
```

(17) Promise Methods

(i) Promise.all(iterable)

- Resolves with an array of results only when all promises are resolved.
- Reject immediately if any promise rejects, without waiting for others.

(ii) Promise.allSettled(iterable)

- Waits for all promise to settle (either fulfilled or rejected).
- Resolves with an array of results objects after all complete.
- Never rejects, it always resolves with results.

(iii) Promise.race(iterable)

- The returned promise settles (resolves or rejects) as soon as the first promise settles, with that first result.
- It does not wait for others to finish.

(iv) Promise.any(iterable)

- Resolves as soon as the first promise is fulfilled.
- Rejects only if all promises reject, with an "AggregateError" containing all rejection reasons.