# JavaScript Execution Behind the Scenes: Detailed Notes

These notes are derived from the provided transcript, which explains how JavaScript code executes internally. The lecture focuses on the Execution Context, hoisting, memory allocation, and differences between var, let, and const. I've structured them in a logical flow for clarity, with code examples, explanations, and key insights. The goal is to provide an in-depth understanding to help with interviews, debugging weird outputs, and errors.

## 1. Why Understand JavaScript Execution?

- JavaScript often produces "weird" outputs or errors due to its internal runtime behavior.
- Knowing how code runs "behind the scenes" helps explain these issues.
- This topic is crucial for interviews: Questions on hoisting, execution context, and variable behavior are common.
- Key Concept: JavaScript is single-threaded and synchronous by default, but its execution involves phases that make it seem asynchronous in some cases (e.g., due to hoisting).

## 2. Basic Code Example Used in the Lecture

Consider this simple code snippet to illustrate the concepts:

JavaScript

```
var a = 10;
var b = 20;

function addNumbers(num1, num2) {
  var sum = num1 + num2;
  return sum;
}

var sumResult1 = addNumbers(a, b);
var sumResult2 = addNumbers(4, 5);

console.log(sumResult1, sumResult2);  // Output: 30 9
```

- Intuitive thinking: Code runs line by line, assigning values and calling functions.

- Reality: It doesn't. JavaScript creates an **Execution Context** and processes in two phases.

## 3. Execution Context: The Core Concept

- **Definition**: When JavaScript code runs, an Execution Context (EC) is created. It's like a "box" or environment where the code is evaluated and executed.
- Every code execution (global or function) creates its own EC.
- **Global Execution Context (GEC)**: Created for the top-level code (outside any function). It's the base context.
- **Function Execution Context (FEC)**: Created when a function is invoked (called).
- ECs are managed in a **Call Stack** (LIFO - Last In, First Out): GEC at the bottom, function ECs pushed/popped as needed.
- Each EC has two phases:
    1. **Memory Allocation Phase (Creation Phase)**: Scans the code to allocate memory for variables and functions (but doesn't execute code).
    2. **Execution Phase (Code Phase)**: Executes the code line by line, assigning values and running instructions.

**Memory Allocation Phase in Detail**

- Scans the entire code (global or function scope).
- For **variables** declared with var:
    - Allocates memory and initializes to undefined.
- For **functions** (declared with function keyword):
    - Allocates memory and stores the entire function code (as a reference to an object in heap).
- For variables inside functions: Only allocated when the function's EC is created (during invocation).
- No execution happens here; just preparation.
- Example from code:
    - a: undefined
    - b: undefined
    - addNumbers: Entire function code stored.
    - sumResult1: undefined
    - sumResult2: undefined
    - Variables inside addNumbers (e.g., sum): Not allocated yet (waits for function call).

**Execution Phase in Detail**

- Re-scans the code line by line.
- Assigns actual values to variables (replacing undefined).
- Executes statements like function calls, console logs, etc.
- For function calls:

    - Creates a new FEC.
    - Passes arguments (e.g., a and b become num1 and num2).
    - Runs the function's Memory Allocation Phase (e.g., sum: undefined).
    - Then Execution Phase: Assigns values (e.g., num1 = 10, num2 = 20, sum = 30), returns result.
    - Returns control to parent EC; FEC is popped from stack.
- Example Walkthrough:
    - Assign a = 10 (replaces undefined).
    - Assign b = 20.
    - Call addNumbers(a, b): New FEC created.
        - Memory: num1 = undefined, num2 = undefined, sum = undefined.
        - Execution: num1 = 10, num2 = 20, sum = 30, return 30.
    - sumResult1 = 30 (replaces undefined).
    - Similarly for sumResult2 = addNumbers(4, 5) → 9.
    - console.log: Prints 30 9.
- After all execution: GEC is popped; memory is garbage collected.

## 4. Memory Management: Stack and Heap

- **Stack**: For primitive types (e.g., numbers, strings) and execution contexts.
    - Fast, fixed-size.
    - Primitives like null, undefined, true/false are stored in heap but referenced from stack.
    - ECs are pushed/popped here (e.g., GEC at base, FEC on top).
- **Heap**: For non-primitive types (objects, arrays, functions).
    - Dynamic size.
    - Functions are objects: Stored in heap; stack holds reference.
    - Example: addNumbers function code is in heap; stack points to it.
- Behind the Scenes:
    - undefined is a fixed value in heap; variables point to it initially.
    - Function code is converted to bytecode (machine-readable) and stored in a "code space" (part of RAM).
    - Garbage Collection: After FEC completes, its memory is freed (popped from stack); heap objects are collected if unreferenced.

**Proof: Functions as Objects**

- In console: console.dir(addNumbers) shows properties like length: 2 (two parameters), name: 'addNumbers'.
- Confirms functions are objects stored in heap.

## 5. Hoisting: What It Really Is

- **Definition**: JavaScript's behavior where it "knows" about variables and functions before code execution (due to Memory Allocation Phase).

- Before Execution Phase, the engine scans and allocates memory, so declarations are "hoisted" to the top of their scope.
- **Not Literal Movement**: Myth: "Variables/functions are moved to the top." Reality: Only memory allocation happens; code stays in place.
- Depends on keyword:
    - var: Hoisted with undefined.
    - Function Declarations: Fully hoisted (code available).
    - let/const: Hoisted but "uninitialized" (Temporal Dead Zone - TDZ).

Example of Hoisting with var:
JavaScript
```
console.log(a);  // undefined (hoisted)
var a = 10;
```

- `console.log(a);  // 10`
    - Memory Phase: a = undefined.
    - Execution: Logs undefined, then assigns 10.

Function Hoisting:
JavaScript
```
console.log(addNumbers(1, 2));  // 3 (function hoisted)
```

- `function addNumbers(x, y) { return x + y; }`
    - Works because function code is stored in Memory Phase.

## 6. Var vs. Let/Const: Key Differences

- **Var**:
    - Hoisted with undefined.
    - Can be accessed before declaration (gives undefined, not error).
    - Function-scoped (or global if not in function).
- **Let/Const**:
    - Hoisted but in **Temporal Dead Zone (TDZ)**: Allocated memory but "uninitialized."
    - Accessing before initialization: ReferenceError ("Cannot access 'x' before initialization").
    - Block-scoped.

Example:
JavaScript
```
console.log(a);  // ReferenceError (TDZ)
let a = 10;
```

    - `console.log(a);  // 10`
    - Memory Phase: a = uninitialized (TDZ until assigned in Execution Phase).
    - Const: Same as let, but cannot be reassigned after initialization.

Function Expressions with Let/Const:
JavaScript

```
console.log(add(1, 2));  // ReferenceError (TDZ)
```

- `const add = function(x, y) { return x + y; };`
    - Treated as variable: Hoisted uninitialized, not as full function.

## 7. Weird Behaviors and Errors Explained

- Undefined Outputs: Due to accessing variables before assignment in Execution Phase.
- No Error on Function Call Before Declaration: Functions are fully hoisted.
- Errors with Let/Const: TDZ prevents access until assigned.
- Console.log is NOT Part of JavaScript Core: It's a browser/host environment feature (e.g., Node.js or browser console). JS itself uses other mechanisms for output.

## 8. Common Myths and Clarifications

- **Myth**: Hoisting moves declarations to the top.
    - **Reality**: It's just memory allocation in the first phase; code doesn't move.
- **Myth**: All variables are treated the same.
    - **Reality**: Depends on var (undefined) vs. let/const (TDZ).
- Explore More: Read MDN docs on hoisting for official definitions.

## 9. Practice and Interview Tips

- Draw Execution Contexts: Use boxes for Memory/Execution phases; simulate stack/heap.
- Test Code: Use browser console to experiment with hoisting, TDZ.
- Interview Questions:
    - Explain why console.log(a); var a=10; logs undefined.
    - Why does function call before declaration work, but not with function expressions?
    - What is TDZ? Give an example.
- Homework: Rewrite examples with arrow functions (() => {}) – they behave like function expressions (not hoisted fully).

## 10. Conclusion

JavaScript code runs in Execution Contexts with two phases: Memory Allocation (prepares variables/functions) and Execution (runs code). This explains hoisting, where declarations are "known" early, but behavior varies by keyword. Understanding stack/heap and TDZ clears up errors and weird outputs. Rewatch the video or test code for mastery. If concepts "chamak gaye" (clicked), you're ready for advanced JS!

No problem! Let's clear the 3 things you found confusing from the video — **function as object**, **string vs bytecode**, and **what actually happens with the function code** — in the simplest and deepest way possible.

## 1. "Function is an Object" – What does that really mean?

In JavaScript, **everything that is not a primitive is an object** — and functions are not primitive.

| Type | Primitive? | Example | Stored in Heap? | Can have properties? |
|---|---|---|---|---|
| number, string, boolean, null, undefined, symbol, bigint | Yes | 10, "hello" | No (stack) | No |
| Object, Array, Function, Date, RegExp etc. | No | {}, [], function(){} | Yes | Yes |

So a function like this:

JavaScript
```
function addNumbers(num1, num2) {
  return num1 + num2;
}
```

is secretly stored exactly like this under the hood:

JavaScript
```
addNumbers = {
  name: "addNumbers",
  length: 2,                    // number of parameters
  [[Call]]: function code here,  // the actual logic
  prototype: { constructor: addNumbers, ... },
  // and many more internal properties
}
```

Proof you can run right now in browser console:

JavaScript
```
function hello() { return 100; }

console.log(typeof hello);          // "function"  (special type of object)
console.log(hello instanceof Object); // true
```

```
console.log(hello.length);          // 0
console.log(hello.name);            // "hello"

hello.myProperty = "I am a property";
console.log(hello.myProperty);      // "I am a property"  → functions can have properties!
console.dir(hello);                 // ← open this, you will see it is an object
```

Key Point:
Because functions are objects → they live in the **heap**, not the stack.

The variable addNumbers in the stack only holds a reference (pointer) to that big object in the heap.

## 2. String (Text) vs Bytecode – Why the video mentioned two places

When you write this in your .js file:

JavaScript
```
function add(x, y) {
  return x + y;
}
```

This is just plain text (human-readable string). The computer cannot run text directly.

So the JavaScript engine (V8 in Chrome/Node, SpiderMonkey in Firefox, etc.) does this:

| Step | What happens | Where it is stored |
|---|---|---|
| 1 | Your source code (text) is parsed | In memory as string |
| 2 | Converted into an intermediate form called **Bytecode** or **Intermediate Representation** | In a special area called **Code Space** |
| 3 | Bytecode is what the CPU actually executes (after possible JIT compilation to machine code) | Code Space (part of RAM) |

Think of it like this:

- You write a letter in Hindi → that's source code (text).
- Google Translate converts it to English → that's bytecode.
- Only the English version in English can be understood by an English speaker (CPU).

So when the teacher said:

> "The function code is stored as text, but the engine points to bytecode"

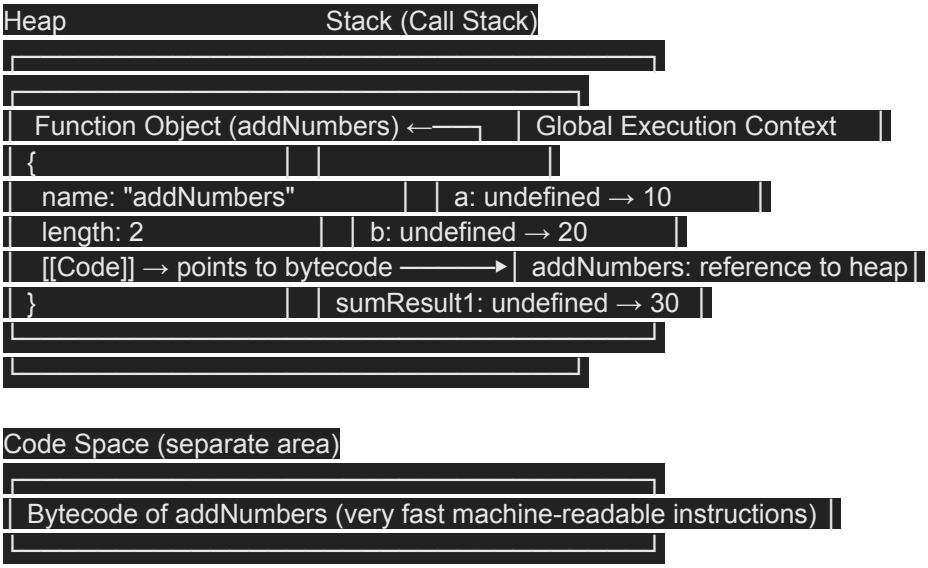
He meant:

- The original text you typed stays in memory.
- But the engine creates a fast executable version (bytecode) separately.
- The function object holds a pointer to that fast bytecode, not the original text.

That's why the function can be called millions of times very quickly.

## 3. Visual Summary (Everything together)

text




## Super Simple Analogy (Hindi style)

Imagine you order a thali in a restaurant:

- Menu card = your source code (text) → kept in kitchen.
- Chef reads the menu and remembers the recipe → parses into bytecode.
- The actual cooking instructions (bytecode) are written on a fast chit and stuck on the wall.
- The waiter (function object) just points to that chit → no need to read the whole menu again.

Every time you order the same thali, the waiter directly looks at the chit (bytecode) → super fast.

## Quick Test You Can Do Right Now

JavaScript

```
function test() {
  console.log("Hello");
}
```

```
// These all prove it's an object
console.log(typeof test);          // "function"
console.log(test.name);            // "test"
console.log(test.length);          // 0
test.customProp = 123;
console.log(test.customProp);      // 123
```

## Summary in 3 Lines

1. Functions are special objects → stored in heap.
2. Your written code (text) is converted to bytecode for speed → stored in code space.
3. The function object only holds a pointer to that bytecode → that's why calling it is lightning fast.

Ab samajh aa gaya na?

If you still have any doubt on any of these 3 points, just type "bhai ek aur example do" and I will give 5 more examples instantly!

Haan bhai, ab bilkul sahi jagah pakda tune!
Video mein jo teacher ne thoda jaldi-jaldi mein bola tha, usse confusion hua tha.

Let me clear this **100% accurately** – exactly kya hota hai function ke code ke saath, aur **FEC (Function Execution Context)** ke time kya store hota hai.

## Final Clear Answer (2025 V8/SpiderMonkey reality)

| Question | Answer |
|---|---|
| Function ka code **string** ke roop mein rehta hai kya? | **Nahi**. Sirf parsing ke time pe string hota hai. Uske baad **kabhi string nahi rehta**. |
| Function ka code **bytecode** mein hota hai kya? | **Haan**, lekin indirectly. Function object ek pointer hold karta hai jo bytecode ko point karta hai. |
| FEC banne ke time function ka code kahan se aata hai? | Function object ke andar ek hidden property hoti hai [[Code]] ya [[FunctionBytecode]] jo already compiled bytecode ko point karti hai. |

## Step-by-Step Kya Hota Hai (Real Engine Behavior)

JavaScript

```
function add(x, y) {
  return x + y;
```

}

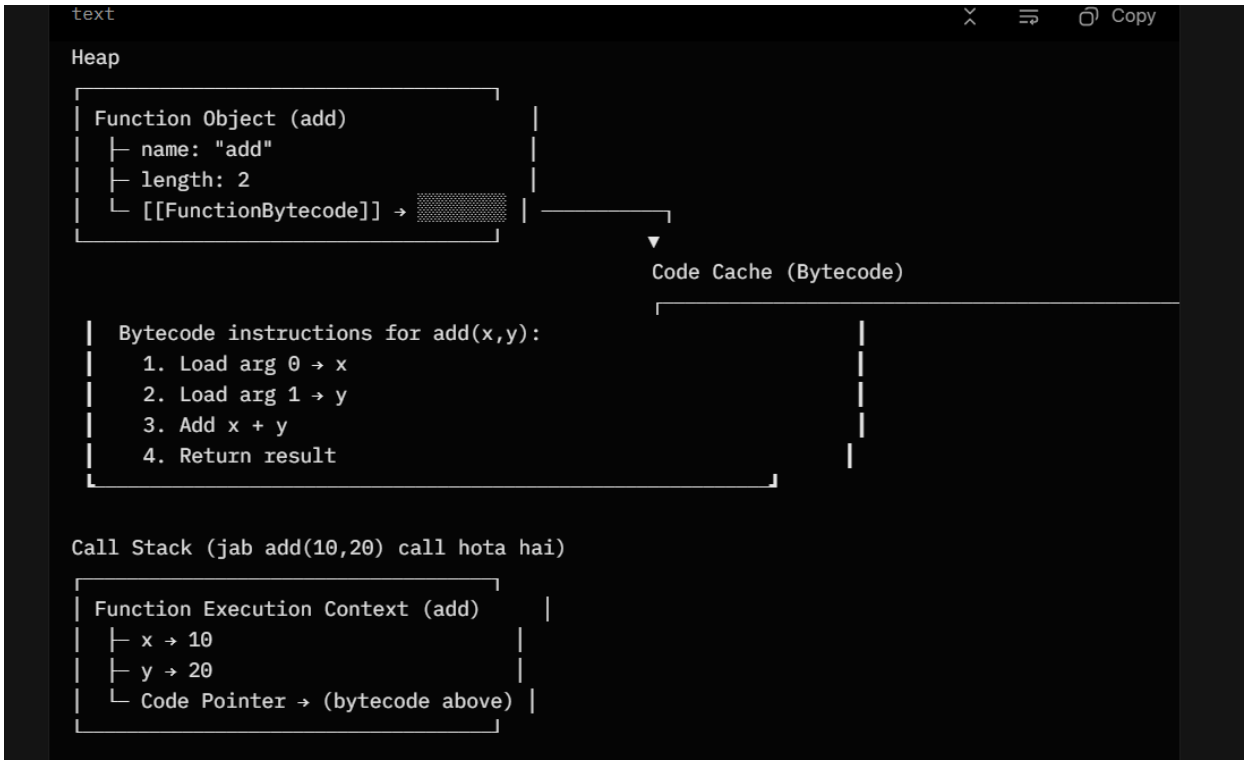
1. **Parsing Phase** (sirf ek baar, jab file load hoti hai)
    - Tera pura code string ke roop mein hota hai
    - Engine parse karta hai → syntax tree banata hai
2. **Compilation Phase** (ek baar hota hai)
    - Function add ka code **bytecode** mein compile ho jata hai
    - Ye bytecode RAM ke ek special area mein store hota hai (code cache / code space)

**Function Object Creation** (Memory Allocation Phase mein)

JavaScript

```
add = {
  name: "add",
  length: 2,
  [[FunctionBytecode]] → pointer to already compiled bytecode,
  prototype: { ... }
```

3. }
   → Isme koi string nahi hoti, sirf pointer hota hai bytecode ka
4. **Jab function call hota hai add(10, 20)** → FEC banta hai
    - Naya Execution Context push hota hai stack pe
    - Is FEC ke andar sirf ye cheezen hoti hain:
        - Arguments: x = 10, y = 20
        - Local variables: (none here)
        - Pointer to the **already compiled bytecode** of the function
    - **Function ka original text code kahi bhi nahi jata FEC mein**




## Summary in 4 Bullet Points

1. Function ka **source code (text)** sirf parsing ke time dikhta hai, uske baad gayab.
2. Function object sirf ek **pointer** hold karta hai jo compiled bytecode ko point karta hai.
3. Jab bhi function call hota hai, FEC mein sirf arguments aur local vars aate hain, **function ka text code nahi aata**.
4. Bytecode already compiled hota hai, isliye function call bahut fast hota hai (microseconds mein).

## Real Proof You Can Run

JavaScript

```
function test() { console.log("hi"); }

console.log(test.toString());
// → "function test() { console.log("hi"); }"
// Ye sirf ek helper method hai jo original source recreate karta hai (agar available ho)
// Actual execution mein ye string use nahi hoti!

// V8 mein internal property dekhne ke hack (Node.js)
if (globalThis.process) {
  console.log(require('util').inspect(test, {showHidden: true}));
  // You will see [[FunctionLocation]], [[Bytecode]] etc. — no raw string
}
```

Ab bilkul clear ho gaya na bhai?
Function ka code **string nahi rehta**, wo **bytecode** ban chuka hota hai aur sirf pointer hota hai FEC mein.

Koi doubt bacha to bol dena, 10 examples de dunga!

## In-Depth Explanation of "Function Code in Text Form, Bytecode Requirement, and Pointer to Code Space"

Hey! Based on your query, you're specifically confused about this part from the transcript (around 00:23:05 to 00:24:54):

> "ये जो कोड है ना आपका ये कोड ये ये आपका नॉर्मल ऐसा प्रेजेंट नहीं होता ये होता है बाइट कोड के फॉर्म में जिसको कंप्यूटर समझ सकता है ये ऐसे टेक्स्ट के फॉर्म में ये तो मैंने आपको दिखाने के लिए कर दिया। ... मेरे अंदर तो वो जो कोड है वो प्लेन टेक्स्ट में लिखा गया है। ... मेरे सिस्टम को अगर उस कोड को रन करना है तो उसको बाइट कोड में होना चाहिए। ... टेक्स्ट फॉर्म को वो समझ नहीं सकता एग्जीक्यूट नहीं कर सकता इसलिए वो पॉइंटर डाल देता है कि वो आपको इस कोड स्पेस से कहां मिलेगा"

In simple Hindi-English: The teacher is saying that the function's code isn't stored as plain text (jo aapne file mein likha hai) in the running program. Instead, it's converted to **bytecode** (a machine-friendly format), and the function object just has a **pointer** (ek reference/address) pointing to where this bytecode is stored in a special memory area called **code space**. This makes execution fast because the system doesn't have to read or understand the original text every time.

I'll explain this **in very very depth**, step by step, drawing directly from the transcript and cross-referencing with how real JavaScript engines (like V8 in Chrome/Node.js) work. I'll use examples, visuals (referencing your image), tables for clarity, and analogies. This is based on up-to-date knowledge (as of December 2025) from V8 docs and articles.

### 1. Context from the Transcript: What the Teacher is Trying to Say

- The teacher is diving into "behind the scenes" (bihaind d seen) of how a function like addNumbers is handled.
- In the Memory Allocation Phase (of Global Execution Context - GEC), the function addNumbers is treated as an **object** stored in the **heap**.
- Inside this object, the function's code isn't stored as the plain text you wrote (e.g., "var sum = num1 + num2; return sum;"). That's just for humans.
- Instead, the engine converts it to **bytecode** (a compact, executable format).
- The function object holds a **pointer** (ek arrow/address) to this bytecode in a special RAM area called **code space**.
- Why? Because the system (CPU) can't directly run your text code—it needs something closer to binary (0s and 1s). Bytecode is that "middle language."

- When the Function Execution Context (FEC) is created (on function call), it uses this pointer to quickly find and run the bytecode—no need to re-process the text.

**Your Image Reference**: In the screenshot, see the arrow from the function in the code to "fn" in the heap? That's visualizing the pointer. The "Byte Code Code space" label points to how the actual runnable code (bytecode) is separate, not embedded as text. The stack has variables pointing to "Undefined" in heap, showing references.

**Analogy**: Imagine your JS code is a recipe written in Hindi (text form). The chef (system) can't cook from Hindi—it needs English instructions (bytecode). So, you translate it once and store the English version in a kitchen notebook (code space). The recipe name (function object) just has a page number (pointer) to that notebook. When cooking (running), the chef flips to that page super fast.

**2. Step-by-Step: From Your Written Code (Text) to Execution**

Let's break it down into phases, using the example from the transcript:

JavaScript

```
function addNumbers(num1, num2) {
  var sum = num1 + num2;
  return sum;
}
```

| Phase | What Happens | Where Stored | Role of Text/Bytecode/Pointer |
|---|---|---|---|
| **1. Source Code Loading** | Your .js file is read as plain text (string). This is the "text form" the teacher mentions. | Temporary in RAM (as a string buffer). | Text is raw input; not executable yet. System can't "run" it directly. |
| **2. Parsing** | V8's parser scans the text, breaks it into tokens (e.g., "function", "addNumbers", "{", etc.), and builds an **Abstract Syntax Tree (AST)**—a tree structure representing the code logic. E.g., AST node for "function declaration" with children for parameters and body. | AST is a heap object (tree of nodes). | Text is discarded after parsing; AST is abstract (no text, just structure). Errors like syntax mistakes are caught here. |

| | | | |
|---|---|---|---|
| **3. Bytecode Generation** | Ignition (V8's interpreter/compiler) takes the AST and generates **bytecode**—a sequence of low-level instructions. This is the "bytecode me hona chahiye" part. Bytecode is like mini-commands: "load num1", "add num2", "store in sum", "return". It's not full machine code but efficient for interpretation. | Bytecode is stored as a BytecodeArray object in **code space** (a special executable RAM area in the heap). This array holds opcodes (e.g., LdaSmi for load small integer) + operands (e.g., registers like r0). | Original text is gone; bytecode is created once (lazily, only when needed). This is why the teacher says "text form ko woh samajh nahi sakta"—CPU needs this format. |
| **4. Function Object Creation** | During Memory Allocation Phase, the function is created as an object in heap. It doesn't store the full bytecode inside—it holds a **pointer** (memory address) to the BytecodeArray in code space. Other properties: name: "addNumbers", length: 2 (params), prototype. | Heap (non-primitive, so dynamic memory). Pointer is a field like [[Bytecode]] or [[Code]]. | Pointer enables efficiency: No copying bytecode every call. Teacher's words: "wo pointer dal deta hai code space me kaha milega." Code space is V8's area for these arrays. |
| **5. Function Call & FEC Creation** | When called (e.g., addNumbers(10, 20)), a new FEC is pushed to call stack. It includes: arguments (num1=10, num2=20), local vars (sum=undefined initially). The FEC uses the function object's pointer to fetch bytecode from code space. | Stack for FEC (primitives/references); bytecode remains in code space. | Pointer makes it fast: Direct jump to bytecode. No re-parsing text. Execution: Ignition interprets bytecode line-by-line, using registers (e.g., accumulator for temp values). |

| | | | |
|---|---|---|---|
| **6. Execution & Optimization** | Ignition runs bytecode: E.g., load 10 to register, add 20, store sum, return 30. Collects "feedback" (types used) in a **Feedback Vector** (another pointer in function object). If hot (called often), TurboFan optimizes to machine code (even faster). | Code space for bytecode/machine code. | Pointer + feedback = efficiency. Original text isn't needed—bytecode is the "runnable" form. After run, FEC pops from stack; garbage collector cleans if unreferenced. |

**Depth on Bytecode Example** (from V8 internals):

For addNumbers:

Bytecode might look like:
text

```
StackCheck          // Check stack overflow
LdaSmi r0, [10]     // Load num1 (10) to register r0 (but actually uses accumulator)
LdaSmi r1, [20]     // Load num2 to r1
Add r0, r1          // Add r0 + r1 into accumulator
Star r2             // Store to sum (r2)
Ldar r2             // Load sum to accumulator
```

- `Return             // Return accumulator (30)`
- Opcodes (e.g., Add) are bytes (8-32 bits). Operands are registers (virtual, not CPU registers) or indices to **constant pool** (stores strings/numbers used in code, like property names if any).
- Accumulator: A special register to hold temp results, reducing instruction size (teacher implies this efficiency).

**Why Not Keep Text?** Text is human-readable but slow/inefficient for machines. Parsing text every call would be wasteful (time + CPU). Bytecode is compact (e.g., 10x smaller) and directly interpretable.

**3. Role of Pointers: Why So Important? (Deep Dive)**

- **Efficiency Gains**:
    - **Speed**: Pointer is just a 64-bit address—fetching bytecode is O(1) time (instant). Without it, engine would re-parse text (slow, like reading a book every meal).
    - **Memory Saving**: Bytecode isn't duplicated in every function instance. Shared via pointer (e.g., closures or multiple calls reuse the same array).
    - **Optimization**: Pointer allows attaching a **Feedback Vector** (array of slots, one per bytecode instruction). E.g., Slot [0] tracks if add was always numbers (monomorphic) for TurboFan to compile to native code.

    - **Portability**: Bytecode is platform-independent; pointers abstract hardware differences.
- **In Transcript Image**: The arrow from code to "fn" in heap is the pointer. "Byte Code Code space" is the separate area (V8's BytecodeArray in isolated heap space to prevent exploits).
- **Real V8 Proof**: In V8 source (C++), functions have SharedFunctionInfo with a bytecode_array_ field—a pointer to BytecodeArray. When FEC runs, it dereferences this to get instructions.

**Analogy Depth**: Pointer is like a bookmark in a library book (code space). The function object is the library card with the bookmark number. When you "call" (read), you go straight to the page—no searching the whole library.

**4. Code Space: What Is It Exactly?**

- In V8, **code space** is a reserved heap region for executable code (bytecode + machine code). It's "executable" memory (OS permission: read/execute, not write, for security).
- Size: Dynamic, but V8 limits (e.g., 1-4 MB initially, grows as needed).
- Management: Garbage collector relocates if needed, updating pointers (compaction).
- Why Separate? Prevents attacks (e.g., code injection); optimizes for CPU caches.
- Teacher's Point: "code space se kaha milega"—the pointer tells exactly where in this space the bytecode starts.

**5. Common Confusions Cleared (from Transcript)**

- **Text Form Hai?**: Yes, initially (your .js file). But after parsing, no—converted and discarded. function.toString() recreates text for debugging, but it's not used for running.
- **Bytecode Me Hona Chahiye?**: Yes, because interpreter (Ignition) executes bytecode, not text. CPU understands binary; bytecode is a bridge.
- **Pointer Dalega?**: Yes, in function object. During FEC, it follows the pointer to run.
- **Weird Outputs/Errors**: This setup enables hoisting (functions ready early via pointer) but causes TDZ for let/const (uninitialized until execution phase).

**6. Test It Yourself (Practical Depth)**

Run in browser console:

JavaScript
```
function add(x, y) { return x + y; }
console.dir(add);  // See '[[FunctionLocation]]' or '[[Bytecode]]' in internals (may need --allow-natives-syntax flag in Node).
console.log(add.toString());  // Recreates text, but not used for execution.
```

- Tools like Chrome DevTools > Performance > Record > JS Profile show bytecode.

**7. Summary: Ab Chamak Gaya?**

The teacher is spot-on but simplifying: Function code starts as text, gets parsed to AST, compiled to bytecode in code space. The function object points to it via a pointer for fast access in FEC. No text is kept for running—bytecode is the key. This makes JS efficient despite being interpreted. If still doubt, ask for a specific part!

Sources: V8 docs, Medium articles on V8 bytecode, freeCodeCamp V8 guide.