HOW JAVASCRIPT EXECUTE CODE ?

What is Execution ?

Execution is the process by which a computer or virtual machine reads and acts on the instructions of a computer program.

In general terms, it is like you write some code, your computer read it and do the actions mentioned in the written code.

This overall process of reading code and doing some actions based on that code is called execution.

How execution is done for javascript ?

The execution process is done by a special program called Javascript Engine.

A popular javascript engine is chrome V8 engine developed by Google.

Some memory and additional resource are needed, to create an environment for code execution. This job is done by js engine.

The environment that is created for execution is called execution context. All the Javascript code runs inside it.

What is JS Engine ?

A **JavaScript Engine** is a **program that reads, compiles, and executes JavaScript code**. It's the "brain" that makes your JS run inside browsers (like Chrome, Firefox, Safari) or environments like Node.js.

- **What does a JS Engine do?**

When you write JavaScript:

console.log("Hello World");

The engine:

1. **Parses** the code into tokens and builds an AST (Abstract Syntax Tree).
2. **Compiles** it into bytecode.
3. **Interprets** or **JIT compiles** the bytecode into machine code.
4. **Executes** it on the CPU.

5. Uses **memory management & garbage collection** to free unused memory.
6. Provides built-in **APIs** (like console.log, Math, Date) from the host environment.

🔹 **Popular JavaScript Engines**

- **V8** → Chrome, Node.js (Google's engine, very fast, uses Ignition + TurboFan).
- **SpiderMonkey** → Firefox (first JS engine ever).
- **JavaScriptCore (Nitro)** → Safari (Apple).
- **Chakra** → Microsoft Edge (old version, now replaced by V8 when Edge moved to Chromium).

🔹 **Key Components inside an Engine**

1. **Parser** – Reads code → creates AST.
2. **Interpreter** – Runs bytecode (quick startup).
3. **JIT Compiler** – Compiles hot code paths into optimized machine code.
4. **Garbage Collector (GC)** – Frees unused memory automatically.
5. **Call Stack & Heap** – Manages function calls and memory.

✅ **In short:**
A JavaScript Engine is a specialized program (like V8, SpiderMonkey) that **turns your JavaScript code into machine code so your computer can run it**, while managing memory and performance optimizations.

How Javascript code is executed ?

1. JavaScript Engine

JavaScript doesn't run directly on hardware (like C/C++). Instead, it needs a **JavaScript engine** inside the browser (like **V8 in Chrome/Node.js**, **SpiderMonkey in Firefox**, **JavaScriptCore in Safari**).

The engine takes the JS code and executes it.

---

2. Execution Context

Whenever JS code runs, it creates an **Execution Context**.
There are mainly two types:

- **Global Execution Context (GEC):** Created when the file first runs. It has the global object (window in browser, global in Node.js) and this.
- **Function Execution Context (FEC):** Created when a function is called.

Each execution context has **two phases**:

1. **Creation Phase (Memory allocation / Hoisting):**
    - Variables (var) and functions are hoisted (memory allocated).
    - let and const are hoisted but stay in the **Temporal Dead Zone (TDZ)** until initialized.
    - Function definitions are fully stored.
2. **Execution Phase:**
    - Code is executed line by line.
    - Variables get their assigned values.

---

3. Call Stack

- JavaScript is **single-threaded** and uses a **Call Stack** (LIFO – Last In First Out).
- The Global Execution Context is pushed first.
- Each time a function is called, a new Execution Context is pushed on top.
- When a function finishes, its context is popped out.

---

4. Event Loop & Asynchronous Execution

JS is synchronous by default, but for asynchronous tasks (setTimeout, fetch, promises):

- The **Web APIs** (in browser) or **Node APIs** (in Node.js) handle async tasks.
- Results are sent to the **Callback Queue** or **Microtask Queue**.
- The **Event Loop** continuously checks:
    - If the Call Stack is empty → takes tasks from the queue → pushes into Call Stack for execution.

---

5. Compilation (JIT – Just in Time)

Modern engines (like V8) don’t interpret line by line slowly. Instead:

- **Parsing:** Code is parsed into an **AST (Abstract Syntax Tree)**.

- **Interpreter (Ignition in V8):** Quickly converts AST into bytecode and starts running.
- **Compiler (TurboFan in V8):** Optimizes hot code paths into machine code for faster execution.
- This **JIT compilation** makes JavaScript much faster.

✅ **In summary:**

1. JS engine loads the code.
2. Execution Context (memory + execution phases) is created.
3. Call Stack manages execution order.
4. Async tasks handled by Web APIs + Event Loop.
5. Modern JIT engines optimize performance.

What happens in the Compilation Step :

When you write JS:

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

The engine doesn't directly run it — it goes through these **compilation phases**:

---

1. Parsing (Syntax Analysis)

- **Lexical Analysis (Tokenization):** The source code is broken into tokens (keywords, identifiers, literals, operators).
    - Example: function, add, (, a, ,, b,), {, return, a, +, b, ;, }
- **Parsing:** Tokens are arranged into a tree structure called **AST (Abstract Syntax Tree)**.
    - Example: add → FunctionDeclaration node with two parameters and a BinaryExpression (a + b).

🔹 **Purpose:** AST is easier for the compiler to analyze and transform than raw source code.

---

2. Intermediate Representation (IR) / Bytecode Generation

- The AST is translated into **bytecode** (low-level instructions for the interpreter).
- Example (simplified):
- Load a
- Load b
- Add
- Return
- This bytecode is like a **mini virtual machine language** (similar to JVM bytecode).

🔹 **Purpose:** Allows quick execution without waiting for full optimization.

---

3. Baseline Execution (Interpreter runs bytecode)

- The engine starts running bytecode using an **interpreter** (e.g., Ignition in V8).
- While running, it **collects type information**:
    - What types of values are passed? (numbers, strings, objects)
    - What properties are accessed?
    - How many times is this function called?

🔹 **Purpose:** Gather runtime behavior to optimize later.

---

4. Optimization (JIT Compilation)

- If a function runs often (hot function), the engine compiles it into **machine code** using the **optimizing compiler** (e.g., TurboFan in V8).
- Optimizations include:
    - **Type specialization:** If a and b are always numbers, compile a fast machine instruction for numeric addition.
    - **Inlining:** Replace small function calls with their body to remove call overhead.
    - **Hidden classes & inline caches:** Optimize property lookups by treating objects with similar structure as having a "class."

🔹 **Purpose:** Produce fast machine code for frequently used functions.

---

5. Speculative Assumptions & Deoptimization

- Optimizations are **speculative** → the compiler assumes things stay consistent.
    - Example: If a+b always sees numbers, it will optimize for numbers.
- If later the code violates assumptions (e.g., add("x", "y")), the engine:
    - **Deoptimizes** → throws away the optimized machine code.
    - Falls back to safe **bytecode execution** in the interpreter.

🔹 **Purpose:** Stay fast for common cases, but safe for dynamic cases.

🔹 **Compilation Pipeline (V8 Example)**

1. **Parser** → AST
2. **Ignition (Interpreter)** → Bytecode + profiling
3. **TurboFan (Optimizing Compiler)** → Optimized machine code
4. **Deoptimization** if assumptions break

✅ **In short:**

- JS code is first parsed into AST.
- Compiled into bytecode (fast startup).
- Interpreter runs bytecode & collects type feedback.
- Optimizer JIT compiles hot functions into machine code.
- If assumptions fail → fallback to interpreter (deopt).

Just-In-Time Compilation (JIT)

JIT (*Just-In-Time Compilation*) is a compilation process in which code is translated from an intermediate representation or a higher-level language (e.g., JavaScript or Java bytecode) into machine code *at runtime*, rather than prior to execution. This approach combines the benefits of both interpretation and ahead-of-time (AOT) compilation.

JIT compilers typically continuously analyze the code as it is executed, identifying parts of the code that are executed frequently (hot spots). If the speedup gains outweigh the compilation overhead, then the JIT compilers will compile those parts into machine code. The compiled code is then executed directly by the processor, which can result in significant performance improvements.

JIT is commonly used in modern web browsers to optimize the performance of JavaScript code.

What is Abstract Syntax Tree ?

An **Abstract Syntax Tree (AST)** is a **tree-shaped data structure** that represents the structure of source code in a way that's easier for compilers or interpreters to work with.

🔹 Why AST?

When you write code, the engine can't work directly with raw text.
So it:

1. **Tokenizes** (breaks into pieces).
2. **Parses** → builds a tree that shows how the code is structured logically.

🔹 Example

JavaScript code:

let x = 5 + 3;

AST (simplified form):

VariableDeclaration
└── VariableDeclarator
    ├── Identifier (x)
    └── BinaryExpression (+)
        ├── Literal (5)
        └── Literal (3)

Here:

- VariableDeclaration = we are declaring a variable.
- Inside it: VariableDeclarator = variable being declared.
- Identifier (x) = variable name.
- BinaryExpression (+) = operation.
- Literal (5) and Literal (3) = numbers.

🔹 Key Features of AST

- It ignores *syntax details* like semicolons, whitespace, or parentheses when unnecessary.
- Keeps only **meaningful structure** (operators, variables, functions, conditions, etc.).

- Provides a **hierarchical view** (parent → child relationships).

🔹 **How AST is used**

1. **Compilation / Interpretation** → Engine converts AST into bytecode/machine code.
2. **Linting (ESLint, Prettier)** → Tools analyze AST to check style, errors, unused variables.
3. **Code Transformation** → Tools like Babel or TypeScript use AST to transpile code (ES6 → ES5).
4. **Optimization** → Engines optimize code based on AST before execution.

✅ **In short:**
An **AST** is the structured, tree-like representation of your source code, used by compilers, interpreters, and tools to **understand, analyze, and transform code**.

JavaScript Compilation

While JavaScript is an interpreted language, it does have some aspects of compilation. Modern **JavaScript engines** have evolved to employ a hybrid approach that combines both compilation and interpretation techniques. While JavaScript's specification **does not explicitly state a compilation step**, the language's implementation does involve a process of parsing, compiling, and interpreting the code practically. This approach is essential because certain language features, such as **hoisting** and **variable** scoping, require processing during the compilation phase. There are 3 use cases that enable us to investigate that approach:

1-) Syntax Errors: Syntax errors occur when the code contains invalid syntax. Here's an example of code that contains a syntax error:

```
const x = 10;

console.log(x);
// should log the value of x

console.log("The value of x variable is: " x);
// throws a SyntaxError
```

In this code, there is a **missing plus sign (+)** between the string literal and the x variable in the console.log statement. When this program is executed it throws a SyntaxError **without**

logging the value of x. The only way the JS engine knows about the error in line 6 before executing lines 1 and 3 is it requires a **parsing step** before executing the entire code.

2-) Early Errors: An early error is a type of error that occurs during the parsing or compilation phase of a program, before the program is executed, and is caused by violating certain language rules or requirements. There we have an example:

```
// SyntaxError: Duplicate parameter name not allowed in this context
function myFunction(a, a) {
  "use strict";
  console.log(a);
}

myFunction(1, 2);
```

In this code, we are trying to declare a function that takes 2 parameters both named “a”. In **strict mode** duplicate parameter naming is **not allowed** for sure, but how does the JS engine know the “myFunction()” function is in strict mode and “a” parameter has been duplicated? Just because code has to be fully parsed before the execution.

3-) Hoisting: Hoisting is a behavior that occurs during the compilation phase and allows functions and variables to be declared and used before they are defined in the code. Here’s an example of code that demonstrates hoisting:

```
x = 10;

console.log(x);

var x;
```

The reason why this code works is the **hoisting** concept in JavaScript. When a variable is declared using the “var” keyword, its declaration is hoisted to the top of the scope allowing it to be used before it is declared. This example also shows us declarations can only be accomplished correctly by parsing the program before execution.

Improving JavaScript Performance with Just-In-Time Compilation

In modern JavaScript engines, the compilation is often done in a **Just-In-Time (JIT)** approach. JIT means that the compilation happens at runtime, just before the code is executed. Initially, the engine interprets the code line by line, but as the code is executed, the engine identifies frequently executed code segments, also known as **“hot code paths”**.The JIT compiler then compiles these hot code paths into executable code, which can be executed more efficiently by the computer’s processor.

By using JIT compilation, JavaScript engines can dynamically optimize code execution at runtime, leading to significantly **improved performance** compared to pure interpretation or pure compilation approaches.

How Programs are Transformed into Executable Code

In general, compiling a program involves these three fundamental stages that play a crucial role in turning the code into executable instructions:

1-) Tokenizing/Lexing: Breaking the code down into meaningful chunks or tokens. For instance, in the following line of code

```
let x = 42;

// the tokens would be "let", "x", "=", "42", and ";".
```

2-) Parsing: the tokens are parsed to create an **Abstract Syntax Tree (AST) —** a structured representation of the code's grammatical structure. This tree can be thought of as a hierarchical model of the code, which can be manipulated and optimized.

3-) Code Generation: AST is translated into executable code by the code generation process, which can vary depending on the language, platform, and other factors. For example, in the case of a JavaScript engine, the AST for the code "let x = 42;" might be translated into machine instructions that **reserve memory** for a variable called "x" and **store the value** 42 in it.

Performance Factors

In addition to Just-In-Time compilation, there are several other factors that can affect the performance of JavaScript code execution. One of the most important factors is the **browser's JavaScript engine**. Modern browsers like **Chrome** and **Firefox** have highly optimized JavaScript engines that can execute code **much faster** than older browsers.

Another important performance factor is ***caching***. When a JavaScript file is loaded, it is cached by the browser so that it can be loaded more quickly in the future. This can significantly improve the performance of JavaScript code execution, especially for large and complex applications.

Precompilation is another technique that can be used to improve the performance of JavaScript code execution. Precompilation involves compiling the JavaScript code before it is executed, which can help to reduce the amount of time needed for JIT compilation.

Modern JavaScript Technologies

Modern JavaScript libraries and frameworks like *React.js* and *Next.js* can also have a significant impact on the performance of JavaScript code execution. These frameworks use advanced techniques like **virtual DOM** and **server-side rendering** to improve the speed and efficiency of web applications.

Virtual DOM is a technique used by React to improve the performance of **updating the user interface**. Instead of updating the **entire UI** every time a change is made, virtual DOM allows the framework to update only the **parts of the UI that have changed**, which can significantly reduce the amount of time needed for updates.

Server-side rendering is another method to improve performance. Server-side rendering involves rendering the initial HTML **on the server instead of the client**, which can significantly reduce the time needed for the initial page load.

Conclusion

JavaScript code execution is a complex process that involves both compilation and interpretation. In addition to JIT compilation, there are other factors that can affect the performance of JavaScript code execution, such as the browser's JavaScript engine, caching, and precompilation. While JavaScript is **primarily an interpreted language**, as we show there are a bunch of use cases that show it does have some aspects of compilation that help to optimize performance.