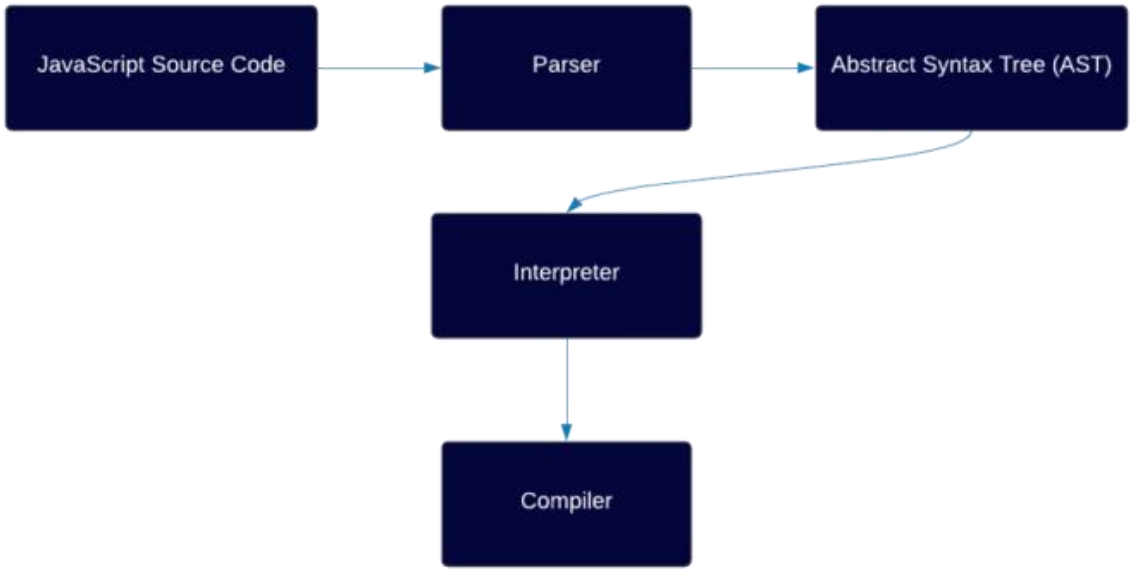
## JS Engine:

A JavaScript engine is a software component that interprets and executes JavaScript code. It is a crucial part of web browsers and many other applications that rely on JavaScript for scripting and interactivity. It is the execution unit of browser.

**List of JS Engines:**

- Google Chrome - V8,
- Edge (Internet Explorer) - Chakra
- Mozilla Firefox - Spider Monkey
- Safari - JavaScript Core

**How JS Engine works:**




1. The JavaScript engine first performs lexical analysis or scanning, where it reads the source code character by character and groups characters into meaningful tokens, like keywords, identifiers, literals, and operators.

2.After lexical analysis, the engine parses the tokens into an Abstract Syntax Tree (AST), which represents the syntactic structure of the code.

The AST is a hierarchical tree-like structure that defines the relationships between different parts of the code, such as statements, expressions, and operators.

3.After the result of AST, the interpreter checks the code line by line and generate the byte code. But in some engines use Just-In-Time (JIT) compilation techniques to optimize the code. This involves analysing the code's execution patterns and applying various optimizations to make it run faster.

4.After compilation and optimization, the JavaScript code is executed. During execution, the engine manages the call stack, which keeps track of function calls and their contexts (local variables and state).

## Garbage Collection:

While executing the code, the engine performs automatic memory management, known as garbage collection. It identifies and frees up memory occupied by objects that are no longer referenced, preventing memory leaks.

## Global Execution Context:

In JavaScript, the Global Execution Context (GEC) is the top-level context or environment in which your JavaScript code is executed. It represents the outermost scope of your code, and it's created when your script is initially run. The GEC has a crucial role in managing and coordinating the execution of your entire JavaScript program. Let's dive into the phases of the Global Execution Context:

**Creation Phase:**

**Variable Object (VO) Creation:**

The GEC begins with the creation phase, during which JavaScript sets up the environment for your code. One of the first tasks is to create the Variable Object (VO), also known as the Global Object (in the case of a browser environment, it's the window object). The VO holds all globally declared variables and functions. Variables are initialized to undefined during this phase.

**Scope Chain:** The scope chain is established, which consists of the VO and any outer function or lexical scopes that might be present. In the global context, the scope chain typically contains only the global object.

**Hoisting:**

During the creation phase, function declarations are hoisted to the top of their containing scope. This means that functions can be called before they are defined in the code.

Variable declarations (using var) are also hoisted to the top of their containing function or global scope but are initialized with undefined. This is why you can access variables before they are declared without getting an error.

**Execution Phase:**

After the creation phase, JavaScript proceeds to the execution phase. This is where the actual code is executed line by line.

For each statement and expression in the code:

- Variables that have been declared (but not initialized) are assigned their initial value, which is undefined.
- Function declarations are made available for use throughout the entire scope.
- Code is executed sequentially, and any functions, variables, or operations are processed as they are encountered.
- If there are function calls, new execution contexts (Local Execution Contexts) are created for those functions, and the same creation and execution phase steps are followed within those contexts.
- The scope chain is used to resolve variable references, starting with the innermost scope and progressing outward until a match is found or the global scope is reached.

**Cleanup:**

Once the code in the GEC has been executed, the GEC itself doesn't get destroyed. However, the variables and functions declared within it remain accessible as properties/methods of the global object (e.g., window in a browser environment).

If a variable or function was declared using let or const in the global scope, they won't become properties of the global object.

## Call Stack in JS:

In JavaScript, the call stack is a data structure used to keep track of function calls and their respective execution contexts during the execution of a program. It operates on a last-in, first-out (LIFO) basis, meaning that the most recently called function is the first one to be executed and removed from the stack when it's completed.

## Promises in JS:

Promises in JavaScript provide a way to work with asynchronous operations in a more structured and manageable manner. They allow you to represent a value that might not be available yet but will be at some point in the future. Promises have become a standard mechanism for dealing with asynchronous code, and they simplify error handling and improve code readability.

## Promise States:

A promise can be in one of three states: pending, fulfilled (resolved), or rejected.

- Initially, a promise is in the "pending" state, meaning it hasn't been resolved or rejected yet.
- When a promise's asynchronous operation completes successfully, it transitions to the "fulfilled" state with a resolved value.
- If an error occurs during the asynchronous operation, the promise transitions to the "rejected" state with a rejection reason (an error object).

## Promise Creation:

You can create a promise using the Promise constructor, which takes an executor function as its argument. The executor function is called immediately and is passed two functions as arguments: resolve and reject.

You call resolve(value) when the asynchronous operation is successful and reject(reason) when an error occurs. The value is the result of the successful operation, and the reason is an error object.

Promise instances in JavaScript come with several instance methods that allow you to interact with and manipulate promises. These methods make it easier to work with asynchronous operations and handle the results or errors.

**then(onFulfilled, onRejected):**

The then method is used to specify what should happen when a promise is fulfilled (resolved) or rejected.

It takes two optional callback functions as arguments: onFulfilled and onRejected. These functions are called based on the promise's state.

If the promise is fulfilled, onFulfilled is called with the resolved value as its argument.

If the promise is rejected, onRejected is called with the rejection reason (an error object) as its argument.

The then method returns a new promise, allowing you to chain multiple then calls together.

**catch(onRejected):**

The catch method is a shorthand for specifying what to do when a promise is rejected. It takes a single callback function, onRejected.

This method is often used for error handling, especially when you want to handle errors for all preceding promise chain steps.

**finally(onFinally):**

The finally method allows you to specify a callback function, onFinally, that will be executed regardless of whether the promise is fulfilled or rejected. It is often used for cleanup operations.

The finally method returns a new promise that is resolved with the original promise's resolution value or rejection reason after onFinally has been executed.

**Promise Chaining:**

Promises allow you to chain asynchronous operations together using the .then() method. The .then() method is called when a promise is fulfilled and takes a callback function that processes the resolved value.

You can chain multiple .then() calls to create a sequence of asynchronous steps.

Additionally, you can handle errors by chaining a .catch() method at the end of the chain to catch any rejections in the promise chain.

Promise static methods:

**Promise.all(iterable):**

The Promise.all method takes an iterable (typically an array) of promises as its input and returns a new promise.

It waits for all the promises in the iterable to either fulfill or reject.

If all promises fulfill, the returned promise fulfills with an array of their resolved values in the same order as the original iterable.

If any promise in the iterable rejects, the returned promise immediately rejects with the reason of the first rejected promise.

**Promise.any(iterable):**

The Promise.any method takes an iterable of promises. It returns a new promise that fulfills with the value of the first promise in the iterable to fulfill. If all promises in the iterable are rejected, it will give aggregate error.

This method is useful when you want to work with the first successful result among multiple promises.

**Promise.allSettled(iterable):**

The Promise.allSettled method is also introduced in ECMAScript 2021 (ES12).

It takes an iterable of promises and returns a new promise that fulfills with an array of objects, each representing the outcome (fulfilled or rejected) of a promise in the iterable.

Each object in the array contains a status property (either "fulfilled" or "rejected") and a value or reason property, depending on whether the promise was fulfilled or rejected.

**Promise.race(iterable):**

The Promise.race() static method takes an iterable of promises as input and returns a single Promise . This returned promise settles with the eventual state of the first promise that settles.

## window.fetch():

The fetch() method in JavaScript is a modern API for making network requests, primarily for fetching resources (like data or files) from a server or other online sources. It provides a more flexible and powerful way to work with HTTP requests compared to older techniques like XMLHttpRequest. fetch() returns a promise, which makes it easy to handle asynchronous operations, and it is widely used for making AJAX requests in web applications.

syntax: fetch(url);

## json():

The json() method in JavaScript promises is used to parse the response body of an HTTP request (typically a fetch request) as JSON. It is a convenient way to work with JSON data that is often received from web APIs or other server endpoints.

## async and await:

async and await are JavaScript language features introduced in ECMAScript 2017 (ES8) that simplify working with asynchronous code, making it more readable and easier to reason about. They are often used in conjunction with promises to write asynchronous code that appears more synchronous.

## async Function:

The async keyword is used to declare a function as asynchronous.

An async function returns a promise implicitly, whether you explicitly return a promise or not.

Inside an async function, you can use the await keyword to pause the execution of the function until a promise is resolved.

**Example:**

```
async function fetchData() {
  const response = await fetch("https://example.com/api/data");
  const data = await response.json();
  return data;  }
```

### await Operator:

The await keyword can only be used inside an async function.

It is used before a promise to pause the function's execution until the promise is resolved.

When used with a promise, await returns the resolved value of the promise.

If the promise is rejected, it throws an error that can be caught using a try...catch block or by chaining a .catch().

### Error Handling:

You can handle errors in async functions using traditional try...catch blocks or by chaining a .catch() after an await expression.

If an error occurs in any async function, it will reject the promise returned by the function.

### Benefits:

async and await make asynchronous code look more like synchronous code, improving readability and maintainability.

They simplify error handling, as you can use standard try...catch blocks for error handling.

They help avoid "callback hell" (also known as "pyramid of doom"), which can occur with deeply nested callbacks in asynchronous code.

### Limitations:

await can only be used inside async functions.

async functions always return promises, even if they return a non-promise value.

### Use Cases:

async and await are used for any asynchronous operation, such as making network requests, reading files, or working with databases.

They are commonly used with promises and other asynchronous patterns to write more readable and maintainable code.

### try and catch block:

In JavaScript, the try and catch blocks are used for error handling, allowing you to handle and gracefully recover from errors or exceptions that may occur during the execution of your code. Additionally, JavaScript provides the Error object and its various subtypes to represent and work with errors in a structured manner.

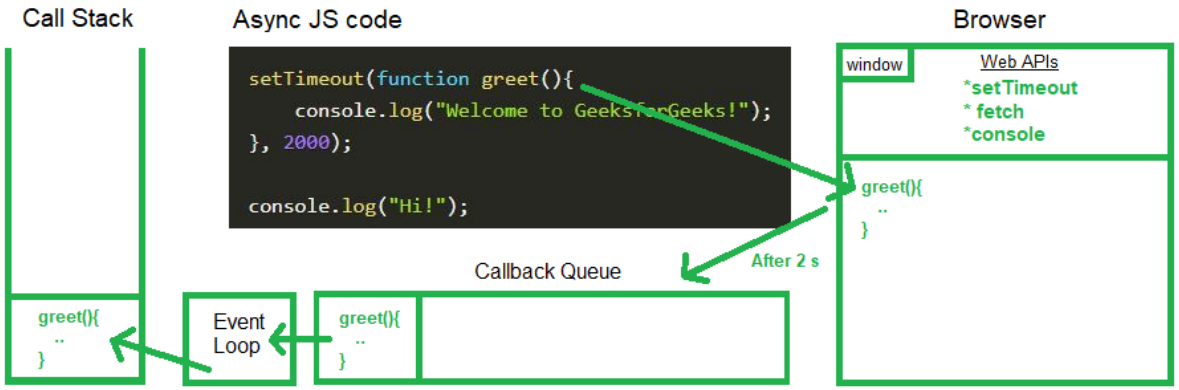
**The try and catch blocks are used together to handle exceptions:**

The try block contains the code where you anticipate an error might occur.

The catch block contains code to handle the error if it occurs.

If an error occurs in the try block, the code in the catch block is executed.

## How Asynchronous code works?




Asynchronous code execution in JavaScript is a fundamental concept that allows you to perform tasks that might take time to complete without blocking the main execution thread.

**setTimeout and setInterval:**

When you call setTimeout or setInterval, the provided callback function (or code) is not executed immediately. Instead, it's added to the browser's Web API environment, and a timer is set.

After the specified delay for setTimeout or at each interval for setInterval, the callback is moved to the callback queue.

The JavaScript engine, while executing code on the main thread, checks the callback queue during each cycle of the event loop to see if there are any functions waiting to be executed.

When the callback reaches the front of the queue and the main thread is free (not executing any other code), the callback is executed.

### Callback queue:

A callback queue is a queue of tasks that are executed after the current task. The callback queue is handled by the JavaScript engine after it has executed all tasks in the microtask queue.

### Microtask Queue:

Microtask Queue is like the Callback Queue, but Microtask Queue has higher priority. All the callback functions coming through Promises and Mutation Observer will go inside the Microtask Queue. For example, in the case of . fetch(), the callback function gets to the Microtask Queue.

### Event loop:

- The event loop is a continuous process that continually checks the state of the callback queue while the execution stack is empty.
- If the execution stack is empty, the event loop takes the first task (callback) from the callback queue and pushes it onto the execution stack.
- The callback function is then executed in the order it was added to the queue.

## Prototypes:

Prototype inheritance is a fundamental concept in JavaScript that enables object-oriented programming and code reusability. In JavaScript, all objects have a prototype, and they can inherit properties and methods from their prototype object.

Every JavaScript object has a prototype, which is another object.

The prototype object contains properties and methods that can be inherited by other objects.

## Prototype Chain:

When you access a property or method on an object, JavaScript first checks if that property or method exists directly on the object.

If it doesn't find the property or method, JavaScript looks for it in the object's prototype.

This process continues up the prototype chain until the property or method is found or the chain ends at the global Object.prototype.

## prototype Property:

Functions in JavaScript have a special property called prototype.

When you create an object using a constructor function, the object's prototype is set to the constructor's prototype property.

You can add properties and methods to the constructor's prototype, and they will be inherited by all objects created from that constructor.

## Modules in JS:

JavaScript modules are a way to organize and encapsulate code into reusable and maintainable units. They provide a mechanism for breaking up your JavaScript code into smaller, self-contained files or modules, each with its own scope. Modules help improve code structure, promote code reusability, and prevent global scope pollution. In modern JavaScript, there are different module systems, including CommonJS, AMD (Asynchronous Module Definition), and ES6 (ECMAScript 2015) modules. The most commonly used module system in modern web development is ES6 modules.

## ES6 Modules (ESM):

ES6 modules are the official standard for module management in JavaScript, and they are supported natively in modern browsers and in Node.js (since Node.js v13.2.0).

**1.Named Export:**

**Exporting from a Module:**

To export values (variables, functions, classes, etc.) from a module, you use the export keyword.

**Importing from a Module:**

To import values from a module into another module, you use the import statement.

**Example:**

**Export:**

```
// myModule.js
export const myVariable = 42;


export function myFunction() {
  // ...
}


export class MyClass {
  // ...
}
```

**import:**

```
// anotherModule.js
import { myVariable, myFunction, MyClass } from './myModule.js';


console.log(myVariable); // 42
myFunction();
const instance = new MyClass();
```

While importing the name should be same.We can able to import more than one object/function/variables using named export.

### Default Exports:

You can export a default value from a module using the export default syntax.

**Export:**

```
// myModule.js
const defaultExport = 42;
export default defaultExport;
```

**import:**

```
// anotherModule.js
import myDefaultExport from './myModule.js';
```

```
console.log(myDefaultExport); // 42
```

we can able to import only one object/function/variable using default export and name need not to be same in the original js file.

### CommonJS Modules (CJS):

CommonJS modules are primarily used in Node.js for server-side JavaScript development. They use require() to import modules and module.exports to export values.

### Example:

Exporting:

```
// myModule.js

const myVariable = 42;

function myFunction() {
  // ...
}

module.exports = {
  myVariable,
  myFunction,
};
```

Importing from a Module:

```
// anotherModule.js

const { myVariable, myFunction } = require('./myModule');

console.log(myVariable); // 42

myFunction();
```

### Arguments Object:

The arguments object is an array-like object available within the body of every function in JavaScript.

It allows you to access the arguments that were passed to the function, even if the function didn't explicitly declare named parameters.

You can access the arguments using numeric indices (e.g., arguments[0], arguments[1], etc.).

The arguments object is not a true array; it is an array-like object, so it lacks array methods like map, forEach, and reduce. You can convert it to a real array using methods like Array.from(arguments) or the spread operator ([...arguments]).

### Arity of a Function:

The term "arity" refers to the number of arguments a function accepts.

For example, a function that takes no arguments has an arity of 0, a function that takes one argument has an arity of 1, and so on.

In JavaScript, "unary," "binary," and "ternary" functions refer to functions that accept a specific number of arguments: one, two, and three, respectively. These terms are often used to describe the arity (the number of arguments) of a function.

**Unary Function:**

A unary function is a function that accepts one argument.

Unary functions are quite common in JavaScript, as many built-in functions and methods take a single argument.

**Binary Function:**

A binary function is a function that accepts two arguments.

Binary functions are also common in JavaScript and are often used for operations that involve two values.

**Ternary Function:**

A ternary function is a function that accepts three arguments.

Ternary functions are less common than unary or binary functions and are typically used for specific scenarios where three arguments are needed.

**n-ary function:**

An "n-ary" function in JavaScript is a function that accepts a variable number of arguments. Unlike unary (1 argument), binary (2 arguments), or ternary (3 arguments) functions, an n-ary function can take any number of arguments, including zero.

## JS Classes:

JavaScript classes are a way to create objects with shared properties and methods, following a blueprint or template. They were introduced in ECMAScript 2015 (ES6) and provide a more structured and object-oriented approach to creating and managing objects in JavaScript. Classes in JavaScript are a fundamental part of modern web development. Here's an overview of how classes work in JavaScript

## Class Declaration:

You can declare a class using the class keyword, followed by the class name. The class can have a constructor method that is called when you create a new instance of the class. Class methods can also be defined within the class.

**EX:**

```
class MyClass {
  constructor(property1, property2) {
    this.property1 = property1;
    this.property2 = property2;  }
```

```
  myMethod() {
    // Method logic
  }
}
```

**Creating Instances:**

To create instances (objects) of a class, you use the new keyword followed by the class name, passing any required arguments to the constructor.

EX: const myObject = new MyClass("value1", "value2");

**Constructor:**

The constructor method is a special method called when a new instance of the class is created. It is used to initialize the object's properties.

**EX:**

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

**Methods:**

You can define methods within a class. These methods can be called on instances of the class.

**EX:**

```
class Circle {
  constructor(radius) {
    this.radius = radius;
  }
  getArea() {
    return Math.PI * this.radius ** 2;
  }
}
const myCircle = new Circle(5);
console.log(myCircle.getArea()); // Output: 78.53981633974483
```

**Inheritance:**

JavaScript classes support inheritance, allowing you to create a subclass (child class) that inherits properties and methods from a superclass (parent class).

**EX:**

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  makeSound() {
    console.log("Animal makes a sound");
  }
}
class Dog extends Animal {
  makeSound() {
    console.log("Dog barks");
  }
}
const myDog = new Dog("Buddy");
myDog.makeSound(); // Output: "Dog barks"
```

**Static Methods:**

Static methods are defined on the class itself, not on instances. They are called on the class, not on instances of the class.

**EX:**

```
class MathUtils {
  static add(x, y) {
    return x + y;
  }
  static subtract(x, y) {
    return x - y;
  }
}
console.log(MathUtils.add(5, 3));      // Output: 8
```

```
console.log(MathUtils.subtract(10, 4)); // Output: 6
```

**Private Class Fields:**

In newer JavaScript versions (ES2022 and beyond), you can use private class fields to create private variables within a class.

**EX:**

```
class Example {
  #privateField = 42;
  getPrivateField() {
    return this.#privateField;
  }
}
```