**1. What is the difference between var, let, and const?**

* **var** declarations are globally scoped or function scoped while **let** and **const** are block scoped.
* **var** variables can be updated and re-declared within its scope; **let** variables can be updated but not re-declared; **const** variables can neither be updated nor re-declared.
* They are all hoisted to the top of their scope. But while var variables are initialized with undefined, let and const variables are not initialized.
* While var and let can be declared without being initialized, const must be initialized during declaration.

**2.Hoisting**

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope (global or function scope) during the compilation phase, before the code is executed. This allows you to use functions and variables even before they appear in the code.

* **Function Declarations are fully hoisted:**

This means that you can call a function even before you declare it in your code. The entire function body is moved to the top of the scope.

A black screen with white text

Description automatically generated

* **Variable Declarations are hoisted, but not their initializations**

The declaration of the variable (i.e., the fact that it exists) is hoisted, but the assignment of the value happens at the original place in the code. Variables declared with var are initialized with undefined by default until they reach the line of assignment.

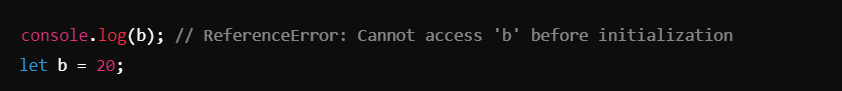
A black screen with white text

Description automatically generated

Here, var a is hoisted to the top, but a = 10 is not. That's why the first console.log(a) prints undefined.

* **let and const have a temporal dead zone (TDZ):**

Variables declared with let and const are also hoisted, but they are not initialized until their declaration is encountered. Accessing them before declaration results in a ReferenceError due to the TDZ.



* **Function Expressions are not hoisted:**

Only the variable declaration is hoisted, not the function definition when using a function expression.

A black screen with white text

Description automatically generated

**3. The difference between null and undefiend**

* **Undefined**
  + Meaning: A variable is declared but hasn't been assigned any value yet.
  + When it's used.
    - When you declare a variable without initializing it, it will be undefined
    - Functions return undefined if no return statement is provided.
    - It represents the absence of a value implicitly (i.e., JavaScript assigns it automatically).
* **Null**
  + Meaning: null represents the intentional absence of any object value.
  + When it's used: It is explicitly assigned by developers to indicate "no value" or "empty value."
  + Intentional: null is typically used when you want to purposely clear a variable or indicate that a variable should hold no value.
* **Key Differences:**
  + Type:
    - undefined is a type itself in JavaScript.
    - null is an object.
  + Assignment:
    - undefined means a variable hasn't been assigned a value.
    - null is explicitly assigned as a value representing "nothing."

**4. Difference between library and framework**

Library and framework are two common terms in software development, but they differ in how they interact with your code and how they influence the development process.

* **Library:**
  + A library is a collection of pre-written code that developers can use to perform common tasks. Libraries are typically smaller and more focused on solving specific problems or adding specific functionality.
  + Key characteristic: You are in control of the flow of the application, and you choose when and how to use the library. It provides tools, but your code calls the library.
* **Framework:**
  + A framework is a comprehensive platform that provides a structure for building applications. It offers a broader foundation with built-in tools and patterns, managing the flow and structure of your application. A framework often dictates how you organize and write your code.
  + Key characteristic: The framework is in control of the flow, meaning your code is integrated into the framework, which determines when and how certain parts of your code are executed. In other words, the framework "calls" your code.

A screenshot of a computer

Description automatically generated

* **React: Why It's a Library**
  + Minimal Prescriptive Structure: React is a library because it focuses on a specific task—building user interfaces (UIs)—without imposing a full application structure. It doesn’t tell you how to handle routing, state management, or API calls. You are free to choose additional libraries or patterns to handle other aspects of the app.
  + Control Over Application Logic: In React, you control the flow of the application. You decide when and how to use React components. It provides functions (hooks, state management, component lifecycle methods, etc.) that help manage UI but doesn’t dictate the complete architecture or how other parts of the application should be structured.
  + Composability: React is meant to be integrated with other tools. You can compose your application by picking different libraries (like Redux for state management, React Router for routing), giving you more freedom and flexibility in designing your app.
* **Laravel: Why It's a Framework**
  + Opinionated Structure: Laravel is a framework because it provides an opinionated and complete structure for building web applications. It comes with built-in features for routing, database management (Eloquent ORM), authentication, request handling, etc. The framework gives you a full stack of tools to build and manage the application from start to finish.
  + Inversion of Control: In a framework like Laravel, the control flow is inverted: instead of you deciding how things flow, the framework calls your code at specific points (e.g., controllers, middleware). It tells you how to structure your application, and you work within its conventions.
  + Batteries Included: Laravel provides almost everything you need to build a web application—authentication, database abstraction, request/response handling, templating, email handling, etc. It’s a full solution, whereas React focuses specifically on the UI layer.

**5.what is DOM(document object model)**

The DOM, or Document Object Model, is a programming interface for web documents. It represents the structure of an HTML or XML document as a tree of nodes, where each node corresponds to a part of the document such as elements, attributes, and text. With the DOM, developers can interact with the content and structure of a webpage dynamically—modifying, adding, or removing elements in real time.

For example, using JavaScript, I can access and manipulate the DOM to update the content of a page without requiring a full reload. This allows for interactive features like form validation, dynamic styling, or handling user events, making the web application more responsive and user-friendly.

In projects, I commonly use methods like getElementById, querySelector, and appendChild to interact with the DOM. Additionally, libraries like React simplify DOM manipulation by using a virtual DOM, which enhances performance by minimizing direct interaction with the real DOM.

Understanding how the DOM works is essential for tasks like debugging, ensuring compatibility across browsers, and improving the user experience."

**6.aynchronous javascript**

Asynchronous JavaScript refers to the ability of JavaScript to execute tasks asynchronously, meaning certain operations (like network requests, file reading, timers, etc.) can run in the background without blocking the main execution thread. This allows the program to remain responsive and handle other operations while waiting for those tasks to complete.

In synchronous programming, each task must be completed before the next one starts, which can cause delays, especially when performing time-consuming tasks like fetching data from a server. In contrast, asynchronous programming allows JavaScript to continue executing other tasks while waiting for these slower operations to finish.

Asynchronous JavaScript allows code to run in a non-blocking way, which means that while waiting for long-running operations like network requests, database queries, or file handling, other code can still execute. This prevents "blocking" or freezing the execution of the program. Asynchronous JavaScript can be handled through callbacks, promises, and the modern async/await syntax.

* **Promises**

A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation. It allows you to write asynchronous code in a more structured way, compared to traditional callbacks. A Promise can be in one of three states:

1. **Pending**: The initial state, where the operation has not completed yet.
2. **Fulfilled:** The operation completed successfully, and a result is available.
3. **Rejected**: The operation failed, and an error message is available.

A Promise has two main methods:

1. **then():** Runs when the promise is fulfilled (i.e., the operation succeeded).
2. **catch():** Runs when the promise is rejected (i.e., the operation failed)

A screen shot of a computer screen

Description automatically generated

* **Async/Await**

Async/Await is built on top of promises and provides a more readable and concise syntax for handling asynchronous operations, resembling synchronous code:

1. **async:** Marks a function as asynchronous. This function will always return a Promise.
2. **await:** Pauses the execution of the function until the promise is resolved or rejected. It must be used inside an async function.
   * An async function always returns a promise. If the function contains a value or an explicit return, that value is wrapped in a resolved promise.
   * The await expression pauses the execution of the async function and waits for the promise to resolve.
   * Once the promise resolves (either fulfilled or rejected), the execution of the function continues from where the await was placed.
   * If the promise is rejected, you can handle the error using a try/catch block.

* **Callback**

A **callback** is a function that is passed as an argument to another function and is executed after some operation has been completed. It's a way to ensure that a certain block of code runs only after another block has finished executing, often in asynchronous programming.

In simpler terms, a callback is a function that is "called back" later after some task is done

A screen shot of a computer program

Description automatically generated

**How It Works**

1. fetchData is called and starts executing.
2. Inside fetchData, there's a setTimeout that simulates an asynchronous operation.
3. After 2 seconds, setTimeout finishes, and it calls the callback function (which is processData in this case).
4. The processData function is executed after the data is fetched.

* **Callstack**

In JavaScript, the call stack is like a to-do list for functions in your program. It follows the rule of "Last In, First Out," meaning the last thing added is the first to be done.

A call stack is like a script's roadmap for a JavaScript Engine. It helps the JavaScript Engine to keep track of which function is currently running and which functions are called from within that function. It's basically a way for the JavaScript Engine to navigate through a script with multiple functions.

**Purpose of Call Stack in JavaScript**

The call stack in JavaScript has an important job. It keeps track of the order in which functions are called and manages the context of each function's execution. Here's what it does:

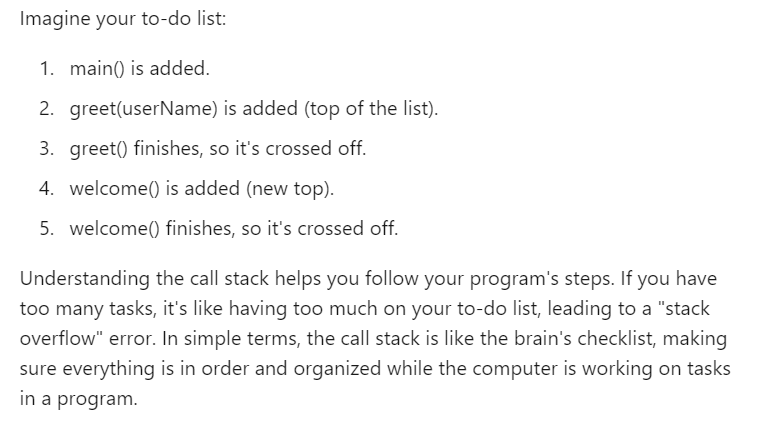
1. **Remembering where to go back:** When a function is called, the call stack remembers where to go back to when that function is done. It's like noting the page number in a book.
2. **Keeping track of local stuff:** Each function has its own set of special things it's using, like variables and information. The call stack keeps track of these so that each function gets what it needs.
3. **Handling repeat tasks:** If a function calls itself (which is called recursion), the call stack is crucial. It keeps track of all the times the function is called, like making a list of all the times you play a game.
4. **Managing computer memory:** The call stack helps the computer use memory efficiently. It keeps track of which functions are active and which ones are done, so the computer can clean up and use memory wisely.

**How Call Stack Works in JavaScript**

1. **Function Calls:** When you call a function, it's like adding a task to the top of your to-do list (the call stack). The task includes details about the function, like what it needs to do.
2. **Execution:** The code in the function runs step by step. If the function calls another function, that new task is added to the top of the list, and you focus on it.
3. **Return:** When a function finishes its job, it's like crossing off the task at the top. Control goes back to the previous task on the list.

A screenshot of a computer program

Description automatically generated



**7.Closures**

A closure in JavaScript is a function that remembers the variables from its outer scope, even after the outer function has finished executing. This happens because JavaScript allows inner functions to access the variables of their parent function, creating a 'closed' environment, or closure. Closures are often used for data privacy, callback functions, and to maintain state.

A screen shot of a computer program

Description automatically generated