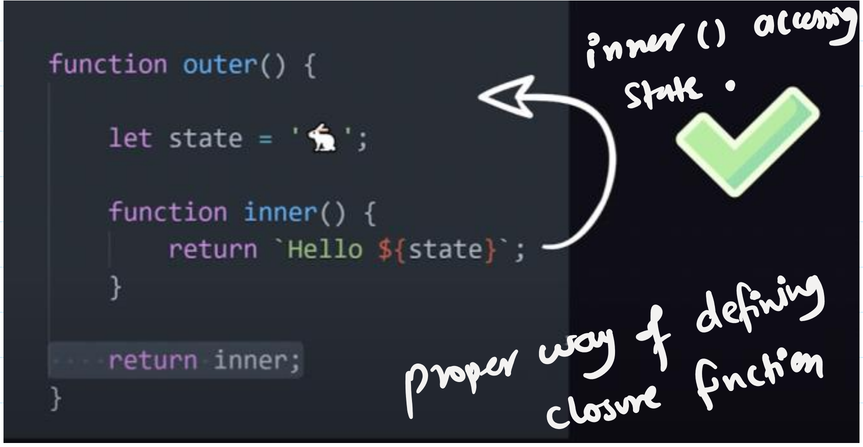
function 02

Saturday, March 23, 2024 2:10 PM

- closure function → function accessing variables/ data outside of Its scope.



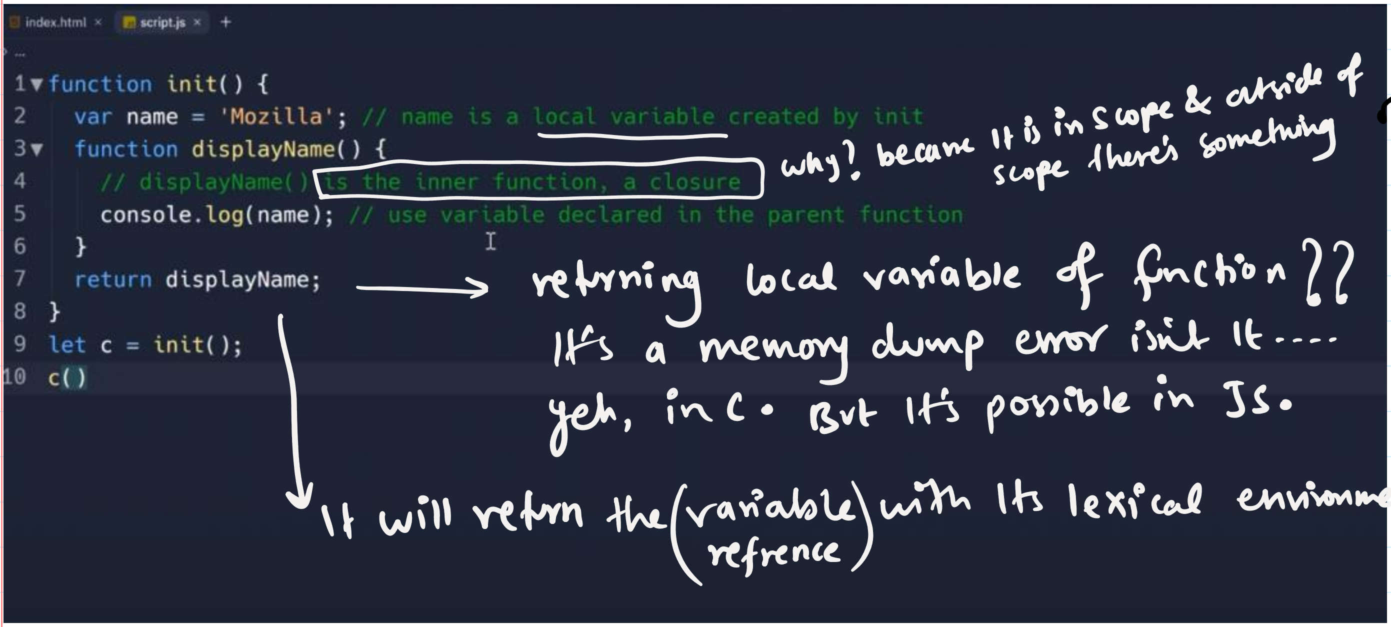
```javascript
function outer() {

    let state = '🐇';

    function inner() {
        return `Hello ${state}`;
    }

    return inner;
}
```

inner () accessing state.

✓

proper way of defining closure function

I'll Show you a magic Trick now.



```javascript
function init() {
  var name = 'Mozilla'; // name is a local variable created by init
  function displayName() {
    // displayName() is the inner function, a closure
    console.log(name); // use variable declared in the parent function
  }
  return displayName;
}
let c = init();
c()
```

why?. because It is in Scope & outside of other too scope there's something

→ returning local variable of function??

It's a memory dump error isn't It....

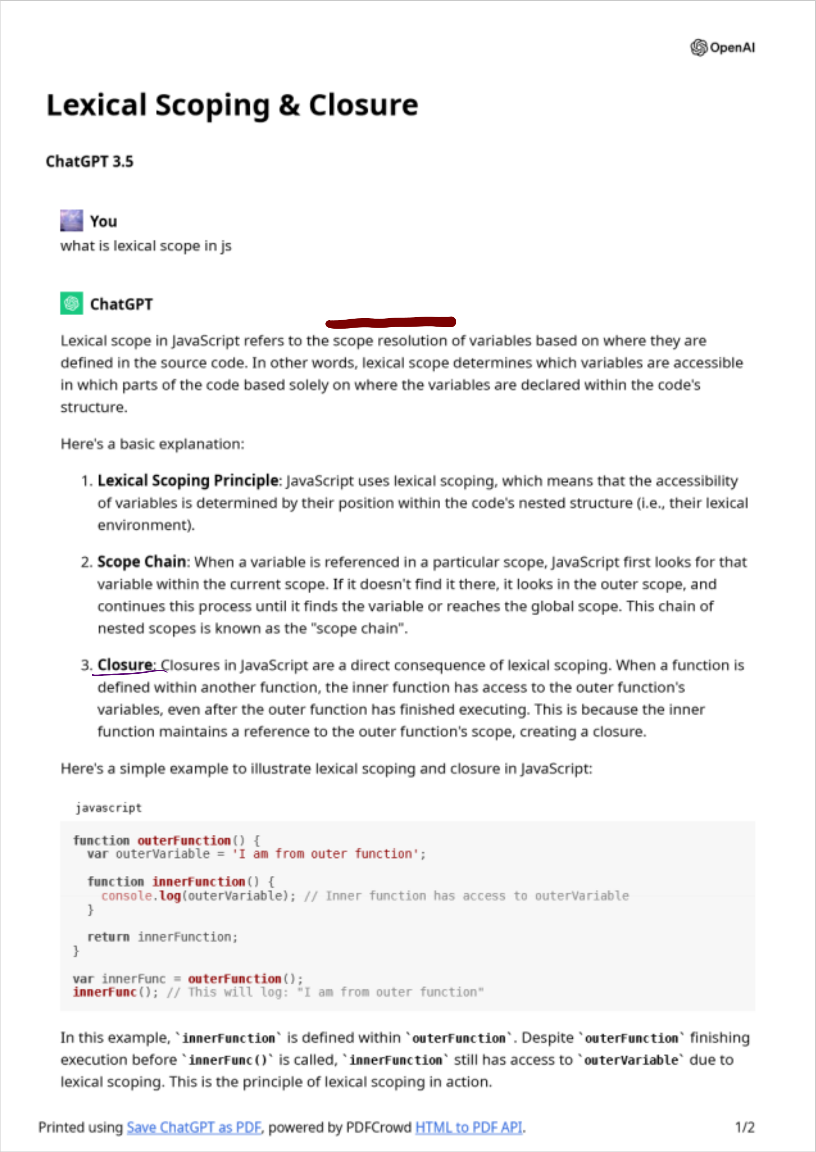
yeh, in C. But It's possible in JS.

It will return the (variable refrence) with Its lexical environment

- what is lexical Environment then?

https://chat.openai.com/share/252a9fff-0563-431c-9364-cb2e4fb5a357

Lexical Scoping &...



OpenAI

Lexical Scoping & Closure

ChatGPT 3.5

You

what is lexical scope in js

ChatGPT

Lexical scope in JavaScript refers to the scope resolution of variables based on where they are defined in the source code. In other words, lexical scope determines which variables are accessible in which parts of the code based solely on where the variables are declared within the code's structure.

Here's a basic explanation:

1. **Lexical Scoping Principle**: JavaScript uses lexical scoping, which means that the accessibility of variables is determined by their position within the code's nested structure (i.e., their lexical environment).
2. **Scope Chain**: When a variable is referenced in a particular scope, JavaScript first looks for that variable within the current scope. If it doesn't find it there, it looks in the outer scope, and continues this process until it finds the variable or reaches the global scope. This chain of nested scopes is known as the "scope chain".
3. **Closure**: Closures in JavaScript are a direct consequence of lexical scoping. When a function is defined within another function, the inner function has access to the outer function's variables, even after the outer function has finished executing. This is because the inner function maintains a reference to the outer function's scope, creating a closure.

Here's a simple example to illustrate lexical scoping and closure in JavaScript:

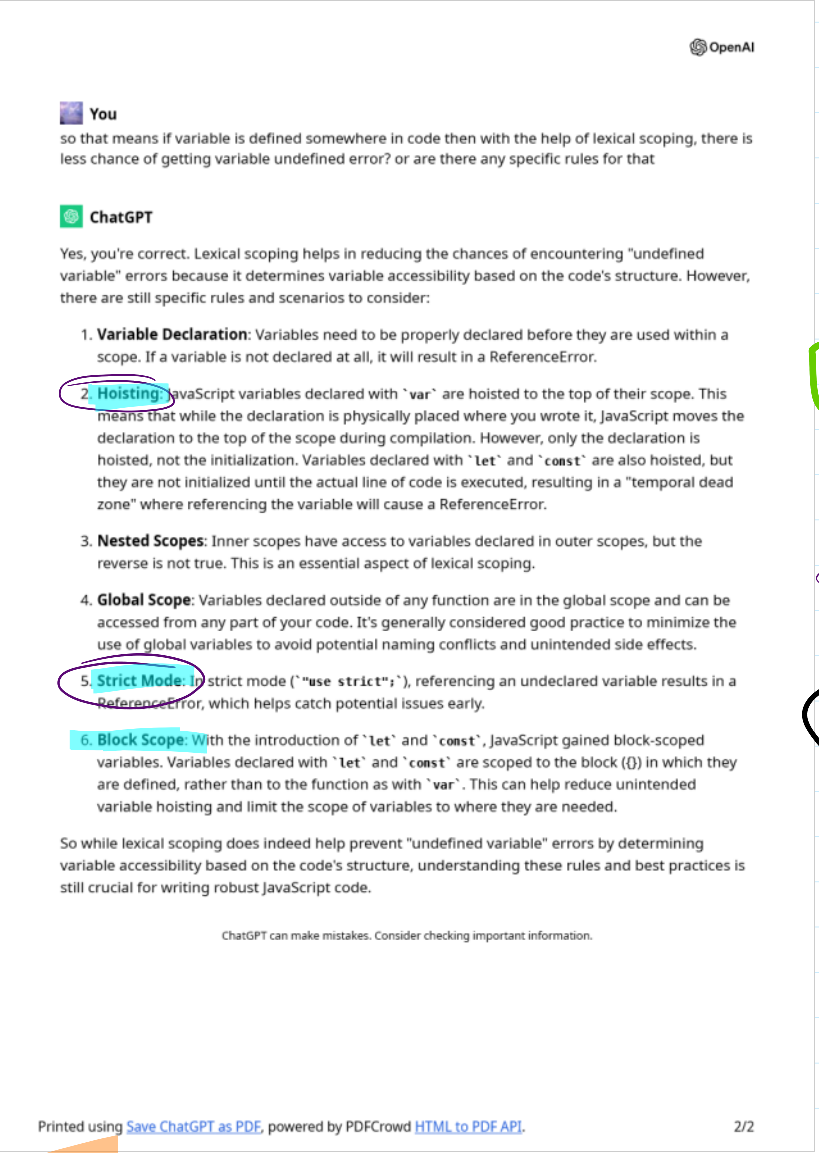
javascript

```javascript
function outerFunction() {
  var outerVariable = 'I am from outer function';

  function innerFunction() {
    console.log(outerVariable); // Inner function has access to outerVariable
  }

  return innerFunction;
}

var innerFunc = outerFunction();
innerFunc(); // This will log: "I am from outer function"
```

In this example, `innerFunction` is defined within `outerFunction`. Despite `outerFunction` finishing execution before `innerFunc()` is called, `innerFunction` still has access to `outerVariable` due to lexical scoping. This is the principle of lexical scoping in action.

Printed using Save ChatGPT as PDF, powered by PDFCrowd HTML to PDF API. 1/2

closure



OpenAI

You

so that means if variable is defined somewhere in code then with the help of lexical scoping, there is less chance of getting variable undefined error? or are there any specific rules for that

ChatGPT

Yes, you're correct. Lexical scoping helps in reducing the chances of encountering "undefined variable" errors because it determines variable accessibility based on the code's structure. However, there are still specific rules and scenarios to consider:

1. **Variable Declaration**: Variables need to be properly declared before they are used within a scope. If a variable is not declared at all, it will result in a ReferenceError.
2. **Hoisting**: JavaScript variables declared with `var` are hoisted to the top of their scope. This means that while the declaration is physically placed where you wrote it, JavaScript moves the declaration to the top of the scope during compilation. However, only the declaration is hoisted, not the initialization. Variables declared with `let` and `const` are also hoisted, but they are not initialized until the actual line of code is executed, resulting in a "temporal dead zone" where referencing the variable will cause a ReferenceError.
3. **Nested Scopes**: Inner scopes have access to variables declared in outer scopes, but the reverse is not true. This is an essential aspect of lexical scoping.
4. **Global Scope**: Variables declared outside of any function are in the global scope and can be accessed from any part of your code. It's generally considered good practice to minimize the use of global variables to avoid potential naming conflicts and unintended side effects.
5. **Strict Mode**: In strict mode (`"use strict";`), referencing an undeclared variable results in a ReferenceError, which helps catch potential issues early.
6. **Block Scope**: With the introduction of `let` and `const`, JavaScript gained block-scoped variables. Variables declared with `let` and `const` are scoped to the block ({}) in which they are defined, rather than to the function as with `var`. This can help reduce unintended variable hoisting and limit the scope of variables to where they are needed.

So while lexical scoping does indeed help prevent "undefined variable" errors by determining variable accessibility based on the code's structure, understanding these rules and best practices is still crucial for writing robust JavaScript code.

ChatGPT can make mistakes. Consider checking important information.

Printed using Save ChatGPT as PDF, powered by PDFCrowd HTML to PDF API. 2/2

Hosting :-> how variables & functions are processed during the compilation phase.

} avoid global scope as much as possible

Strict mode :- learn to write code in Strict mode.

① Strict mode in entire file

⑪ strict mode in function only.

①
```
"use strict";
```
using in 1st line of code in page.

⑪
```
function add(){
  "use strict";
    return a+b;
}
```
using in 1st line of code inside f(x)

Block Scope

{ } → block

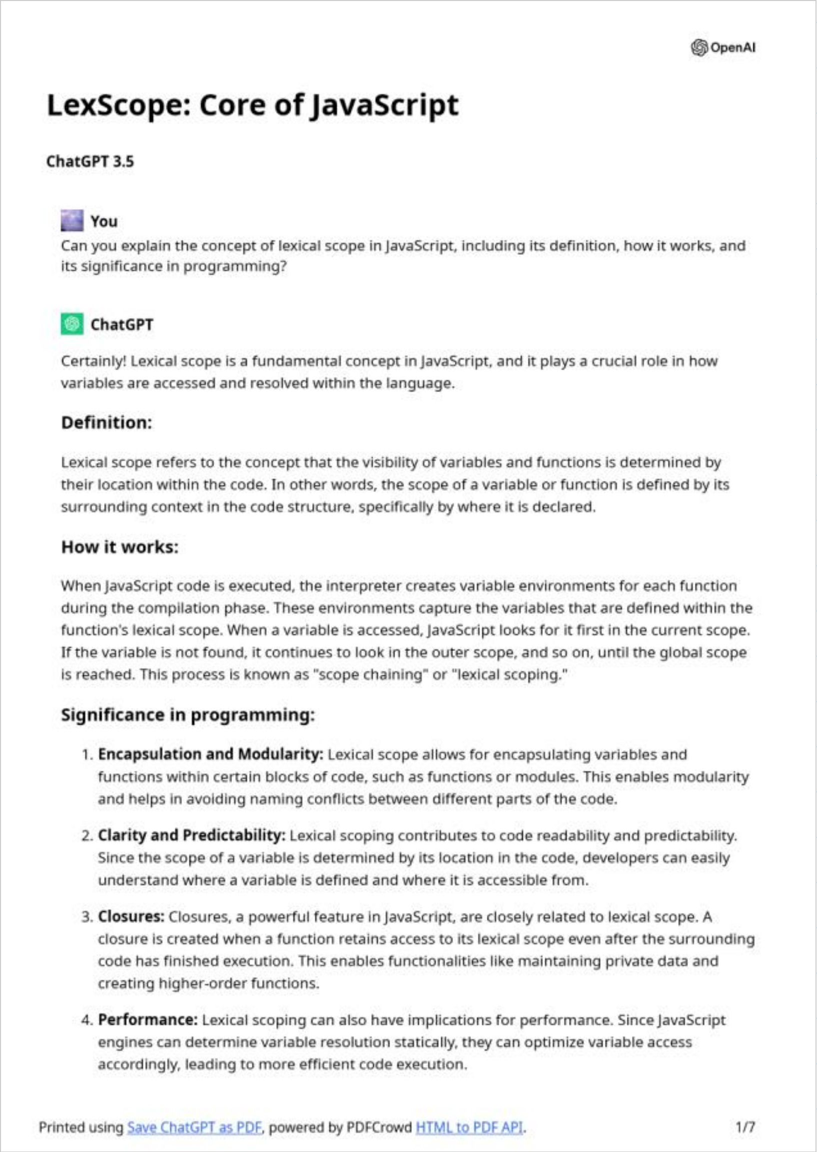
→ variable declared is only accesible in that block.

* let, Const works in Block scope

* var → do not works in block Scope, It creates globally.

Discussed below ↓

LexScope_ Core of Ja...



LexScope: Core of JavaScript

ChatGPT 3.5

You

Can you explain the concept of lexical scope in JavaScript, including its definition, how it works, and its significance in programming?

ChatGPT

Certainly! Lexical scope is a fundamental concept in JavaScript, and it plays a crucial role in how variables are accessed and resolved within the language.

Definition:

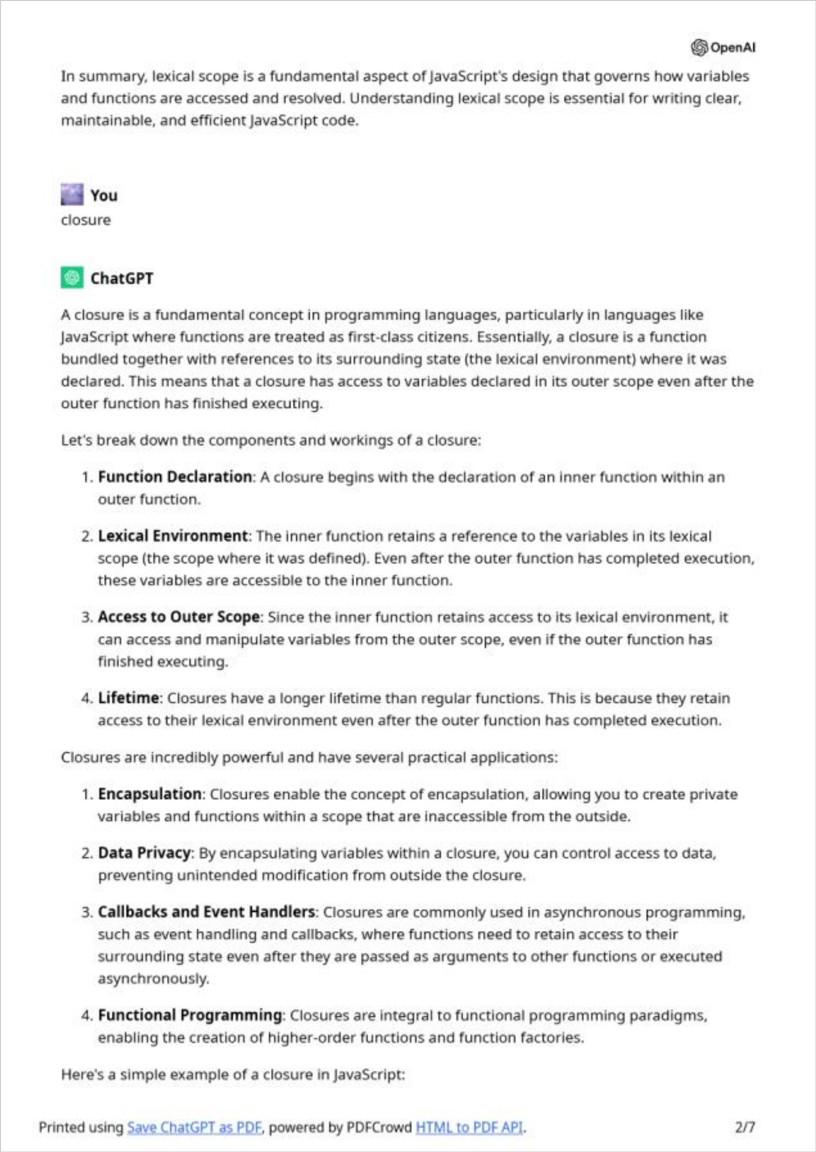
Lexical scope refers to the concept that the visibility of variables and functions is determined by their location within the code. In other words, the scope of a variable or function is defined by its surrounding context in the code structure, specifically by where it is declared.

How it works:

When JavaScript code is executed, the interpreter creates variable environments for each function during the compilation phase. These environments capture the variables that are defined within the function's lexical scope. When a variable is accessed, JavaScript looks for it first in the current scope. If the variable is not found, it continues to look in the outer scope, and so on, until the global scope is reached. This process is known as "scope chaining" or "lexical scoping."

Significance in programming:

1. **Encapsulation and Modularity:** Lexical scope allows for encapsulating variables and functions within certain blocks of code, such as functions or modules. This enables modularity and helps in avoiding naming conflicts between different parts of the code.
2. **Clarity and Predictability:** Lexical scoping contributes to code readability and predictability. Since the scope of a variable is determined by its location in the code, developers can easily understand where a variable is defined and where it is accessible from.
3. **Closures:** Closures, a powerful feature in JavaScript, are closely related to lexical scope. A closure is created when a function retains access to its lexical scope even after the surrounding code has finished execution. This enables functionalities like maintaining private data and creating higher-order functions.
4. **Performance:** Lexical scoping can also have implications for performance. Since JavaScript engines can determine variable resolution statically, they can optimize variable access accordingly, leading to more efficient code execution.



In summary, lexical scope is a fundamental aspect of JavaScript's design that governs how variables and functions are accessed and resolved. Understanding lexical scope is essential for writing clear, maintainable, and efficient JavaScript code.

You

closure

ChatGPT

A closure is a fundamental concept in programming languages, particularly in languages like JavaScript where functions are treated as first-class citizens. Essentially, a closure is a function bundled together with references to its surrounding state (the lexical environment) where it was declared. This means that a closure has access to variables declared in its outer scope even after the outer function has finished executing.

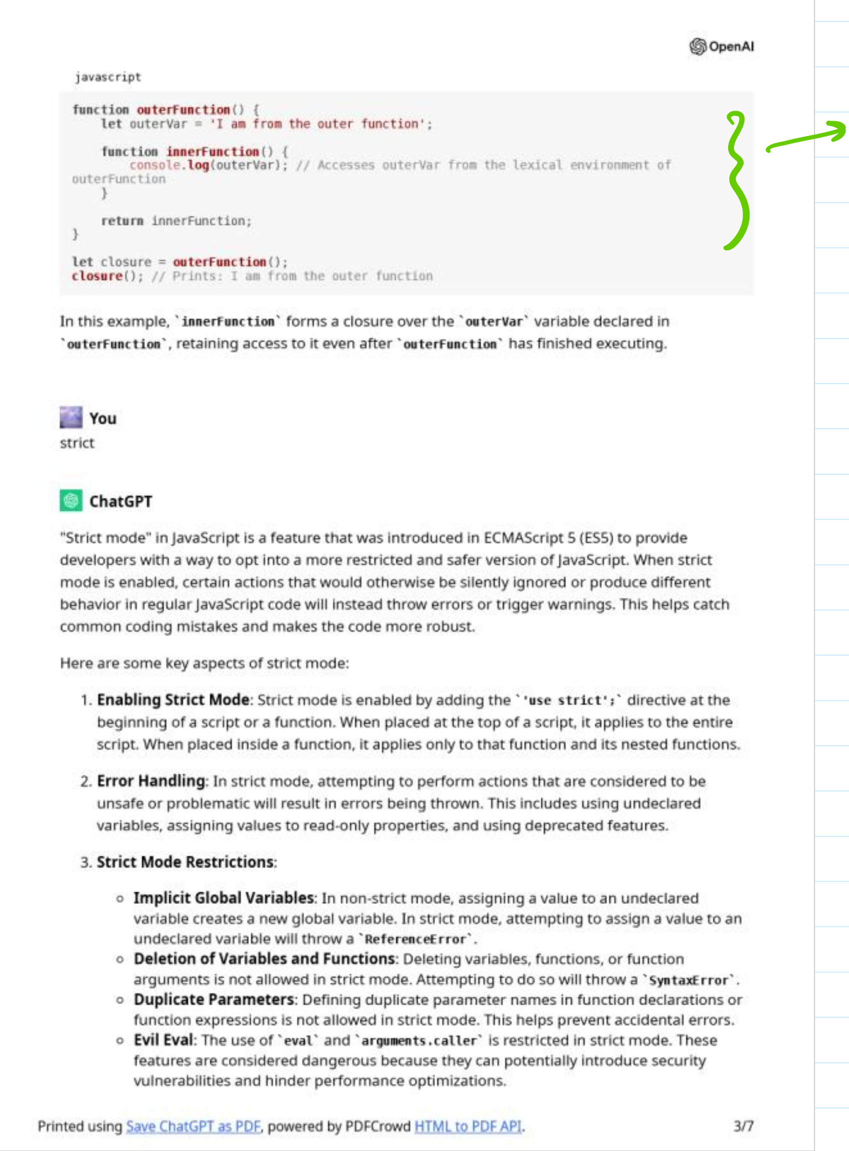
Let's break down the components and workings of a closure:

1. **Function Declaration**: A closure begins with the declaration of an inner function within an outer function.
2. **Lexical Environment**: The inner function retains a reference to the variables in its lexical scope (the scope where it was defined). Even after the outer function has completed execution, these variables are accessible to the inner function.
3. **Access to Outer Scope**: Since the inner function retains access to its lexical environment, it can access and manipulate variables from the outer scope, even if the outer function has finished executing.
4. **Lifetime**: Closures have a longer lifetime than regular functions. This is because they retain access to their lexical environment even after the outer function has completed execution.

Closures are incredibly powerful and have several practical applications:

1. **Encapsulation**: Closures enable the concept of encapsulation, allowing you to create private variables and functions within a scope that are inaccessible from the outside.
2. **Data Privacy**: By encapsulating variables within a closure, you can control access to data, preventing unintended modification from outside the closure.
3. **Callbacks and Event Handlers**: Closures are commonly used in asynchronous programming, such as event handling and callbacks, where functions need to retain access to their surrounding state even after they are passed as arguments to other functions or executed asynchronously.
4. **Functional Programming**: Closures are integral to functional programming paradigms, enabling the creation of higher-order functions and function factories.

Here's a simple example of a closure in JavaScript:



javascript

```javascript
function outerFunction() {
    let outerVar = 'I am from the outer function';

    function innerFunction() {
        console.log(outerVar); // Accesses outerVar from the lexical environment of
outerFunction
    }

    return innerFunction;
}

let closure = outerFunction();
closure(); // Prints: I am from the outer function
```

Something like python scoping thingo.

In this example, `` `innerFunction` `` forms a closure over the `` `outerVar` `` variable declared in `` `outerFunction` ``, retaining access to it even after `` `outerFunction` `` has finished executing.

You

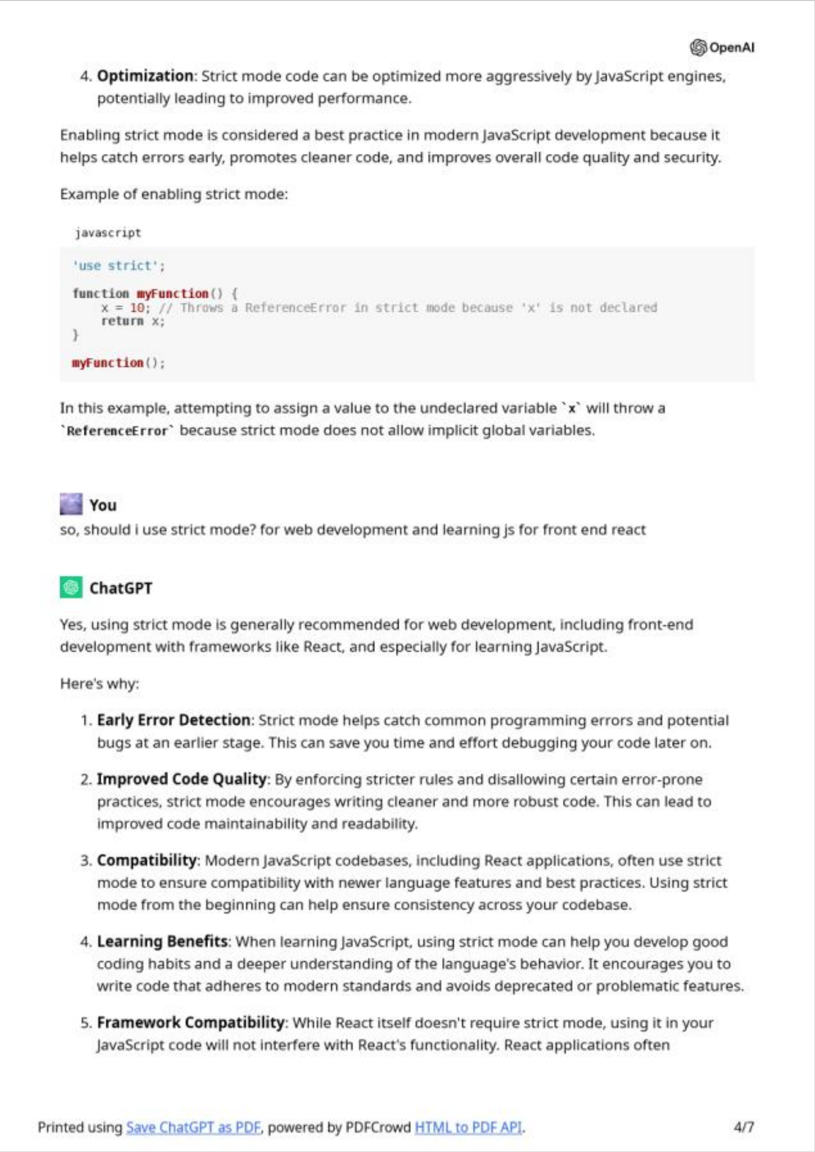
strict

ChatGPT

"Strict mode" in JavaScript is a feature that was introduced in ECMAScript 5 (ES5) to provide developers with a way to opt into a more restricted and safer version of JavaScript. When strict mode is enabled, certain actions that would otherwise be silently ignored or produce different behavior in regular JavaScript code will instead throw errors or trigger warnings. This helps catch common coding mistakes and makes the code more robust.

Here are some key aspects of strict mode:

1. **Enabling Strict Mode**: Strict mode is enabled by adding the `` `'use strict';` `` directive at the beginning of a script or a function. When placed at the top of a script, it applies to the entire script. When placed inside a function, it applies only to that function and its nested functions.
2. **Error Handling**: In strict mode, attempting to perform actions that are considered to be unsafe or problematic will result in errors being thrown. This includes using undeclared variables, assigning values to read-only properties, and using deprecated features.
3. **Strict Mode Restrictions**:
    - **Implicit Global Variables**: In non-strict mode, assigning a value to an undeclared variable creates a new global variable. In strict mode, attempting to assign a value to an undeclared variable will throw a `` `ReferenceError` ``.
    - **Deletion of Variables and Functions**: Deleting variables, functions, or function arguments is not allowed in strict mode. Attempting to do so will throw a `` `SyntaxError` ``.
    - **Duplicate Parameters**: Defining duplicate parameter names in function declarations or function expressions is not allowed in strict mode. This helps prevent accidental errors.
    - **Evil Eval**: The use of `` `eval` `` and `` `arguments.caller` `` is restricted in strict mode. These features are considered dangerous because they can potentially introduce security vulnerabilities and hinder performance optimizations.



4. **Optimization**: Strict mode code can be optimized more aggressively by JavaScript engines, potentially leading to improved performance.

Enabling strict mode is considered a best practice in modern JavaScript development because it helps catch errors early, promotes cleaner code, and improves overall code quality and security.

Example of enabling strict mode:

```javascript
'use strict';

function myFunction() {
    x = 10; // Throws a ReferenceError in strict mode because 'x' is not declared
    return x;
}

myFunction();
```

In this example, attempting to assign a value to the undeclared variable `x` will throw a `ReferenceError` because strict mode does not allow implicit global variables.

You

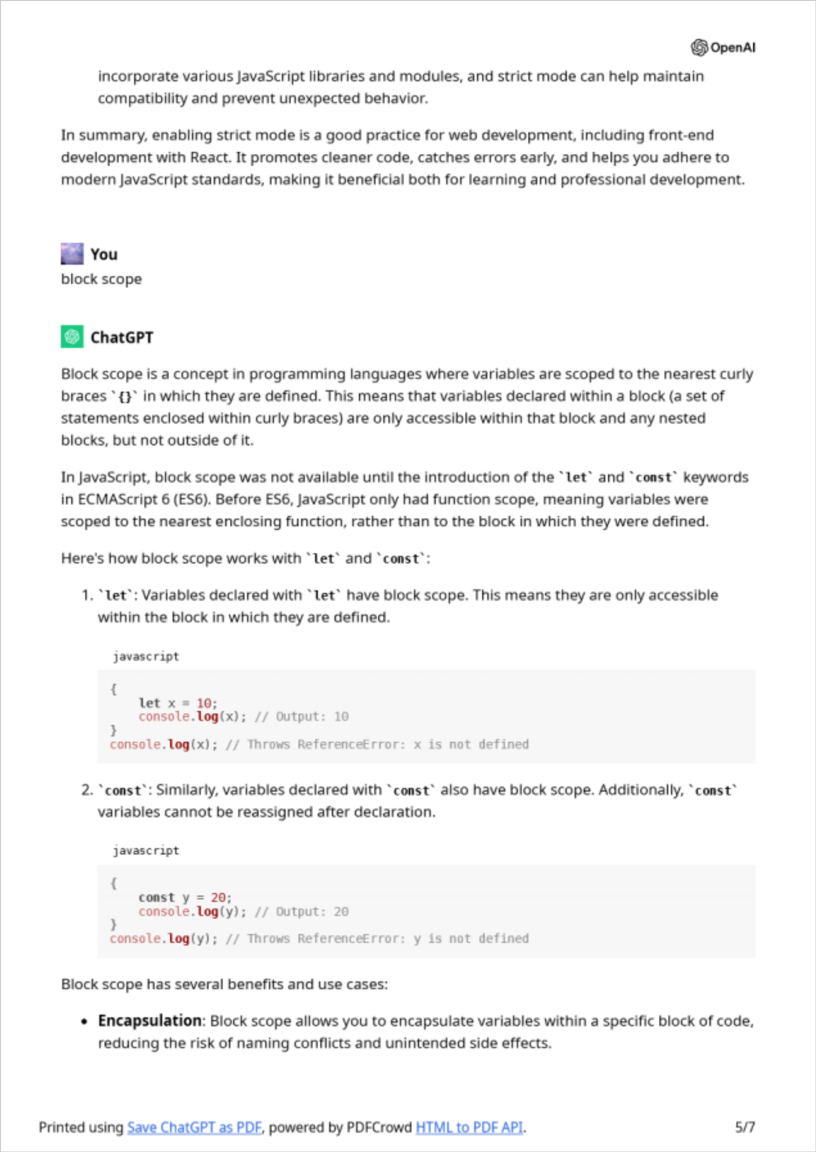
so, should i use strict mode? for web development and learning js for front end react

ChatGPT

Yes, using strict mode is generally recommended for web development, including front-end development with frameworks like React, and especially for learning JavaScript.

Here's why:

1. **Early Error Detection**: Strict mode helps catch common programming errors and potential bugs at an earlier stage. This can save you time and effort debugging your code later on.
2. **Improved Code Quality**: By enforcing stricter rules and disallowing certain error-prone practices, strict mode encourages writing cleaner and more robust code. This can lead to improved code maintainability and readability.
3. **Compatibility**: Modern JavaScript codebases, including React applications, often use strict mode to ensure compatibility with newer language features and best practices. Using strict mode from the beginning can help ensure consistency across your codebase.
4. **Learning Benefits**: When learning JavaScript, using strict mode can help you develop good coding habits and a deeper understanding of the language's behavior. It encourages you to write code that adheres to modern standards and avoids deprecated or problematic features.
5. **Framework Compatibility**: While React itself doesn't require strict mode, using it in your JavaScript code will not interfere with React's functionality. React applications often



incorporate various JavaScript libraries and modules, and strict mode can help maintain compatibility and prevent unexpected behavior.

In summary, enabling strict mode is a good practice for web development, including front-end development with React. It promotes cleaner code, catches errors early, and helps you adhere to modern JavaScript standards, making it beneficial both for learning and professional development.

You

block scope

ChatGPT

Block scope is a concept in programming languages where variables are scoped to the nearest curly braces `{}` in which they are defined. This means that variables declared within a block (a set of statements enclosed within curly braces) are only accessible within that block and any nested blocks, but not outside of it.

In JavaScript, block scope was not available until the introduction of the `let` and `const` keywords in ECMAScript 6 (ES6). Before ES6, JavaScript only had function scope, meaning variables were scoped to the nearest enclosing function, rather than to the block in which they were defined.

Here's how block scope works with `let` and `const`:

1. `let`: Variables declared with `let` have block scope. This means they are only accessible within the block in which they are defined.

   javascript

   ```javascript
   {
       let x = 10;
       console.log(x); // Output: 10
   }
   console.log(x); // Throws ReferenceError: x is not defined
   ```

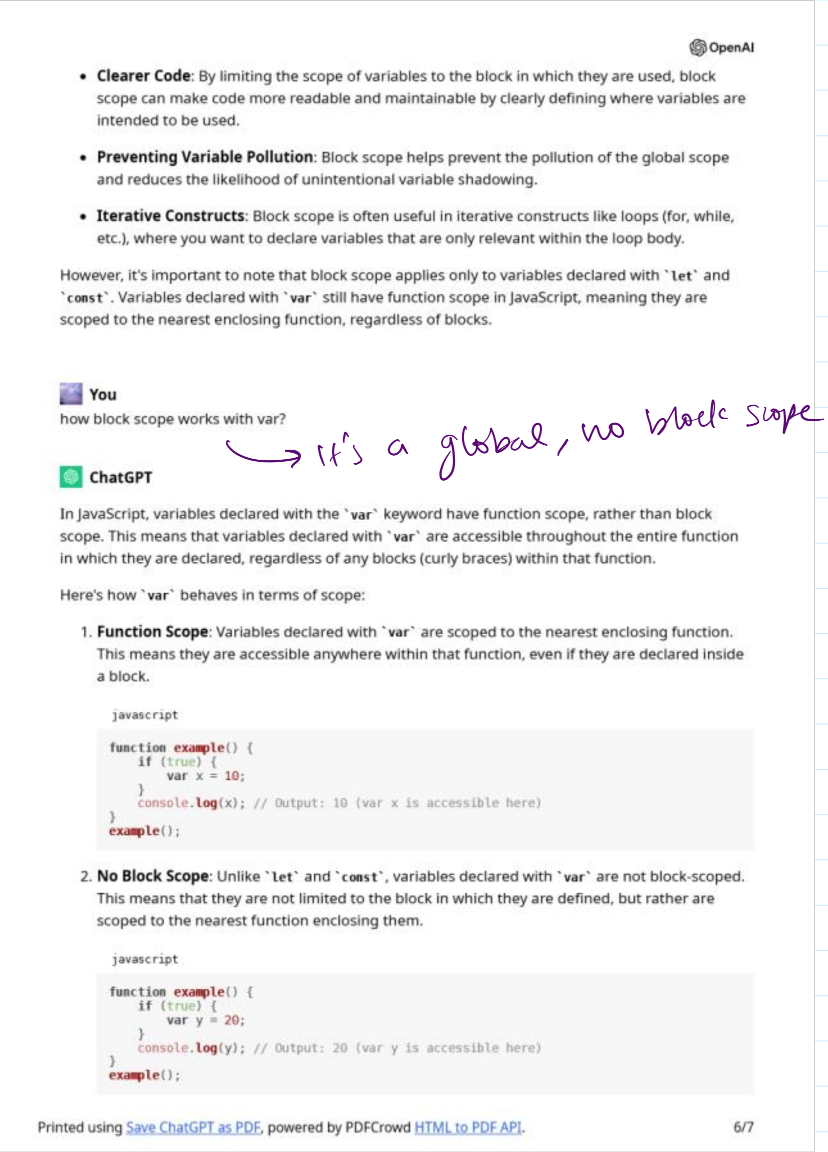
2. `const`: Similarly, variables declared with `const` also have block scope. Additionally, `const` variables cannot be reassigned after declaration.

   javascript

   ```javascript
   {
       const y = 20;
       console.log(y); // Output: 20
   }
   console.log(y); // Throws ReferenceError: y is not defined
   ```

Block scope has several benefits and use cases:

- **Encapsulation**: Block scope allows you to encapsulate variables within a specific block of code, reducing the risk of naming conflicts and unintended side effects.



- **Clearer Code**: By limiting the scope of variables to the block in which they are used, block scope can make code more readable and maintainable by clearly defining where variables are intended to be used.
- **Preventing Variable Pollution**: Block scope helps prevent the pollution of the global scope and reduces the likelihood of unintentional variable shadowing.
- **Iterative Constructs**: Block scope is often useful in iterative constructs like loops (for, while, etc.), where you want to declare variables that are only relevant within the loop body.

However, it's important to note that block scope applies only to variables declared with `` `let` `` and `` `const` ``. Variables declared with `` `var` `` still have function scope in JavaScript, meaning they are scoped to the nearest enclosing function, regardless of blocks.

You

how block scope works with var?

→ it's a global, no block scope here.

ChatGPT

In JavaScript, variables declared with the `` `var` `` keyword have function scope, rather than block scope. This means that variables declared with `` `var` `` are accessible throughout the entire function in which they are declared, regardless of any blocks (curly braces) within that function.

Here's how `` `var` `` behaves in terms of scope:

1. **Function Scope**: Variables declared with `` `var` `` are scoped to the nearest enclosing function. This means they are accessible anywhere within that function, even if they are declared inside a block.

   javascript

   ```javascript
   function example() {
       if (true) {
           var x = 10;
       }
       console.log(x); // Output: 10 (var x is accessible here)
   }
   example();
   ```

2. **No Block Scope**: Unlike `` `let` `` and `` `const` ``, variables declared with `` `var` `` are not block-scoped. This means that they are not limited to the block in which they are defined, but rather are scoped to the nearest function enclosing them.

   javascript

   ```javascript
   function example() {
       if (true) {
           var y = 20;
       }
       console.log(y); // Output: 20 (var y is accessible here)
   }
   example();
   ```

Because of its function scope behavior, using `var` can sometimes lead to unexpected behavior, particularly in cases where variables are accidentally accessed outside of the intended scope. This is one of the reasons why `let` and `const` were introduced in ECMAScript 6 (ES6) to provide block scope, which offers more predictable behavior and helps prevent common programming errors.

ChatGPT can make mistakes. Consider checking important information.