Global Scope Pollution

Global scope pollution in JavaScript refers to a situation where the global scope (also known as the "window" object in browsers) becomes cluttered with a large number of variables, functions, and other identifiers. When the global scope is polluted, it can lead to naming conflicts, unexpected behavior, and difficulties in debugging and maintaining the codebase.

Global scope pollution occurs when variables and functions are defined without proper scoping mechanisms, causing them to become part of the global scope. Here are some common causes of global scope pollution:

Implicit Global Variables

If a variable is assigned a value without being explicitly declared using var, let, or const, it becomes a global variable automatically. This often happens accidentally and should be avoided.

```
a = 10;
console.log(a); //10
```

Overwriting Global Properties

Assigning values to global properties like window, document, or built-in objects can lead to unintended behavior and conflicts with existing methods or properties.

Third-Party Libraries

Poorly designed third-party libraries that define global variables or functions can lead to conflicts with your own code or other libraries.

Global Event Handlers

Global event handlers in JavaScript refer to event handler functions that are attached to the global scope or the global objects like window or document. Defining event handler functions in the global scope can lead to naming conflicts if the same function name is used elsewhere.

Common Variable Names

Using common variable names in the global scope increases the likelihood of conflicts with other parts of the codebase like data, count, value, element, result, error, temp, tmp, callback, options, config etc.

Immediately Invoked Function Expression

IIFE stands for Immediately Invoked Function Expression. It's a common design pattern in JavaScript that involves defining and invoking a function all in one step. The primary purpose of an IIFE is to create a new scope for your code, avoiding variable collisions and polluting the global scope. IIFEs were particularly popular before the widespread adoption of block-scoped variables (let and const) and the introduction of modules in ES6.

Advantages of using IIFE

Avoiding Global Namespace Pollution

Variables declared inside an IIFE are local to the IIFE's scope and do not become global variables, reducing the risk of conflicts with other code.

Isolation

The IIFE creates a scope that isolates your code from the surrounding code. This can be especially important when working with libraries or third-party code to prevent unintended interactions.

Data Privacy

By keeping variables within the IIFE's scope, you can achieve a level of data privacy, making it harder for outside code to access or modify your variables.

Older JavaScript Environments

Before the introduction of block-scoped variables (let and const) and modules in ES6, IIFEs were a common way to create private scope and modularize code.

```
//Named IIFE
(function demo(){
  console.log('I am a named IIFE');
})();


// Anonymous IIFE
(function(){
  console.log('I am an anonymous IIFE');
})();

// Arrow IIFE
(()=> console.log('I am an arrow IIFE'))();

// Argument IIFE
((name)=> console.log(`I am an argument IIFE with name :
${name}`))("Anubhav Gupta");
```