***!! Shri Ganeshay Namah: !!***

Java Script Coding Standards

Javascript Coding Standards Are A Set Of Guidelines That Developers Follow To Write Clean, Readable, And Maintainable Code. Adhering To These Standards Improves Collaboration, Code Quality, And Overall Project Success. Below Is A Comprehensive Overview Of Javascript Coding Standards:

1. **File and Folder Structure**

* **File Names**: Use Lowercase Letters And Hyphens (-) To Separate Words. Avoid Using Spaces, Underscores, Or Special Characters.

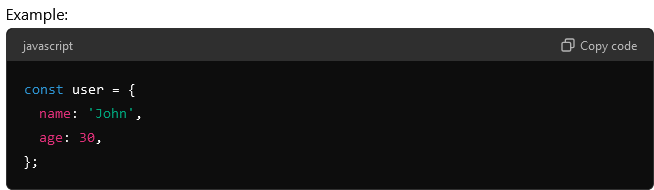
Example: user-profile.js, main-app.js

* **Folder Structure**: Organize Files In A Logical Structure, Grouping Similar Files Together. Common Practices Include Separating Concerns (E.G., Components, Utils, Services).



1. **Code Formatting**

* **Indentation:** Use 2 Or 4 Spaces Per Indentation Level. Consistency Across The Project Is Key.
* **Line Length:** Limit Lines To 80-100 Characters. If A Line Exceeds This Length, Consider Breaking It Up For Readability.
* **Semicolons:** Always End Statements With Semicolons. While Javascript Allows You To Omit Them, It Can Lead To Unexpected Behavior.
* **Quotes:** Use Single Quotes (') For Strings. Use Double Quotes (") Only For JSON Or When Escaping Is Necessary.
* **Trailing Commas:** Use Trailing Commas In Multiline Objects And Arrays. This Helps With Version Control Diffs.



1. **Variables And Constants**

* **Variable Naming:** Use camelCase For Variables. Use PascalCase For Functions, Classes And Constructors.

Example: let userName = 'John';, class UserProfile {}, function Sum();

* **Constants:** Use UPPERCASE\_SNAKE\_CASE For Constants That Are Not Meant To Change.

Example: const MAX\_USERS = 100;

* **Variable Declarations:** Prefer const For Variables That Won’t Change, And let For Variables That Might Change. Avoid Using var Due To Its Function-Scoped Behavior.

1. **Functions**

* **Function Declarations:** Use PascalCase For Function Names. Name Functions Based On Their Action Or Outcome (e.g., GetUserData, CalculateTotal).
* **Arrow Functions:** Prefer Arrow Functions (=>) For Anonymous Functions And When Not Using This.

Example: const add = (a, b) => a + b;

* **Function Length:** Keep Functions Short And Focused. If A Function Exceeds 20-30 Lines, Consider Breaking It Into Smaller Functions.
* **Default Parameters:** Use Default Parameters To Handle Optional Arguments.

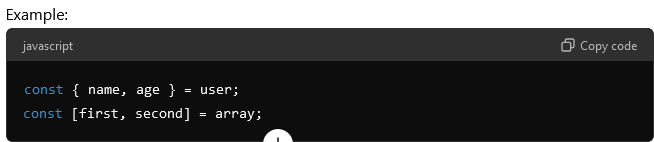


1. **Objects And Arrays**

* **Object Literals**: Use shorthand properties and methods where possible.



* **Destructing:** Use Destructuring To Extract Values From Objects And Arrays.



* **Spread Operator:** Use The Spread Operator (...) For Copying Objects And Arrays Or Merging Them.

Example: const newArray = [...array1, ...array2];

1. **Control Structures**

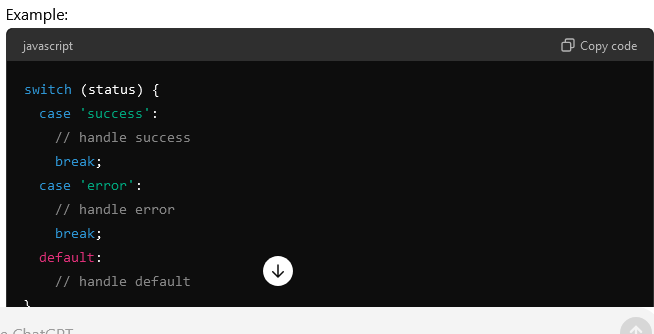
* **If-Else Statements**: Always Use Curly Braces ({}) Even For Single-Line Blocks. This Improves Readability And Avoids Mistakes.



* **Ternary Operators**: Use Ternary Operators For Simple Conditions, But Avoid Nesting Them As It Reduces Readability.

Example: const status = isLoggedIn ? 'Logged In' : 'Logged Out';

* **Switch Statements**: Use Switch Statements For Multiple Conditions. Always Include A Default Case.

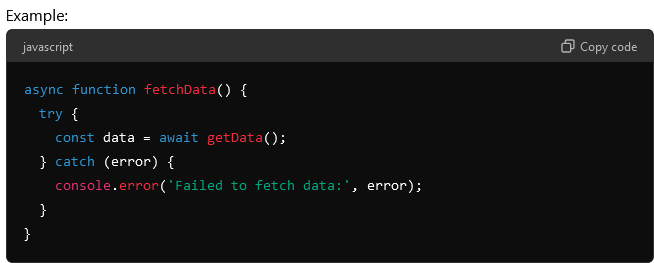


1. **Error Handling**

* **Try-Catch**: Use try-catch Blocks For Handling Exceptions. Ensure Errors Are Logged Or Handled Appropriately.



* **Promises:** Handle promise rejections using .catch() or async/await with try-catch.

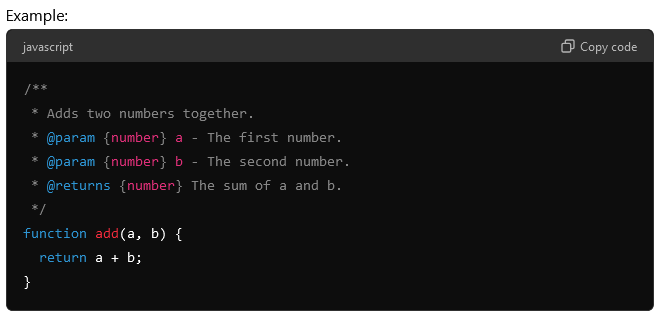


1. **Comments**

* **Inline Comments**: Use Inline Comments Sparingly And Only To Clarify Complex Code. Ensure They Are On The Same Line As The Code They Describe.

Example: const result = add(a, b); // add two numbers

* **Block Comments**: Use Block Comments For More Detailed Explanations, Especially At The Beginning Of A File Or Function.

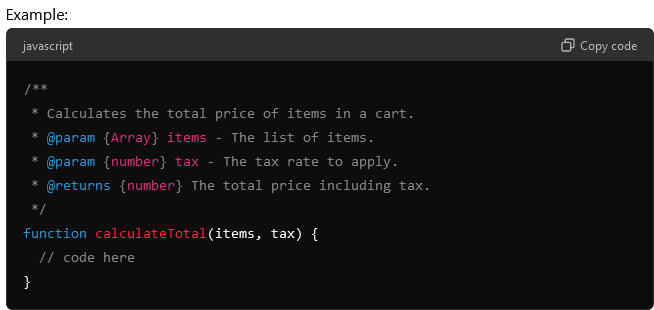


1. **ESLint and Prettier**

* **ESLint:** Use Eslint To Enforce Coding Standards And Catch Potential Errors. Customize The .eslintrc File According To Your Project Needs.
* **Prettier:** Use Prettier For Consistent Code Formatting. Integrate It With Eslint To Avoid Conflicts.

1. **Documentation**

* **JSDoc**: Use JSDoc Comments To Document Functions, Classes, And Modules. This Makes It Easier For Other Developers To Understand The Purpose And Usage Of Your Code.



1. **Performance Consideration**

* **Avoid Global Variables:** Minimize The Use Of Global Variables To Reduce The Risk Of Conflicts And Improve Performance.
* **Optimize Loops:** Be Mindful Of Loop Performance, Especially In Large Datasets. Consider Using Methods Like .Map(), .Filter(), Or .Reduce() For More Readable And Performant Code.
* **Lazy Loading:** Implement Lazy Loading For Resources That Are Not Immediately Required To Improve Initial Load Time.

1. **Security Best Practices**

* **Input Validation:** Always validate and sanitize user input to prevent security vulnerabilities like XSS or SQL injection.
* **Use HTTPS:** Ensure Your Application Is Served Over HTTPS To Protect Data In Transit.
* **Environment Variables:** Store Sensitive Information, Like API Keys Or Database Credentials, In Environment Variables. Do Not Hard-Code Them In Your Codebase.

1. **Code Review**

* **Peer Reviews:** Conduct Code Reviews Regularly To Catch Issues Early And Ensure That The Code Adheres To The Standards.
* **Review Tools:** Use Tools Like Github Pull Requests Or Gerrit To Facilitate Code Reviews.

**Conclusion**

Following These Javascript Coding Standards Ensures That Your Code Is Consistent, Maintainable, And Easy To Understand. While Some Standards May Vary Depending On The Project Or Team, Adhering To A Consistent Set Of Guidelines Will Improve Collaboration And Reduce Errors In The Long Run.

The performance difference between the two selectors is generally **negligible** in most cases, but if you’re looking for a detailed explanation:

**1. $('#assessment-year', myModal).val(columnValues[1]);**

* **Scope Restriction**: This selector looks for #assessment-year only within the context of the myModal element. This is **more efficient** if myModal is a smaller subset of the DOM because it limits the search to a smaller area.
* **Performance**: Since you're restricting the scope to a subset of the DOM (myModal), jQuery has fewer elements to search through. This can make the query slightly faster compared to searching the entire DOM, especially if myModal contains only a few elements.

**2. $('#assessment-year').val(columnValues[1]);**

* **Global Search**: This selector looks for #assessment-year **in the entire DOM**. If there are many elements on the page, jQuery will have to search through all of them to find the element with that ID, which could be slightly slower if the DOM is large.
* **Performance**: This might be **slightly slower** if the page has a lot of elements, but for most typical web pages, this difference is usually not noticeable unless you're dealing with very complex, large DOMs (e.g., hundreds of thousands of elements).

**Which is faster?**

* **In practice**, the difference in performance will likely be so small that it won’t matter for most web applications.
* If you're dealing with a small modal with relatively few elements, **the performance difference will be almost imperceptible**.
* **If the page has a very large DOM** and you need to optimize, limiting the scope (like in the first example) can be **slightly faster**, but again, for typical scenarios, this is rarely a performance bottleneck.

**Conclusion:**

For most cases, you won’t notice any significant performance difference. However, if the modal (myModal) is large and you're concerned about performance in a highly dynamic, complex application, narrowing the search scope (i.e., using $('#assessment-year', myModal)) is a good practice.