

DWA_01.3 Knowledge Check_DWA1

1. Why is it important to manage complexity in Software?

It can lead to catastrophic failures if not managed properly. Therefore it is crucial to be aware of issues before code launches or have a way to fix and recover from them.

2. What are the factors that create complexity in Software?

Lack of abstraction: When code lacks proper abstraction, it becomes harder to understand and maintain. The more we need to know about the internal workings of a code module, the more complex our software becomes.

Poor design and architecture: If the overall design and architecture of a software system are not well thought out, it can lead to unnecessary complexity.

Lack of documentation: Insufficient or unclear documentation can make it difficult for developers to understand how to use a code module correctly, leading to complexity.

3. What are ways in which complexity can be managed in JavaScript?

Modularity: Break your code into smaller, manageable modules or functions. Each module should have a specific purpose and be responsible for a specific task.

Abstraction: Hide unnecessary details and expose only the essential information. This makes your code more readable and reduces complexity.

Encapsulation: Enclose related data and functions within objects or classes. This helps in organizing code and prevents unwanted access to internal details.

Documentation: Document your code to provide clear explanations of its purpose, functionality, and usage. This helps other developers understand your code and reduces the complexity of working with it.

Code Reusability: Write reusable code by creating functions or classes that can be used in multiple parts of your program. This reduces duplication and simplifies the overall structure.

Testing: Regularly test your code to ensure that it behaves as expected. This helps in identifying and fixing issues early, reducing the complexity of debugging later on.

4. Are there implications of not managing complexity on a small scale?

Yes;

Reduced Readability: Unmanaged code tends to lack clear structure and organization. Variable names may be arbitrary, and the overall flow of the code might be confusing. Therefore developers, including the original coder, may struggle to understand the code. This can lead to difficulties in debugging, maintaining, or extending the software.

Increased Bug Count: Unmanaged code is more likely to have hidden bugs. Poorly structured logic and unclear variable scopes can introduce errors that go unnoticed. Therefore uncaught bugs can lead to unexpected behavior or even software failures. Debugging becomes challenging without a clear understanding of the code.

Difficulty in Collaboration: Lack of organization makes it hard for multiple developers to collaborate. Inconsistent naming conventions and coding styles may confuse team members. Therefore collaboration becomes inefficient, and the chances of introducing errors during collaborative efforts increase. Code reviews may take longer, and misunderstandings among team members may occur.

Maintainability Challenges: Unmanaged code lacks modularity and may have redundant or overly complex sections. Changes or updates become risky without a proper understanding of the existing code. Therefore small-scale projects can become difficult to maintain. Simple updates or feature additions may lead to unintended consequences or require extensive modifications.

Risk of Technical Debt: Technical debt, which results from choosing quick solutions over well-structured ones, accumulates without proper code management. Therefore technical debt increases the cost of future development. Quick fixes may have to be revisited, leading to additional work. This can hinder long-term project sustainability.

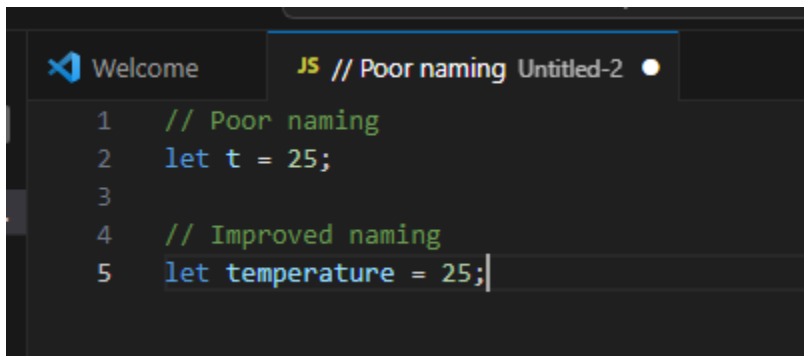
Negative Impact on Developer Experience: Developers working on poorly managed code may find the coding experience frustrating and less enjoyable. Therefore dissatisfaction among developers may impact their productivity and motivation. It can also contribute to higher turnover rates.

5. List a couple of codified style guide rules, and explain them in detail.

1.Descriptive Variable Naming:

- Rule: Use meaningful and descriptive names for variables.
- Explanation: Descriptive variable names enhance code readability. Instead of using generic names or abbreviations, opt for names that convey the purpose or content of the variable.
- Importance: Descriptive variable names make the code self-explanatory, reducing the need for additional comments. It helps developers understand the intent and role of variables, leading to better maintainability.

Example:

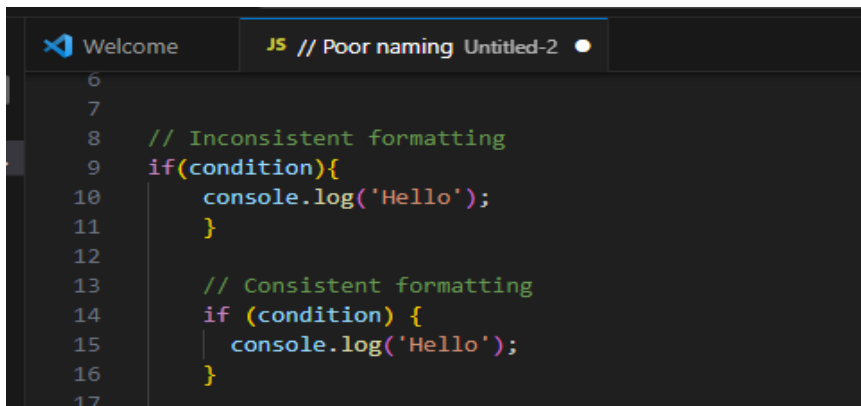
A screenshot of the Visual Studio Code editor interface. The top bar shows a 'Welcome' tab and a file named 'JS // Poor naming Untitled-2'. The editor area contains five lines of JavaScript code. Line 1 is a comment: '// Poor naming'. Line 2 is a variable declaration: 'let t = 25;'. Line 3 is an empty line. Line 4 is a comment: '// Improved naming'. Line 5 is a variable declaration: 'let temperature = 25;'. The code is syntax-highlighted with green for comments, blue for the 'let' keyword, and black for the variable names and values.

```
1 // Poor naming
2 let t = 25;
3
4 // Improved naming
5 let temperature = 25;
```

2.Consistent Code Formatting:

- Rule: Maintain consistent code formatting, including indentation and spacing.
- Explanation: Consistent formatting improves code aesthetics and readability. Use a consistent number of spaces for indentation, align brackets properly, and add whitespace between operators and operands.
- Importance: Consistent formatting makes the code visually coherent. It eases collaboration among team members who follow the same conventions. Tools like linters can help enforce consistent formatting.

Example:

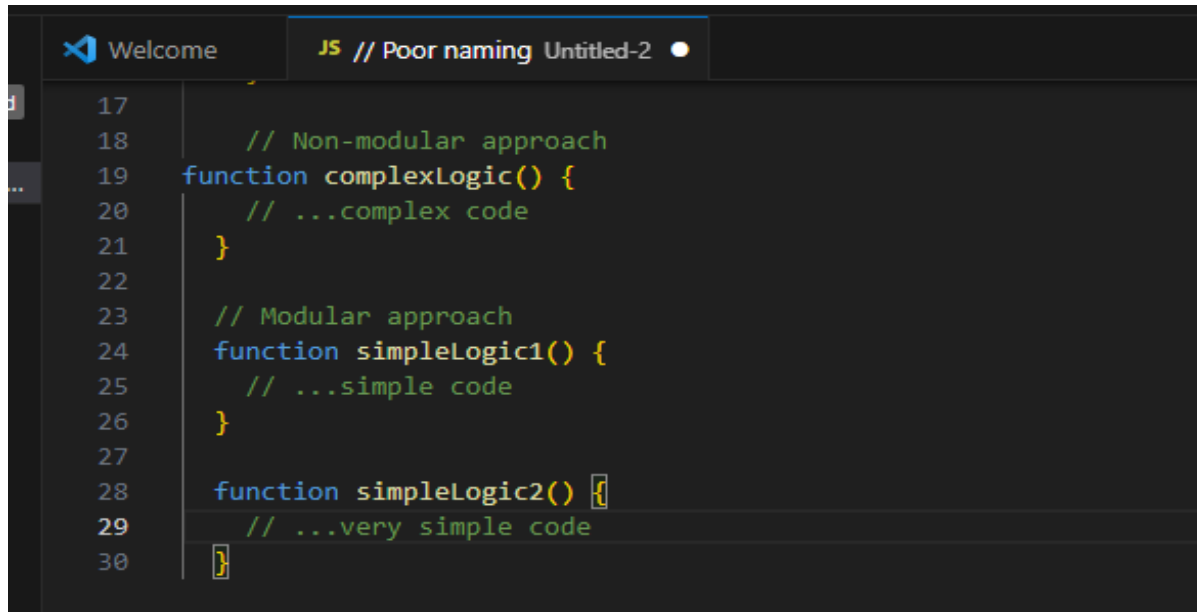
A screenshot of the Visual Studio Code editor interface. The top bar shows a 'Welcome' tab and a file named 'JS // Poor naming Untitled-2'. The editor area contains code from line 6 to line 17. Lines 8-11 show 'Inconsistent formatting' with an if-statement where the opening and closing curly braces are not aligned. Lines 13-16 show 'Consistent formatting' with an if-statement where the opening and closing curly braces are aligned. The code is syntax-highlighted with green for comments, blue for the 'if' keyword, and black for the variable names and values.

```
6
7
8 // Inconsistent formatting
9 if(condition){
10     console.log('Hello');
11 }
12
13 // Consistent formatting
14 if (condition) {
15     console.log('Hello');
16 }
17
```

3.Modular Code Organization:

- Rule: Break down complex code into smaller functions or methods.
- Explanation: Modularization improves code maintainability by isolating specific functionalities. Each function or method should have a single responsibility.
- Importance: Modularization simplifies understanding, testing, and updating of code. Each module can be developed and tested independently, contributing to a more organized and scalable codebase.

Example:



```
17
18 // Non-modular approach
19 function complexLogic() {
20     // ...complex code
21 }
22
23 // Modular approach
24 function simpleLogic1() {
25     // ...simple code
26 }
27
28 function simpleLogic2() {
29     // ...very simple code
30 }
```

6. To date, what bug has taken you the longest to fix - why did it take so long?

I have had many bugs that have taken me too long to fix, therefore leaving me far behind the rest of my cohort or still haven't fixed but of the top of my head the one I cannot forget is;

IWA9: I had a "whitespace bug" - space on line 40 in the below screenshot. My interpolation did not correspond to the object, once I removed the space I was able to complete the challenge and get the correct calculated value.

The reason it took me so long was because I could not identify the problem, I was also still developing my debugging skills and VScode did not show or highlight the bug. Only after console logging each line of code, I managed to locate my error.

