DWA_01.3 Knowledge Check_DWA1

---

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

When software fails, there are consequences ranging from inconveniences, such as loss of productivity, to catastrophic incidents like the Ariane 5 and Mars Climate Orbiter incidents. Increasing complexity results in a larger number of points of potential failure in an application.

Code needs to be easily read and understood by many different people that may be working on the project - including the original author, since they may forget the details of how their own code works. Increasing complexity makes code harder to understand which makes it harder to debug and fix problems that arise - mistakes are both harder to spot and harder to correct. In addition, managing complexity prevents misunderstandings and faulty assumptions that can result in software failures.

Increasing complexity will also increase the time required to do any work on the codebase such as refactoring or adding new features, since someone will need to first understand the code before making changes.

---

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

- Software has inherent complexity as a result of needing to solve the problem at hand
- Evolving requirements that change as the project develops can contribute complexity as changes are introduced
- Rushing a project can increase complexity - the code works but has not been streamlined
- Increasing the scale of a project will increase complexity - there are more components and more ways in which these components will interact.

---

3. What are ways in which complexity can be managed in JavaScript?
- Following code style guides

- Documenting and commenting code, especially using JSDoc
    - Describing what the code does
    - Setting out expected data types, values, returns and shapes of data
- Building code in a modular way - less code to comprehend at a time
- Utilizing abstraction to keep unnecessary details hidden from the person working with the code
- Prioritize readability rather than ‘cleverness’ or performance

---

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

Yes - even at small scales, increasing complexity unnecessarily can result in easily avoided errors. Furthermore, not managing complexity on a small scale will result in an accumulation of complexity at larger scales, should the project be expanded upon later.

---

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

Coding style guides comprise a set of rules for how to write and structure code so that it is clean, readable and maintainable. This includes formatting conventions, naming conventions, file/folder structures and general good practices.

Standard JavaScript Style Guide

1. Indentation: Code should use 2 spaces for indentation. Tabs should not be used.
2. Semicolons: Semicolons should be used at the end of each statement.
3. Line Length: Lines should not exceed 80 characters. If a line becomes too long, it's recommended to break it into multiple lines.
4. Variable Declarations: Prefer using `const` and `let` for variable declarations instead of `var`. Use `const` for variables that won't be reassigned, and `let` for variables that will be reassigned.
5. Naming Conventions: Use camelCase for variable and function names. Names should be descriptive and meaningful.
6. Spacing: Use spaces around operators and after commas. There should be no space between a function name and its opening parenthesis. Use a space before the opening brace of a block.

7. Quotation Marks: Use single quotes (') for strings unless the string contains a single quote. In such cases, use double quotes ("") for the string and escape any contained double quotes.
8. Object and Array Formatting: Use the following formatting for object and array literals:

```
const person = {
  name: 'John',
  age: 25,
  hobbies: ['reading', 'coding'],
};

const numbers = [1, 2, 3, 4, 5];
```

9. Function Declarations: Use named function expressions or arrow functions instead of function declarations.`(a, b) => a * b;`
10. Conditional Statements: Use braces for all multi line blocks in `if`, `else`, `for`, `while`, etc. statements, even if the block only contains a single statement.
11. Error Handling: Always handle errors explicitly, either through `try-catch` blocks or using promises.
12. Comments: Use comments to explain complex logic, important assumptions, and any workarounds or hacks used. Comments should be concise and provide meaningful explanations.
13. Modules: Use ES6 modules (`import` and `export`) for handling dependencies and organizing code.
14. File Naming: Use lowercase filenames with hyphens to separate words (e.g., `my-module.js`).

AirBnb JavaScript Style Guide

1. File naming: JavaScript files should use lowercase letters with hyphens as word separators. For example, `my-script.js` or `utils.js`.
2. Indentation: Use 2 spaces for indentation. Avoid using tabs.
3. Semicolons: Use semicolons to terminate statements.

4. Variables: Use `const` for variables that do not reassign, and `let` for variables that are reassigned. Avoid using `var`.
5. Object and Array declaration: Use object and array literal syntax (`{}` and `[]`) instead of their constructors (`new Object()` and `new Array()`).
6. String quotes: Use single quotes (`''`) for strings, unless the string itself contains single quotes, in which case you can use double quotes (`""`).
7. Function declaration: Use function expressions or arrow functions instead of function declarations. This helps in preventing hoisting-related issues.
8. Naming conventions: Use camelCase for variable, function, and method names. Use PascalCase for class and constructor names.
9. Spacing: Use spaces around operators, after commas, and after colons. For example, `const result = 5 + 3;`.
10. Braces: Use the "one true brace style" (OTBS) with opening braces on the same line as the corresponding statement and closing braces on a new line.
11. Comments: Use descriptive comments to explain complex code or indicate the intention behind it. Avoid unnecessary or redundant comments.
12. Modules: Use a module bundler like webpack or rollup to organize and bundle JavaScript modules. Avoid using the module pattern.

Google JavaScript Style Guide

1. File Naming: JavaScript files should use lowercase letters with words separated by underscores. For example, `my_script.js`.
2. File Encoding: JavaScript files should be encoded in UTF-8 without a byte order mark (BOM).
3. Indentation: Two spaces should be used for indentation. Avoid using tabs.
4. Line Length: Lines should not exceed 80 characters. If necessary, it's recommended to break long lines into multiple lines.
5. Whitespace: Use spaces around operators, after commas, colons, and semicolons. Use whitespace to improve code readability, but avoid excessive or unnecessary whitespace.
6. Curly Braces: Curly braces should be used even when they are optional. Opening braces should be placed on the same line as the corresponding statement, and closing braces should be on a new line.
7. Semicolons: Semicolons should be used at the end of statements.
8. Variables: Declare variables with `const` or `let`. Avoid using `var`. Declare each variable on a separate line and initialize it.

9. Constants: Constants should be named using uppercase letters with underscores. For example, `CONSTANT_NAME`.
10. Object Literal: When defining an object literal, use the shorthand notation for properties when possible.
11. Function Declarations: Use function declarations instead of function expressions whenever possible.
12. Function Parameters: Avoid using more than three parameters in a function. If a function has more than three parameters, consider using an options object instead.
13. Function Calls: When calling a function with multiple arguments, each argument should be on a new line.
14. Constructor Functions: Constructor functions should start with a capital letter and should be called with the `new` keyword.
15. Classes: Use the class syntax introduced in ECMAScript 2015 for creating objects and constructors.
16. Modules: Use JavaScript modules (ES modules) for managing dependencies and exporting values.
17. Strings: Use single quotes for string literals. Use template literals (backticks) for concatenating strings with variables or expressions.
18. Comments: Use comments to explain complex code, document functions, and provide an overview of the code. Prefer descriptive variable and function names over comments.
19. Error Handling: Use appropriate error handling techniques, such as `try-catch` blocks, to handle and report errors.
20. Naming Conventions: Use descriptive and meaningful names for variables, functions, classes, and constants. Use camel case for functions, variables, and object properties. Use Pascal case for class names.

---

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

In the IWA17 Calendar app project I had a bug in the nested for loops where I had placed a variable to store the resulting array in the wrong place, so it was feeding the results of the inner loop back into the outer loop and building it out exponentially with each iteration, instead of restarting a new array with each iteration in the outer loop.

It took a long time to solve because I didn't really understand the behavior of the nested for loop, which was a more complex structure than what I had previously worked with,

so I wasn’t able to anticipate how the code would run and couldn’t see where the problem was arising.
