DWA_01.3 Knowledge Check_DWA1

---

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

- Managing complexity is vital for software **maintainability:** code with reduced complexity makes it easier for other developers to comprehend the code and its intended functionality. This further simplifies the task of maintaining the code as it can be modified and enhanced without introducing issues.
- It makes **debugging** easier: less complex code is easier to read through and identify issues, and work to fix them. Muddled-up code makes it difficult to isolate problem-areas and can be time-consuming and costly.
- Excellent for software **perfomance:** convoluted code can breed inefficient code operation and slow down a programme's execution. Quality, clean code can also reduce redundant and unnecessary operations, further improving performance.

---

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

- Larger **project scale** will result in large codebases, deep levels of nesting as well as extensive functions, which will often increase the code complexity with each line.
- **Lack of commenting** can result in the code being hard to navigate, and thus hard to enhance or update as there is no clear rationale and logic demonstrated, and hence summing it up to being complex.
- **Inconsistent coding styles** can make code incredibly complex, leading to difficult long-term maintainability, as well as make team collaboration hard.

---

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

- **Use of clear and descriptive naming:**
- descriptive names for functions, classes, objects, variables and such improve code readability and general understanding as they clearly define the purpose and functionality of each case
- generic naming such "arr1, arr2", "var1, var2" and similar choices should be avoided as they can create confusion and breed bugs as they do not clearly represent their purpose or intended use.

- Good **code structure:**
- Grouping related functions and solutions together in a logical manner
- using indentation to visually organise the code and its intended execution also makes it easier to spot issues

- **Error handling:**
- Catch specific errors with 'try….catch' blocks, and add error messages to aid for easier debugging

- **Early testing:**
- Early testing and breakpoint checking helps verify code-correctness and prevents larger eventual bugs or issues

- Use of **const** and **let:**
- Use 'const' for variables that should not be reassigned, providing immutability and reducing the chance of accidental modifications.
- use 'let' for variables that need to be reassigned or have a limited scope within a block

---

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

Absolutely.

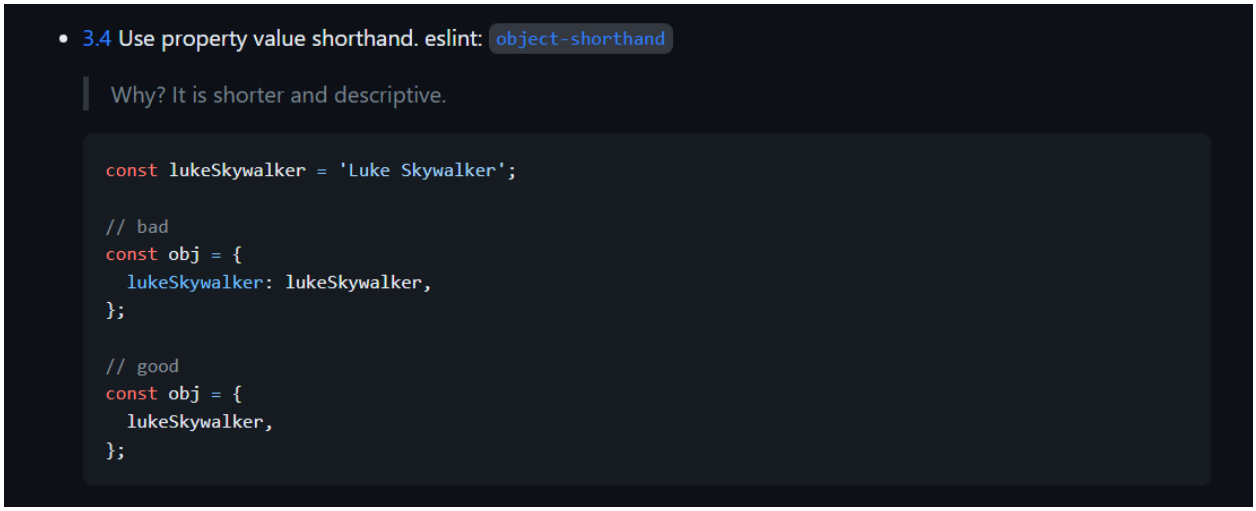
Disregarding or not managing complexity on a small scale can have implications such as but not limited to:

- **Testability**: if code units are lacking coherence or clear order, they make it difficult to single-out and address individual issues
- **Collaboration**: complex code that is incorrectly managed can deter understanding amongst team members, making it difficult for them to work with efficiently or productively
- **Maintainabilty**: poorly managed code could possibly lead to poor and difficult maintainability as there is no clear area to debug or modify.
- **Cost**: the inability to efficiently maintain complex code from a small scale can result in project delays further down the timeline as it accumulates and will require more time, effort and cost to address issues.

---

5. List a couple of codified style guide rules, and explain them in detail.
As per the AirBnB JavaScript style guide:
https://github.com/airbnb/javascript

1. Use of shorthand



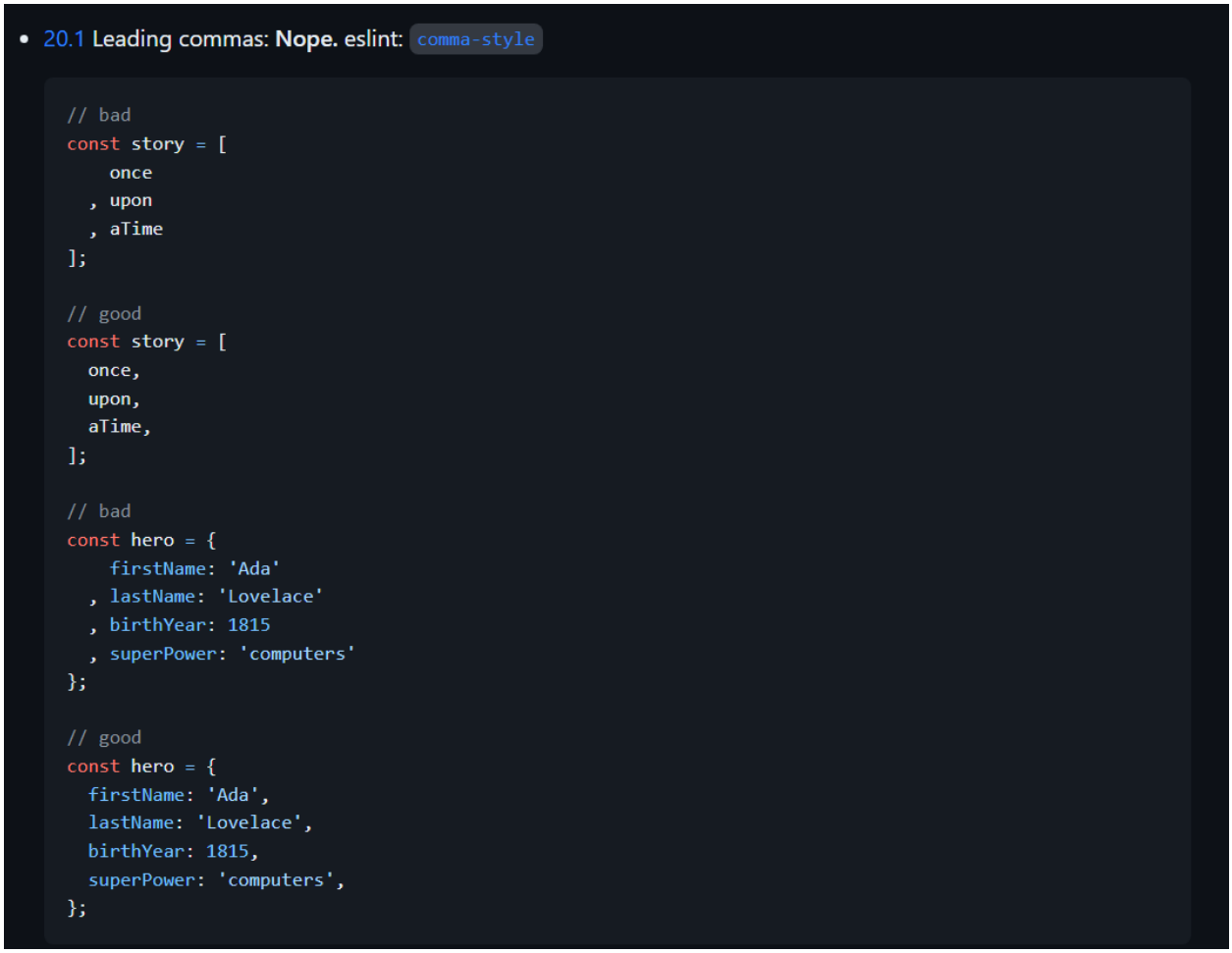
In the first '**bad**' code, the property name is explicitly mentioned as 'lukeSkywalker', and the value is assigned from the variable lukeSkywalker. However, this approach is considered bad in terms of code style for reasons including:

- **Redundancy**: The property name is unnecessarily repeated twice, making the code less concise. It creates redundancy, especially when the variable name and property name are the same, as in this case.
- **Readability**: The redundant repetition can reduce code readability and increase the chance of errors. It may lead to confusion or mistakes if the property name and variable name are not kept in sync.
- **Maintainability**: If you decide to change the variable name, you would need to remember to update the property name as well, which adds an extra step and increases the possibility of introducing bugs.

The second '**good**' code uses the property value shorthand, and the value from the lukeSkywalker variable is assigned directly to a property with the same name. This approach is considered good for these reasons:

- **Conciseness**: The shorthand notation eliminates the redundancy of repeating the property name, resulting in cleaner and more concise code.
- **Clarity**: The code becomes more readable because the property name and its corresponding value are placed together within the object literal, making it easier to understand the structure at a glance.
- **Maintenance**: If you decide to change the variable name, you only need to modify it in one place. The property name will automatically match the updated variable name, ensuring consistency and reducing the chance of introducing bugs.

2. Leading vs Trailing commas



- 20.1 Leading commas: **Nope.** eslint: `comma-style`

```
// bad
const story = [
    once
  , upon
  , aTime
];

// good
const story = [
  once,
  upon,
  aTime,
];

// bad
const hero = {
    firstName: 'Ada'
  , lastName: 'Lovelace'
  , birthYear: 1815
  , superPower: 'computers'
};

// good
const hero = {
  firstName: 'Ada',
  lastName: 'Lovelace',
  birthYear: 1815,
  superPower: 'computers',
};
```

In the '**bad**' code, the commas in the array are placed at the beginning of each line, following the "leading commas" style. This style is discouraged because of these reasons:

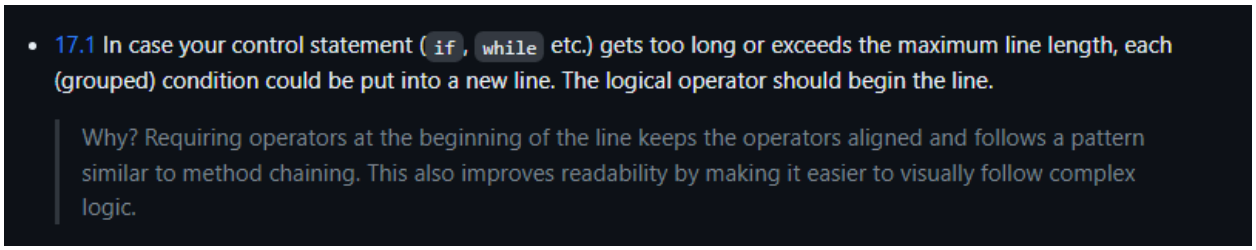
- **Inconsistency**: The leading commas in this style break the common convention of placing commas at the end of lines. Most JavaScript codebases and style guides follow the trailing comma style, making this code inconsistent with the expected format.
- **Visual Noise**: The leading commas create visual noise, as they stand out at the beginning of each line. This can make the code harder to read and understand, as it breaks the natural flow of reading left-to-right.

In the '**good**' code, the commas are placed at the end of each line, following the trailing comma style. This style is considered good because of these reasons:

- **Consistency**: The trailing comma style is widely adopted in JavaScript codebases and aligns with common coding conventions. It ensures a consistent and predictable comma placement throughout the code, making it easier for developers to read and maintain.

- **Readability**: Placing the commas at the end of each line allows the code to flow naturally from left to right. It reduces visual noise and improves readability by maintaining a consistent indentation pattern.

3. 'If' statements



- 17.1 In case your control statement (`if`, `while` etc.) gets too long or exceeds the maximum line length, each (grouped) condition could be put into a new line. The logical operator should begin the line.

  > Why? Requiring operators at the beginning of the line keeps the operators aligned and follows a pattern similar to method chaining. This also improves readability by making it easier to visually follow complex logic.

```
// bad
if ((foo === 123 || bar === 'abc') && doesItLookGoodWhenItBecomesThatLong() && isThisReallyHappening()) {
  thing1();
}

// bad
if (foo === 123 &&
  bar === 'abc') {
  thing1();
}

// bad
if (foo === 123
  && bar === 'abc') {
  thing1();
}

// bad
if (
  foo === 123 &&
  bar === 'abc'
) {
  thing1();
}

// good
if (
  foo === 123
  && bar === 'abc'
) {
  thing1();
}

// good
if (
  (foo === 123 || bar === 'abc')
  && doesItLookGoodWhenItBecomesThatLong()
  && isThisReallyHappening()
) {
  thing1();
}

// good
if (foo === 123 && bar === 'abc') {
  thing1();
}
```

In these bad examples, the conditions are split across multiple lines, but the logical operator (&& in this case) is not placed at the beginning of the line. These styles are considered bad for the following reasons:

- **Inconsistency**: The placement of the logical operator does not follow a consistent pattern, making the code harder to read and understand. It breaks the convention of aligning operators and deviates from widely accepted coding styles.
- **Readability**: When the logical operator appears at the end of a line, it can be visually disconnected from the conditions, especially when the conditions themselves span multiple lines. This can decrease readability, especially for complex logic with many conditions.
- **Maintenance**: If conditions need to be added, modified, or removed, adjusting the formatting by moving the logical operators can be error-prone and time-consuming.

---

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

For IWA17, the calendar that automatically calculates the current month and day and displays all dates in a grid format:

I spent days struggling to have my weekends, alternate weeks and current day displayed to my webpage despite being sure that the logic was correct.

Initial code which resulted in the bug:

```
// Setting class for weekends
    if (dayOfWeek === 1 || dayOfWeek === 7) {
      classString += 'table__cell_weekend';
    };
```

Resolved code which removed the bug and displayed the weekends (as well as alternate weeks and current day:

```
// Setting class for weekends
    if (dayOfWeek === 1 || dayOfWeek === 7) {
      classString += ' table__cell_weekend';
    };
```

The crucial difference between the two examples is the presence of a space character in Code A. Specifically, in Example A, there is a space character between the existing value of classString and the string ' table__cell_alternate' being appended to it.

In Code B, the string being appended, 'table__cell_alternate', does not have a leading space. As a result, when Code B is executed, the appended class name will be directly concatenated to the existing value of classString without any space separation.

The space character in Code A is crucial for maintaining proper class separation when multiple class names are appended to the classString variable. Class names in HTML and CSS are typically separated by space characters. Without the space character, the appended class name would be concatenated without any separation, resulting in an incorrect class name format.

```
const createHtml = (data) => {
  let result = '';

  data.forEach ((week) => {
      let inner = ''
      inner += addCell('table__cell table__cell_sidebar', `Week ${week.week}`)

      week.days.forEach((day) => {
          const { dayOfWeek, value } = day;
          let classString = 'table__cell';

          // Setting class for weekends
          if (dayOfWeek === 1 || dayOfWeek === 7) {
            classString += ' table__cell_weekend';
          };

          // Setting class for alternate weeks
          if (week.week % 2 === 0) {
            classString += ' table__cell_alternate';
          }

          // Setting class for current date
          if (value === new Date().getDate()) {
            classString += ' table__cell_today';
          }

          inner += addCell( classString, value);
      });

      result += `<tr>${inner}</tr>`;
      });
      return result;
};
```
