DWA_07.4 Knowledge Check_DWA7

---

1. Which were the three best abstractions, and why?

    1. DOM Elements object

```
const domElements = {
    list : {
    dataListItems : document.querySelector('[data-list-items]'),
    dataListButton : document.querySelector('[data-list-button]'),
    dataListClose : document.querySelector('[data-list-close]'),
    dataListActive : document.querySelector('[data-list-active]'),
    dataListMessage : document.querySelector('[data-list-message]'),
    dataListBlur : document.querySelector('[data-list-blur]'),
    dataListImage : document.querySelector('[data-list-image]'),
    dataListTitle : document.querySelector('[data-list-title]'),
    dataListSubtitle : document.querySelector('[data-list-subtitle]'),
    dataListDescription : document.querySelector('[data-list-description]'),},
    search : {
    dataSearchGenres : document.querySelector('[data-search-genres]'),
    dataSearchAuthors : document.querySelector('[data-search-authors]'),
    dataSettingsTheme : document.querySelector('[data-settings-theme]'),
    dataSearchCancel : document.querySelector('[data-search-cancel]'),
    dataSearchOverlay : document.querySelector('[data-search-overlay]'),
    dataSearchTitle : document.querySelector('[data-search-title]'),
    dataSearchForm : document.querySelector('[data-search-form]'),},
    settings : {
    dataSettingsCancel : document.querySelector('[data-settings-cancel]'),
    dataSettingsOverlay : document.querySelector('[data-settings-overlay]'),
    dataSettingsForm : document.querySelector('[data-settings-form]'),
    dataHeaderSearch : document.querySelector('[data-header-search]'),
    dataHeaderSettings : document.querySelector('[data-header-settings]')}
}
```

For this abstraction, all DOM elements into an object. This allows for all of the same instances to be grouped together and for more controlled access in other areas of the code. It was then also further abstracted into nested objects of DOM elements with similar data-keys

2.

```
// @ts-ignore
domElements.list.dataListButton.addEventListener('click', () => {
    const fragment = document.createDocumentFragment()

    for (const { author, id, image, title } of matches.slice(page * BOOKS_PER_PAGE, (page + 1) * BOOKS_PER_PAGE)) {
        const element = document.createElement('button');
                // @ts-ignore
                element.classList = 'preview';
        element.setAttribute('data-preview', id);

        element.innerHTML = `
            <img
                class="preview__image"
                src="${image}"
            />

            <div class="preview__info">
                <h3 class="preview__title">${title}</h3>
                <div class="preview__author">${authors[author]}</div>
            </div>
        `;

        fragment.appendChild(element);
    };

    // @ts-ignore
    domElements.list.dataListItems.appendChild(fragment);
    page += 1;
});
```

This section of code was on one of the better areas of abstraction as the function created was meant to serve one purpose only ie:
To create the HTML preview code snippet of a book , which would include the name of the authors and books with the cover images

3.

```
/**
 * Function to close overlay boxes
 * @returns {boolean}
 */
const openFalse = (item) => {
    return item.open = false;
};

/**
 * Function to open overlay boxes
 * @returns {boolean}
 */
const openTrue = (item) => {
    return item.open = true;
};
```

These 2 functions were created with the functionality of opening and closing overlay boxes when the 'theme, search, preview' buttons were clicked to open the overlay and then also when the cancel/close buttons were clicked on said overlays to close them. This abstraction callback was then placed in more than one area of the code.

---

2. Which were the three worst abstractions, and why?

1.
```
for (const { author, id, image, title } of matches.slice(page * BOOKS_PER_PAGE, (page + 1) * BOOKS_PER_PAGE)) {
    const element = document.createElement('button');
            // @ts-ignore
            element.classList = 'preview';
```

   The for-of loops had multiple properties extracted from an iterable. The functionality became overcrowded because there was another method placed on the iterable.

2. Multiple fragments

```
const genreHtml = document.createDocumentFragment()
const firstGenreElement = document.createElement('option')
```

```
const authorsHtml = document.createDocumentFragment()
const firstAuthorElement = document.createElement('option')
```

```
domElements.list.dataListItems.innerHTML = '';
const newItems = document.createDocumentFragment();
```

In the code for genres, authors and previews there are multiple fragments elements created. All have unique identifiers but serve the same purpose

3. HTML duplication

```
// @ts-ignore
domElements.list.dataListButton.innerText = `Show more (${books.length - BOOKS_PER_PAGE})`;
// @ts-ignore
domElements.list.dataListButton.disabled = (matches.length - (page * BOOKS_PER_PAGE)) > 0;

// @ts-ignore
domElements.list.dataListButton.innerHTML = `
    <span>Show more</span>
    <span class="list__remaining"> (${(matches.length - (page * BOOKS_PER_PAGE)) > 0 ? (matches.length - (page * BOOKS_PER_PAGE)) : 0})</span>
`
```

Here, the same element has HTML created for it in more than one instance. The same logic is being used to check the amount of books remaining, and generates the HTML to be displayed. This creates code redundancy

---

3. How can The three worst abstractions be improved via SOLID principles.

1. Extraction from objects with additional functions

```
for (const { author, id, image, title } of matches.slice(page * BOOKS_PER_PAGE, (page + 1) * BOOKS_PER_PAGE)) {
    const element = document.createElement('button');
            // @ts-ignore
            element.classList = 'preview';
```

This could be further abstracted by taking the iterable (matches), and creating a new separate variable that selects the required amount of books and stores it. This makes use of the Open-Close principle , in the sense that the contents of the variable is not modified , but extended so that just its name replaces the complexity

2. Multiple fragments

```
const genreHtml = document.createDocumentFragment()
const firstGenreElement = document.createElement('option')
```

```
const authorsHtml = document.createDocumentFragment()
const firstAuthorElement = document.createElement('option')
```

```
domElements.list.dataListItems.innerHTML = '';
const newItems = document.createDocumentFragment();
```

One function can be created in order to generate a fragment with default content, and callbacks can be added where needed with different inputs to give the required outputs. This is in reference in essence to the Liskov Substitution Principle (LSP), which states that the properties of the parent can be extended to the child (in this case where the main function is called in different areas), and even replaced, provided that the main functionality of the parent is not affected.

3. HTML duplication

```
// @ts-ignore
domElements.list.dataListButton.innerText = `Show more (${books.length - BOOKS_PER_PAGE})`;
// @ts-ignore
domElements.list.dataListButton.disabled = (matches.length - (page * BOOKS_PER_PAGE)) > 0;

// @ts-ignore
domElements.list.dataListButton.innerHTML = `
    <span>Show more</span>
    <span class="list__remaining"> (${(matches.length - (page * BOOKS_PER_PAGE)) > 0 ? (matches.length - (page * BOOKS_PER_PAGE)) : 0})</span>
`
```

As a solution to the redundancy , most of the functionality could be placed together since they server the same purpose, The Single Responsibility Principle (SRP) is applied here, which states that a class, a module, or a function should have one responsibility, and making changes to that one function should not affect anything else

---