Workshop - a simple book-reading web app

JavaScript events

1. Description

The goal of this exercise is to build a simple book-reading web application. The focus is on events, event handling and document manipulation using JavaScript. There are a small amount of new concepts, which will be explained when you get to them.

The HTML and CSS for everything necessary is already provided. Of course, if you happen to have a different aesthetic preference, feel free to change how it looks! This is an exercise after all 😃 (make sure to finish the JS part first though!)

For convenience, use the provided *bookmenu.js* file, since it's already linked to the document.

Tip: Follow the CSS classes to the letter, as they have already been pre-written. Otherwise, you risk your app looking all sorts of weird.

General hint: Each step builds upon the previous one. You can often reuse event bindings and functions, so think a bit and develop a strategy of approach before writing code.

One rule - no CSS allowed. You can solve some of the things below easily with CSS, but that's not the point.

Good Luck!

2. Tasks

1. Using the paragraphs in the document, generate a menu with in-document links, that scroll to the paragraph when clicked. Add them to the existing sidebar ('nav' element) like this: `ul > li > a`. That means - you first need to create a li element, and then create an a element inside the li element.

    - *Hint*: since `href` is an attribute, you can change it by accessing `.href` of the `a` element you create, assigning a value to it.
    - *Hint*: internal document links work by id - if you set the href to the id of the paragraph, it will lead to it when clicked.
    - *Hint*: the `+` operator also works on strings - '#' + 'something' results in '#something' (you'll need this one).

2. Add a click handler to all of the menu links in the sidebar. When a menu item is clicked, it should receive a css class of 'active'.

    - *Hint*: It's important NOT to prevent the default behavior here - the links are local, and we still want them to scroll to the appropriate paragraph.
    - *Hint*: It may also be wise to remove all 'active' classes from links before activating the one clicked on. Simply because it looks weird having multiple 'active' links 😃

3. Similar as with the previous task, paragraphs that are currently viewed / activated, should receive a class of 'activeP'.

- *Hint*: As with the links, simply remove all 'activeP' classes from the list of paragraphs before activating one. We still want to avoid weirdness.
- *Hint*: To make this simpler, have a look at the *haschange* event. It triggers when the active hash changes in the URL. Since we're changing it via the links, it's simpler to catch it and add the class to the appropriate paragraph.
- *Hint*: The event itself is not enough - but we can get the value of the active paragraph by using `window.location.hash`.

4. So far so good - but there's still some weirdness when we reload the page - namely, nothing activates on reload (and the URL doesn't change). So to fix that, re-activate the appropriate paragraph and link in the sidebar.

    - *Hint*: There's an event for that! Use the 'load' event on window and simply re-apply the classes to the appropriate link and paragraph.
    - *Hint*: Selecting the paragraph is easy - but selecting the link is a bit trickier. Luckily, `.querySelector` allows us to select by *attrbitues* too. So, to select something by its link value, we could `document.querySelector(a[href=someLinkHere)`. Also see hint 3 of task 1 for futher guidance 😃

5. BONUS ROUND: Also scroll the element into view on reload.

    - *Hint*: Granted, this one is totally new. However, each element that's currently out of view, can be scrolled into view by calling `.scrollIntoView()` on the appropriate element.

6. No book reading app is complete without a visible tracker where we're at. So, add a div element with a class of "tracker" to the document which simply shows the paragraph that's currently active.

    - *Hint*: At this point, we're already using 'hashchange'. Maybe just add some extra functionality to that?
    - *Hint*: For the sake of consistency, we should also do the thing from the hint above on page reload. Simply because we want the active paragraph to be visible, right?

7. First off, congratulations if you got this far. Now it gets really hard - but that doesn't mean you get to quit 😃 Modify the application so it shows and hides paragraphs when their link is clicked, instead of scrolling them into view.

    - *Hint*: `.style.display = 'none'` and `.style.display = 'block'` are your friends here. First, we'd need to hide all paragraphs, then detect which one is active, and then show that one on screen.
    - *Hint*: A small reminder - `window.location.hash` tells us directly which paragraph is active. Be careful however, because it's not always defined.

8. Still here? Good. Since we're showing a single paragraph, it might be wise to extend the functionality of this app with "previous" and "next" buttons, that to exactly that - show the previous and the next paragraph. Note that, the sidebar should still respond and update when "previous" or "next" gets clicked. Because this one is a bit bigger (not to mention harder), you even get some steps:

    1. Create a container div for the buttons, with a CSS class of 'pager'.
    2. Create two buttons, with "< previous" and "next >" as `innerText`. Append them to the container div at 1.

   3. Append the container div to the 'main' element.
   4. Define two functions (to be used as event handlers): nextPage and previousPage.
   5. *Hint:* we can get the information about which paragraph is active via `window.location.hash`. However, we care about the previous and next elements of that paragraphs (since we need to activate those). So, have a look at `.nextElementSibling` and `.previousElementSibling`. They give you exactly what you need.
   6. While writing the functions, think a bit about how to *activate* the links in the sidebar - we want things consistent, after all.
   7. We have a limited amount of paragraphs - so if we get to the first one and press previous, we should alert that this is the first paragraph. On the flipside, if we get to the last one and press next, we should alert that it is the last one.
   8. *Hint*: There are a ton of ways to do this. The simple one is by looking at the length of the paragraphs. The fun one is by detecting what type of tag comes next / previous. You can do that by: `nextElement.tagName === P`. That tells use whether the coming element is a paragraph or not. And we only care for those.

9. Yup, still more stuff here. Add a checkbox to the document that switches between the scrolling mode before 7 and the paging mode at 7 and after.

   - Create a div with a class of 'scrollCheck'. Add an input with a type of checkbox to it, and a label saying "Enable scrolling mode".
   - Also, no more hints for you. If you're this far, you'll figure it out 😉

10. We're slowly running out of things to do, but there's something more - favorites. Add a way of making a paragraph your favorite. Store that somewhere (hint: localStorage comes to mind). When the page loads, it always shows that paragraph first. We can only have one favorite at a time.

11. Done with 10? Make the page responsive. Currently it sucks on mobile screens (CSS is obviously allowed here).

12. Go have a beer or something. You're done here 😃