Exercise 13 – Functions and Closures

Objectives

In the first part of this exercise, you will create a simple tab system using an anonymous self-executing function. In the second part, you will explore strict mode.

Overview

The first part of the exercise will allow us to create a simple tabs system using self-executing functions and exploring closures. In the second part of the exercise, we will refine our code to use JavaScript's new strict mode.

This exercise is scheduled to run for around **30 minutes**.

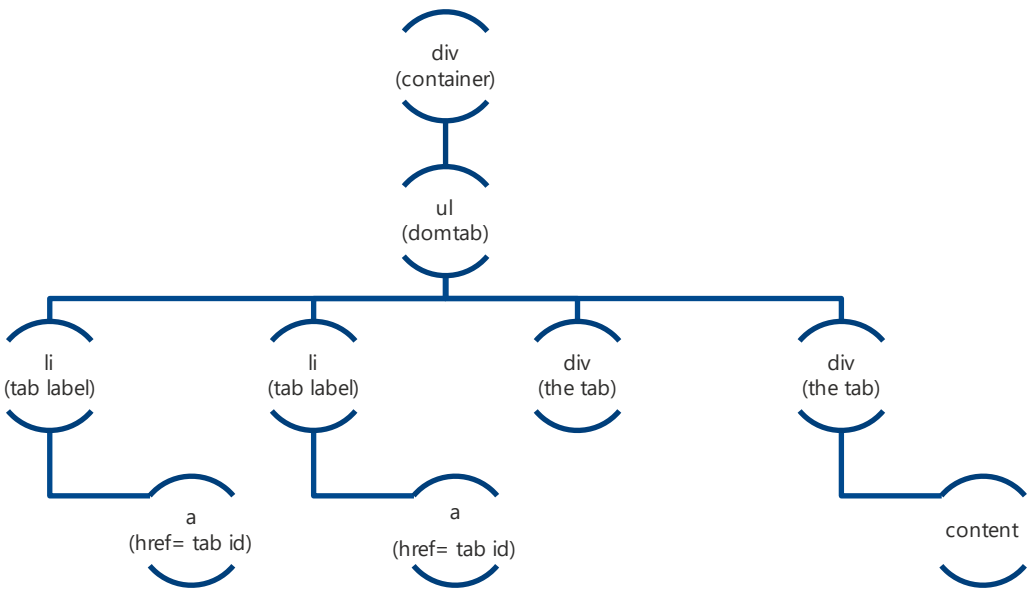
Exercise Set Up

1. In VSCode, open the file called **SelfExectuingFunction.html** from the **Starter** folder inside **Ex13** of the **Exercises** folder.
2. If you have rebooted your computer or stopped *live-server* from running since the last exercise, then use VSCode's integrated terminal to navigate to the **Exercises** folder and run the command **live-server**.
3. Open your browser (if it doesn't automatically) and navigate to **http://localhost:8080** and follow the path to open **Ex13→Starter→ SelfExecutingFunction.html**.

Part 1 - Validating Form Inputs

Exercise Instructions

1. Open the code for SelfExecutingFunction.html and examine the markup used for the tabs.



The container consists of a **<ul>** where each **<li>** contains an **<a>** with a **href** that refers to the **id** of a ** ** we wish to use as a tab. A CSS file styling the list and each tab is already in place.

2. In the <script> element, find the comment **// Part 1** and create 3 variables:
   a) **container** set to be the **ul** element with the **class domtabs**;
   b) **divs** set to be an *array of elements* that are a **div** that is *a direct child* of a **div**;
   c) **tabs** set to be an *array of elements* that are an **a** that *are descendants* of a **ul** with a **class** of **domtabs**.

These variables will retain the pointers necessary to work with the DOM. Currently, they belong to the script scope. That is okay if you want to pass them between multiple objects. It does tend to overfill the scope space and, more worryingly if we are using a lot of JavaScript, overwrite variables unexpectedly.

3. Inspect the code at the bottom of the script tab under the comment **// Leave me alone!**.
4. Save your code and check the browser console.

You should find that a WARNING! is displayed informing you that the variable tabs is in scope.

An **assert** function is useful when you start dealing with more complex scope and longer pieces of code. You assume everything is going well, making a Boolean check. If the check is passed, then nothing happens. If it fails, an error is displayed in the console. Test the page in Chrome – it will warn you the variables you need are currently in scope. This is the key thing we want to remove by switching to self-enclosed execution.

Next, we will create a self-executing enclosure for the rest of our functionality. This function will fire as soon as we hit the script block.

5. Enter the following code *AFTER* the declarations but *BEFORE* the **console.assert** to test everything is working:

```
(() => { console.log("Self executed function working"); } ();
```

6. Save your code and check the console in the browser. There should be a **console.log** with the message in it.

This has shown us the pattern works, though we don't yet have parameters to pass in, nor do we have any real functionality. However, something very important is happening here. The rounded brackets surrounding the function tell the browser to execute now. The function has no name, it is anonymous. The functionality executes as soon as we hit it, in page processing it encloses the inner function call. After that, it can never be called directly.

Everything else we are going to do will sit inside the first set of rounded brackets. Remember that as we move forward!

7. Move the variables you created into the anonymous function body and remove the **console.log** call.
8. After the variable declarations, add another self-executing function that takes **d** as an argument and is passed in **divs** as parameter.

As soon as the code parser hits this line of code, it begins to execute this new anonymous function. Importantly, we are maintaining scope and thus the variables **div**, **tabs** and **container** are available to its enclosing block. The parameter we are passing in is **divs**, which is mapped to the argument **d** inside the function call.

Within the body of the anonymous function, we will move through the **divs** array and hide every '**tab**' we have created.

9. Create a loop for the array of **divs** that sets the **style.display** property of each to "none".

This effectively takes the content out of the flow of the document on the screen.

10. Test the page in the browser once again – the three 'tabs' areas will now be gone. Investigate the live DOM using the Chrome development tools and you will find the style property reflects this.
11. Next, check the console. The **WARNING!** message you created an assert check for earlier in this exercise is no longer showing. It has been replaced by an INFO message informing you that **tabs** in not in this scope.

This reflects that **tabs** is no longer in scope. tabs is now a variable of the enclosing anonymous function call. Once this function has executed, the self-executing anonymous object moves out of scope and is removed. This is one of the most powerful features of this development pattern. The object is removed from the stack and does not remain, a never-to-be-used-again function on the global stack.

We can now see how beneficial closures are, but we will take this a little bit further to create the functionality to show a tab when a link is clicked. First, we need a third self-executing scope block beneath the function we created in step 7.

12. The new self-executing function should:
    a) Take an argument of **t**;
    b) Loop through this array and:
        i) Add a *click event listener* to each element with *an anonymous function* that: ***Note: Do not use an arrow function here!***
        ii) Takes an argument of **e** (an event);
        iii) Logs out the text **"test";**
        iv) *Stops the default action occurring* for the event;
        v) Does not bubble;
    c) Be passed the parameter **tabs.**

This function takes in the array tabs and adds an event listener to the click function of each <a> element. Currently, it simply prevents the default behaviour and fires an alert.

13. Test the function in the browser. Each of the three tabs will now fire a click event and fire the console logging.

Associating events are always great candidates for closures. We want to ensure that all relevant objects are hooked up to an event, but we want to ensure no other objects have access to the event-raising function. This technique allows a scope controlled self-initialising encapsulated event model.

Now we will add the real functionality to our tabbing control. The key to this functionality is to extract the **#id href** value of the URL. The **href** property is interesting. Even though we only have values such as **#about** in our code, once the page is served, this becomes a *full http url string*.

14. Test this by changing the value of the console.log to **this.href** then testing it in the browser.

Remember that this will refer to the object raising the event. We do not need to know exactly which <a> element we are adding the code to, just that it has an href property.

As a result, the href property is pretty useless right now. We need to work with that string and extract out only the id of the URL dumping the unnecessary part of the URL. The string manipulation functions will achieve this quite happily.

15. Remove the console.log and replace it with 3 lines of code:
    a) The first should declare a variable **url** and set it to the **value** of the **href** for this link;
    b) The second should declare a variable named **s** and set this to the *position of* the **#** in **url** *(HINT: Use indexOf);*
    c) The third should declare a variable called **id** and set this to the **substring** created when we take **url** from the value of **s** to the *end* of the string.

id will now hold a string only holding the id.

16. Use **id** as a selector value to get hold of the matching .

We now have a pointer to the appropriate DOM object. The rest of our work is CSS manipulation.

17. Change the properties of divToShow so that it has:
    a) An *extra* **className** of domtab;
    b) A **display** property of **block**;
    c) A **clear** property of **both**;
    d) A **backgroundColor** of **rgb(102, 102, 102)**.
    Note: Remember that some of these require manipulation of the element's style object!
18. Test the page in the browser and click on the links in turn. You will now find the links will once again display. Unfortunately, they do not disappear once we click on another tab. That's the next thing to resolve!

Before the statements to set the properties of **divToShow,** we will check if any ** ** elements being used as tabs are visible and remove them if necessary. To do this, we need to examine the **divs** array again. This array was defined way before in the original anonymous function block. We are still in its function chain, so this is still available to us. Yay to closures!

19. Before the 4 statements that change properties of **divToShow**, write a loop that checks to see if the **display** property of each element in the **divs** array is set to **none** and *if it isn't*, set it to **none**.
20. Now test your code in the browser. You will find the correct tab will show after the previous tabs have been hidden.

Part 2 (if you have time) - Examining Strict Mode

Strict mode stops us working with bad practice and deprecated approaches, such as incorrectly-scoped variables.

1. Open the Solution file in the browser (assuming that you have not made any changes to the code during the exercise!).

You will notice that when you try to click on the tabs, nothing appears on the page and that there are console errors.

2. Use your knowledge of strict mode to clear these errors.