Session 6 exercises

Advanced JavaScript for Web Sites and Web Applications

Exercise 1

Download the workshop 6 files from Moodle.

Extract them in your workspace and open `exercise1.html` and `vat-calculator.js` in your editor. You should also open `exercise1.html` in your browser.

Review the code in `vat-calculator.js`.

The `VatCalculator` module is a simple module with 3 public methods that allow you to:

- increase the stored rate of VAT
- get the current VAT rate
- Add VAT to a number

The module creates a custom event when the IIFE is executed (`vatEvent`). This event is vital to the operation of the module. This event is fired whenever the VAT rate is changed.

Code outside the module listens for this event, and runs the `updateVatDisplay` function when it is fired. The `updateVatDisplay` function simply logs a message containing the current VAT rate in the console.

a) Create an init() method

1. Create an `init` method within the module that *creates* the custom event object. A pointer to the `init` method should be added to the module's returned object.
2. Extend your `init` method so that it accepts an argument: the VAT rate to use within the module. Inside the method, store the passed value in the module's `rate` variable. If you like, you can use a config object for this, but, if you do, you will need to modify the module code.

b) Import global objects

In the module's `raiseVat` method, the *document object* is used when the event is dispatched.

1. Amend the module definition so that the *document object* is passed to the IIFE as an argument. You can alias the object to `d` or similar.
2. Modify the `raiseVat` method of the `VatCalculator` module so that it uses the *alias* you have given to the document object, as opposed to referencing `document` directly.

Exercise 2

Open the exercise 2 files in your editor and review the code.

- `math.js` - a module that provides a simple way of adding or subtracting numbers and returning the result.
- `index.html` - the HTML interface for a simple calculator.
- `calculator.js` - this is where you will place your code

We want to link the functionality contained in `MATH` with the interface from `index.html`, creating a simple calculator.

- `In calculator.js`, you will add another module that manages the interface and events for the calculator
- The new module will extend the MATH module using whichever method you prefer:
  - extend MATH directly,
  - use a sub-module
  - create a new module that depends on MATH.

The second module's structure will be fairly simple...

- We will have an `init` function where we register our event handler
  - To manage the user's "clicks"
- We will have an event handler function
  - To manage the event delegation
- The module will return a pointer to the `init` function, but nothing else.
  - All functionality will be triggered via *clicks*
- The event handler function will check which button was clicked, and perform an appropriate task.
  - Using event delegation!!!

The desired application behaviour:

To perform a calculation, the user will:

- Click a number button (or more than one button if number greater than 9)
- Click "add" or "subtract"
- Click another number button (or buttons)
- Click "="

After clicking =, the result of the calculation will be displayed in the input field.

Hints:

- HTML `id` attributes have been used to identify different elements in `index.html`, which should make it easier to select them from your code (`getElementById`).
- You will need to set a *flag* variable so that you know whether the user clicked the *add* or *subtract* button as part of their calculation.

- When the user clicks on the = button, you will use this flag variable to either add or subtract the input field value from/to the `MATH` total.
- Note that `MATH` expects number variables. However, `element.value` always returns a string (even if the value is a number).
  - You can parse a string as a number using `parseInt('stringnumber')`.

The event handler function will be very busy... but you can simplify it with some event delegation. (Note how the buttons all have a `data-type` attribute... you can probably make use of this)

When your handler *hears* a click event:

- if the button was one of the digits:
  - Append its value to the view (i.e. the input field).
- if the button was *clear*:
  - Clear the view; set the total of MATH to zero.
- if the button was *add*:
  - Set flag to “add”; use the set method of MATH to change the value of `total` to match the value from the input field; clear the view.
- if the button was *subtract*:
  - Set flag to “subtract”; use the set method of MATH to change the value of `total` to match the value from the inut field; clear the view.
- if the button was *result*:
  - Check if the flag is set to *add* or *subtract*; use the `add` or `sub` method of `MATH` to alter `total`; update the view with the result.