Session 4 exercises

Advanced JavaScript for Web Sites and Web Applications

Exercise 1

Download the *workshop4.zip* file from Moodle and extract it to your workspace.

In *workshop4.js*, a skeleton module has been provided for you to use as a “template” for the modules you create (i.e. make a copy of it for each module you build).

In *workshop4.js*, using the *module pattern*, or the *revealing module pattern*, build a simple calculator module with the following methods:

- A method that will **add** two arguments that are passed to it and return the result.
- A method that will **multiply** two arguments passed to it and return the result.
- A method that will **divide** two arguments passed to it and return the result.

The module should *return* an object containing the 3 methods that you have defined (i.e make them public methods).

Remember, if you are using the *revealing module* pattern, you should return *references* to the methods, not the methods themselves.

Assuming you call the module `Calculator`, you would call the methods like this:

```
var result1 = Calculator.add(7, 8);
console.log(result1); // result = 15

var result2 = Calculator.multiply(3, 8);
console.log(result2);// result = 24
```

Exercise 2

Part 1

Use the revealing module pattern to re-implement the exercise we saw last week:

Exercise 7: using `this` to create chainable methods

(You can download the solution from Moodle if you have not completed it)

Note, when you are finished, the module should work exactly as it did before (i.e. your methods should remain chainable!)

Remember, when you call the module methods from the global scope, you will be calling them in the context of the returned object. In other words, “this” will be a reference to the returned object, not the *module*.

Therefore, “this” will be useful when you need to make the methods chainable… but possibly not so useful when you want to interact with the “total” variable.

Part 2

When you have the basic module working correctly…

Previously we were setting the initial value of `total` to `0` and storing it as a variable in the module. Now we want to allow the outside code to set the value of `total` (if it wants to)

Add a config object to your module with a single property: `total`. The default value for this property should be `0`.

Add a method to your module that allows the value of `total` to be set via a *config object*, something like:

```
var myConfig = {total: 10};
myModule.setConfig(myConfig);
```

You will have to ensure that your module methods reference the value stored in the config object property.

Test your object both with a custom config object, and without one (`total` should default to 0)

Exercise 3

Build a simple shopping list module with *public* methods that allow you to:

- add an item to the shopping list (`add`)
- remove an item from the list (`remove`)
- return the list itself (`getList`)
- return the number of items in the list (`count`)

The list can be stored in the module as an array.

The items that can be added to the list can be simple strings (“apple”, “banana”, “pear”, etc.)

Caveats:

- The `remove` and `add` methods should be chainable (i.e. `return this`).
- The `count` and `getList` methods do not have to be chainable (they return values).
- You will want to make sure an item doesn’t already exist in the list before you add it! (using `indexOf`?)
  - What do you think your `add` method should do when it encounters a duplicate item?

When done, you should be able to interact with your module like this:

```
var itemsAdded = myShoppingList.add("apple").add("banana").count();
console.log(itemsAdded); // 2 (i.e. the value returned by "count");

var myList = myShoppingList.getList();
console.log(myList); // ["apple", "banana"];
```

Hint: Don’t let the complexity of real-world shopping baskets distract you from the task. What you are actually bulding is a simple list manager. However, the resulting code could easily become a shopping basket!