ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

SESSION 9 - MORE PATTERNS AND EXAMPLE

1. Facade Pattern
2. A larger app

1. Facade Pattern

Let's have a look at another pattern, the facade pattern, used along side the module pattern.

Facades hide how they work internally from the outside and provide an api type layer for other parts of the application to interact with it.

Mixed with the module pattern, they are a good way to break the code down further into small modules, hidden to the outside and expose only a top level API.

Ideally, the API we publish would never change, even if the inner functions do.

This pattern is a good way to provide a constant, stable API for other parts of the cide.

```
var APP =  (function() {

    var privateObject = {
        i:5,
        get : function() {},
        set : function() {},
        run : function() {},
        jump: function() {}
    };

    return {
        facade: function (args) {
            privateObject.set(args.val);
            privateObject.get();
            if (args.run) {
                privateObject.run();
            }
        }
    };
}());

APP.facade({run: true, val:10});
```

Also remember that if you don't need private methods using the module pattern, if you don't need to "hide" sections of code from the outside, you can use a simple object literal notation:

```
var APP =  {
  i:5,
  get: function() {},
  set: function() {},
  run: function() {},
  jump: function() {}
};
```

2. A larger app

Let's build a larger app with multiple modules.

We want to build an Ecommerce app. And the first module we need is a basket module.

We'll start with a main app, that holds various functions for common things across the site, and add the basket module to this.

part 1

Start with a `main.js` file. This file will hold generic scripts for the application, that apply globally over the site. The pattern will be empty for now, but we will come back to it.

Use the module pattern, call the main application `MAIN`.

```
var MAIN =  (function() {

  var

    init = function () {
    };

  return {
    init: init
  };

}());

MAIN.init();
```

part 2

Now create a `basket.js` file.

In this file, we're extending the `MAIN` object and adding a basket method to it.

This method itself will use the revealing module pattern.

```
MAIN.basket =  (function() {

  var
    init = function () {

    };

  return {
    init: init
  };

}());

MAIN.basket.init();
```

This module needs one important variable, the `basket`. Make sure it is declared at the top of the `var` statement.

```
MAIN.basket =  (function() {

  var

    basket,
    init = function () {

    };

  return {
    init: init
  };

}());

MAIN.basket.init();
```

part 3

The module works the following way:

1. the basket is an array
2. you can add to the array, by adding a new item, or incrementing the quantity of an exisitng item
3. you can remove an item
4. you can count the total number of items in the array (using the quantities)
5. you can count the total to pay for the basket

Items can be objects:

```
{
  id: "item id",
  price: 200,
  description: "string",
  url: "url",
  quantity: 3
}
```

And the basket is an array of items:

```
basket = [
  {
    id: "item id",
    price: 100,
    description: "string",
    url: "url",
    quantity: 1
  },
  {
    id: "item id",
    price: 200,
    description: "string",
    url: "url",
    quantity: 3
  }
]
```

In JavaScript, this means we need to:

1. select the DOM elements we need
2. retrieve the basket array
3. declare the different functions that we need.
4. declare the different event handlers to map the DOM elements to the functions

We need the following functions:

- `getBasketItems`
- `removeFromBasket`
- `addToBasket`
- `updateBasket`
- `countItems`
- `basketTotal`
- `setUpListerners`

`removeFromBasket`, `addToBasket` and `updateBasket` will all need to loop through the basket and find an item.

We can create another function called `findInBasket` to do this.

`getbasketitems`

This function needs to retrieve the basket array. We won't populate it for now, but it needs to output the current basket as an array.

The basket could be retrieved through Ajax, or could be added to the DOM in a basket var that we'll retrieve. Or we could parse the current html to create this array (not efficient...).

However the basket array is populated, the function will set a variable `basket` with the content it retrieves.

```
getBasketItems = function () {

  // retrieve the basket array through method of choice

  var retrievedBasket = [
    {
      id: "item id",
      price: 100,
      description: "string",
      url: "url",
      quantity: 1
    },
    {
      id: "item id",
      price: 200,
      description: "string",
      url: "url",
      quantity: 3
    }
  ];

  basket = retrievedBasket;

},
```

findinbasket

The functions needs to loop through the basket, passed through as an argument and find an item, passed as an argument.

It needs to return the position of the item in the array, if it finds one, or `null`.

It's a good idea to add comments to function like this, that indicate what the function takes and what it outputs.

There's different methods and conventions of doing this, we'll just use a simple commet for now:

```
// input: the type of input for the function
// output: what the function ouputs
```

```
findInBasket = function (basket, item) {

  // input: basket array, an item object
  // output: the index of item in array, or null

  for (var i = 0; i < basket.length; i++) {
    if (basket[i].id === item.id + "") {
      return i
    }
  }

  return null

}
```

`countitems`

This function needs to loop through the array and add all quantities together.

It takes tha basket as an input and returns a number.

```
countItems = function (basket) {

  // input: basket array
  // output: the number of item

  var total = 0;

  for (var i = 0; i < basket.length; i ++) {
    total = total + basket[i].quantity
  }

  return total

}
```

addtobasket

This functions takes the basket as an input, and the item to add, as an object.

It needs to loop through the basket and search for the item (using `findInBasket`).

If it finds an item, it needs to increment its quantity by one, the item doesn't exist in the basket, it needs to push to the array.

Remember that `findInBasket` returns `null` or a position.

```
addToBasket = function (basket, item) {

  // input: basket array, an item object

  // item object looks like:
  // {id: "item id", price: 100, description: "string", url: "url",quanti

  var itemFound = findInBasket(basket, item);

  if (itemFound) {
    basket[itemFound].quantity ++;
  }
  else {
    basket.push(item);
  }

},
```

removefrombasket

This function takes the basket and an item (as object) as inputs.

It needs to loop through the basket, look for the item (using findInBasket) and remove the item if it exists or do nothing if it doesn't.

```
removeFromBasket = function (basket, item) {

  // input: basket array, an item object

  var itemFound = findInBasket(basket, item);

  if (itemFound) {
    //removes element from basket
    basket.splice(itemFound, 1);
  }

},
```

updatebasket

This function needs to update the quantity of an item.

It is used when the quantity of an item is changed by the user directly (using the input fields).

Again, it takes the basket and item to update as inputs. It also takes the new quantity as argument.

It needs to loop through the basket, and if it finds the item, update its quantity.

```
updateBasket = function (basket, item, newQuantity) {

  // input: basket array, an item object, a quantity as a number

  var itemFound = findInBasket(basket, item);

  if (itemFound) {
    basket[itemFound].quantity = newQuantity;
  }

},
```

baskettotal

This function needs to loop through the basket and add the total of all items.

```
basketTotal = function (basket) {

  // input: basket array
  // output: the total amount, as a number

  var total = 0;

  for (var i = 0; i < basket.length; i++) {
    total = total + basket[i].price * basket[i].quantity;
  }

  return total
},
```

set up listerners and init

We need two final functions, one to hold all our event listeners, one to initialise the module.

We won't declare the actual event handlers, but here's what they need to do:

Any remove button needs to:

1. removeFromBasket(), passing basket and the itemID. Item ID can be retrieved on button
2. basketTotal() and update the relevant dom element: totalAmountHolder
3. countItems() and update relevant dom element: totalItems

Any add button needs to:

1. addToBasket(), passing basket and the itemID. Item ID can be retrieved o button
2. basketTotal() and update the relevant dom element: totalAmoutHolder
3. countItems() and update relevant dom element: totalItems

When the quantity is changed in the interface:

1. run updateBasket(), passing basket, item and new quantity
2. basketTotal() and update the relevant dom element: totalAmountHolder
3. countItems() and update relevant dom element: totalItems

```
setUpListerners = function () {
  // when hit a remove button, run:
  // 1. removeFromBasket(), passing basket and the itemID. Item ID can be
  // 2. basketTotal() and update the relevant dom element: totalAmountHol
  // 3. countItems() and update relevant dom element: totalItems

  // when hit an add button run:
  // 1. addToBasket(), passing basket and the itemID. Item ID can be retr
  // 2. basketTotal() and update the relevant dom element: totalAmountHol
  // 3. countItems() and update relevant dom element: totalItems

  // when keyup on input for each item:
  // 1. run updateBasket(), passing basket, item and new quantity
  // 2. basketTotal() and update the relevant dom element: totalAmountHol
  // 3. countItems() and update relevant dom element: totalItems

},
```

init

Finally the `init` function needs to call `getBasketItems` and `setUpListerners`.

At the moment, the only methods this module needs to expose publicly is its `init` function.

```
init = function () {

  getBasketItems();
  setUpListerners();

}
```

part 4

At this point, our application has two files, `app.js` that contains generic scripts for our app, and `basket.js` that holds scripts specific to the basket.

Both files need to be loaded in the html through a script tag, or concatenated using a build tool before being pushed, or the basket module needs to be loaded asynchronously etc.

Let's continue adding modules!

let's add another module that creates a carousel on elements it finds.