Exercise 14 – Object Orientated JavaScript

Objectives

The following exercise has two sections. In the first, you take the slideshow we have been building and extract it into a separate JavaScript file for reusability. In part two, you investigate some simple JavaScript inheritance.

Overview

In Part 1, you re-factor the build a slideshow object, and work with private and scope-based variables to a significant degree.

In Part 2, you will examine how inheritance works within JavaScript.

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

Exercise Set Up

1. In VSCode, open the file called **ImageReusable.html** and **SlideShow.js** from the **Starter** folder inside **Ex14** 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 **Ex14→Starter→ImageReusable.html**.

Part 1 - The SlideShow object

Exercise Instructions

The initial set up for this exercise has an HTML file that has a linked CSS file and a linked JavaScript file. The SlideShow.js file is where we will create the basis for out object. The skeleton for the class has been provided. Remember that in the underlying JavaScript, the constructor actually creates a function with the name of the class and any functions defined within the class are added to its prototype which is where its behaviours will sit. The file also contains an array of objects that represent the images we will use in the slide show. The actual images can be found in the images folder.

1. Complete the constructor for the **SlideShow** class by adding 6 properties all set to **null**:
   a) **theGallery**;
   b) **theImage**;
   c) **imageCaption**;
   d) **next**;
   e) **prev**;
   f) **startPosition**.
   Remember to use the this keyword here.

As a result of this code, when we create a new element, the constructed object will gain a copy of these four variables, all rather boringly set to null at the moment. Let's check to see if we can create a **SlideShow** object correctly.

2. Under the code provided, add 2 lines of code:
    a) Declare a variable called **mySlideShow**, setting it as a *new instance* of SlideShow;
    b) Output **mySlideShow** to the console using **console.dir**.
3. Check the output of your work in the browser and its console.

We have proved that **mySlideshow** now contains the relevant properties for us to write to. Next, we want to set those properties to be something other than null.

4. Amend the **constructor** so that:
    a) It takes **5** arguments, representing the *first five properties* of the object;
    b) Sets the *initial values* of the *first five properties* to the *value passed in*;
    c) Generates a *random number* based on the *length of* **theGallery** and sets the *value* of **startPosition** to it.

   Now if we pass parameters into the constructor, we'll be in business!
5. Add to or modify the code at the bottom of the **SlideShow.js** file so that it:
    a) Declares a variable **image** and sets it to *the element* with an *id* of **slideShow**;
    b) Declares a variable **caption** and sets this to *the element* with an *id* of **captionText**;
    c) Adds the parameters **images**, **image** and **caption** in the constructor call for **mySlideShow** with variables **images**, **image** and **caption**.

With the parameters passed in, we can now enter the behaviour. The constructor has added instance specific state (i.e. each instance of the slideshow has its own theGallery, etc.). That is important as each object needs to have its own idea of state so it can remember different things.

6. Add 3 functions to the **SlideShow** class for **_displayImage**, **previousImage** and **nextImage.**

Unless you solved it in the **Further activities** part of the **Events** exercise, we were repeating ourselves in the code with the constructor, next and previous button also using the same code to write to the image. In this implementation, we will remove the repetitive code and place it inside the **_displayImage** function.

7. Before we get to that, what is the purpose of the underscore at the start of the key name?

The biggest change to the code for the method is that we must prefix the variable access with the **this** keyword. This keeps the execution to the specific instance of the **SlideShow** we have created.

8. Add code to the body of the **_displayImage** function that:
    a) Sets the **src** property of **theImage** to the **src** property of *the element* in **theGallery** at the *index* of **startPosition;**

   b) Sets the **alt** property of **theImage** to the **alt** property of *the element* in **theGallery** at the *index* of **startPosition;**
   c) *If a* **caption** *exists on the element* in **theGallery** at the *index* of **startPosition**, sets the **innerHTML** property of **imageCaption** to it, otherwise sets the **caption** to the *same element's* **alt** property.
9. Save your code and refresh the page a few times to ensure that a different image is randomly shown. Check your console for any errors and rectify before moving on.

Next, we will add functionality to the previous and next buttons.

10. Add code to the **previousButton** function body that:
    a) *Decreases* the value of **startPosition** by 1;
    b) Sets **startPosition** to *one less than* the **length** of **theGallery** IF **startPosition** is *less than zero*;
    c) Calls the **_displayImage** function.
11. Before the code that declared **slideShow**, add 2 lines of code that:
    a) Declare variables **next** and **prev** and set them to the *element on the page* with the **id** of **next** and **prev**;
12. Add these 2 (**next** and **prev**) variables to the instantiation call for **mySlideShow**.
13. In the **constructor**, add values to the **onclick** property of **next** and **prev**. You should use *an arrow function* that takes *no parameters* and *executes* the appropriate **nextImage** or **previousImage** function call.
14. Save your code and test that the buttons work with no errors.

Congratulations, you now have an image slide show, created in JavaScript that can be used on any page as long as a suitable images array are supplied and the page matches the ids used to select the elements within it.

Part 2 - JavaScript and Inheritance

Exercise Set Up

1. In VSCode, open **Pets.html** from the **Starter** folder for **Ex14** 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 **Ex14→Starter→ Pets.html**.

Exercise Instructions

1. In the script block define a **class** called **Pets**
2. Create a **constructor** for the class that takes *3 arguments* and initialises them. The properties are **name**, **species** and **hello** and should be *notionally private*.
3. Under the class, instantiate a Pet:
   a) The variable name should be **myFirstPet**;
   b) Arguments for the constructor are **'Fido'**, **'Corgi'** and **'Woof!'**.
4. Create a *reference* to the *element* with an **id** of **output** called **output**.
5. Declare a variable called **outputString**, initialised to be an *empty string*.

6. Use a **for...in** loop for **myFirstPet**, *appending* each *property value* to **outputString** *inside a paragraph tag*.
7. *Append* the value of **outputString** to the **innerHTML** of **output**.
8. Save your code and test the output in the browser. It should read out the three values from the constructor.

The next stage is to add some behaviour to the class by adding functions to it. Remember that the underlying JavaScript will be adding these functions to the prototype of Pet.

9. *In the class*, add a function called **sayHello**, it should:
   a) Log out the value of **_hello**.
10. Add a call to **sayHello** on **myFirstPet** *under the code for part 7.*
11. Save your code and check that the expected value is on the console.

Well done, you have created a simple object, but it would be great to take this further and work towards some object inheritance.

12. Under the last code you wrote, add *another* **class** called **Cat** that *inherits* from the **Pet** class.
13. Create a **constructor** that takes *4 parameters*: **name**, **hello**, **breed** and **whiskerLength**.
14. Make a **super** call, passing in **name**, **"Cat"** and **hello**.
    Note that we are making species a constant here as it will not change for any instance of Cat.
15. Set the remaining properties (notionally private) to those passed into the constructor (i.e. **breed** and **whiskerLength**).
16. Under the class, instantiate a **Cat** object and call its **sayHello** method - you can use anything you choose as the arguments.

Next, we want to extend the Cat to have a cat-only behaviour.

17. Cats are renowned for their ability to sleep, so add a **catNap** function to the **Cat class,** it should *log out* the *cat's name* and a number of 'z's.
18. Add a call to the cat's **capNap** function under the **sayHello** call at the bottom of your code.
19. Save the code and check that the output on the console is as expected.

Before ES2015, this process was much more involved. To inherit in this way your code would have looked like:

```
// The Pet class
function Pet(name, species, hello) {
    this._name = name;
    this._species = species;
    this._hello = hello;
}
Pet.prototype = {
    sayHello: function () {
        console.log(this._hello);
    }
}
```

```
// The Cat class - properties inherited from Pet
function Cat(name, hello, breed, whiskerLength) {
    Pet.call(this, name, "Cat", hello);
    this._breed = breed;
    this._whiskerLength = whiskerLength;
}
// Inherit the Pet prototype in Cat
Cat.prototype = new Pet();
// Extend the Cat prototype
Cat.prototype.catNap = function () {
    Console.log(this._name + ': zzzzz');
}
// Create instance of cat and use behaviours
var myFirstCat = new Cat('Meg', 'Miaow!', 'Tabby', 15);
myFirstCat.sayHello();
myFirstCat.catMap();
```

Now we will work towards polymorphic behaviour in JavaScript, introducing overriding and type checking. We will type check first...

20. Create an array called **petStore** assign element 0 to **myFirstPet** and element 1 to **myFirstCat**.
21. Use a **for...of** loop to iterate the **petStore**, calling **sayHello** for each element.
22. Use a *selection statement* to see if the current element is an *instance of* **Cat**, calling **catNap** if it is and logging out **"I can't catNap"** if it isn't.
23. Save the code and check that the output on the console is as expected.

Overriding behaviours defined in parent classes is as easy as writing the method we want in the class that we want to override the behaviour. Let's override the behaviour for **sayHello**.

24. In the **Cat** class, create a function called **sayHello**.
25. Make this function *log out the value of* **_hello** with the *additional string* of **"from a cat called"**, finishing the statement with the **_name** of the cat.
26. Save the code and check that the output on the console is as expected.