

# Exercise 13 – arrays, functions and objects

## Objective

In this series of exercises, you will explore the building blocks of reusable code, arrays, functions and objects.

**This is the most important part of the course to date, so please ensure you are happy with the code you are writing. If you have any questions, talk to your instructor.**

## Overview

In Part 1 of the exercise, we will introduce indexed arrays for the first time. When we introduce functions in Part 2, the focus will be on variable scope and the call stack. Finally, in Part 3, you will create object literals and use them in an array. This exercise is scheduled to run for around **40 minutes**.

## Exercise instructions

### Part 1 – arrays

Programming is all about arrays. As we will see later in the course, the entire basis of HTML and JavaScript programming is about searching through and manipulating arrays.

1. Arrays are a type of object, as we will explore later in the exercise. There are two ways to create the object, either with a new `Array()` keyword or a set of square brackets. We at QA, assume you want to write less not more code, so will take the short hand approach.
2. Open **ArraysAndFunctions.htm** from the starter folder of **Exercise 13** and within the script block add the following code:

```
let sentence = ["Luke", "I", "am", "your", "father"];
```

That's right, we are introducing the most important concept in programming with the most important line in cinema history!

3. What is the outcome of the code you just entered?

- 
4. We now have an array of strings. If we do not mind the commas in the way, we can print this to the console:

```
console.log(sentence.toString());
```

5. Test the page in the browser. You should see the entire contents of the array is displayed, but we need to get rid of those commas. We could just do this as sequential code, which has been the approach used in the course so far. It seems likely though that we, at some point, will want to print a sentence to the screen again, so we should use a function to assist us.
6. Before we get to that, let's be sure you are comfortable with accessing elements from an array. Use `console.log()` to display the first element of the sentence array. Refer back to your notes or shout for your instructor if you need help.
7. Once you have done that, enter the following code:

```
function displayArray(array) {  
    let output = "";  
    //additional code to go here  
    return output;  
}
```

8. In the above code, you have created your first function. It takes a parameter that we have helpfully called **array** to make it easier to work with. Shortly, we will work with the code to build the output string but again we have followed best practice and initialised the variable to be a string. Once we have finished working with the array, this function returns an array. The return is optional – if you failed to add it, the closing curled brace would be an implicit return.
9. We will now add a loop that iterates over the array. In the previous exercise, you used a `for` loop but entered a fixed value for the number of loops. Indexed arrays helpfully provide a length attribute, so we can create dynamic loops. Enter the following code:

```
for (let i = 0; i < array.length; i++) {  
    output += sentence[i] + " ";  
}
```

10. The first pass of our function is now written, so we need to test it. Remember that our function returns a value, so we need to assign its result to a variable and display that through `console.log`.
11. However, something as important as this statement surely needs to end with an exclamation mark, so add an `if` statement that checks if this is the last element and appends an exclamation mark to the end of the output string. (Hint: length may be of use here.)
12. Again, test it in the browser. It works, but there is a space before the exclamation mark. Consider how you could use an `else` statement with the `if` to not add a space if it is the last entry in the array.

## Part 2 – functions

In this part of the exercise, you will investigate variable scope within functions and the call stack of functions within JavaScript.

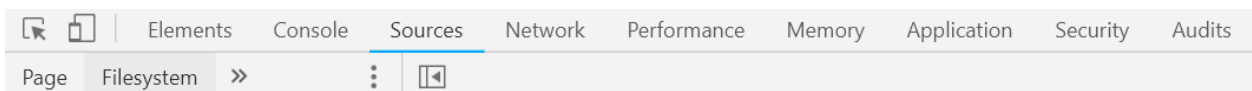
13. Open **FunctionScope.htm** from the start folder of **Exercise 13**. Four script blocks have been prepared for you to understand how local and global variables work in a slightly different way.
14. Within the script block, enter the following simple functions:

```
function test() {  
    flag = true;  
    console.log(flag);  
    test1();  
    console.log(flag);  
}  
  
function test1() {  
    flag = false;  
    return  
}  
  
test();
```

15. A simple nested function call that alters the value of the **flag** variable. Note what you think the value of the two console.log calls are going to be:

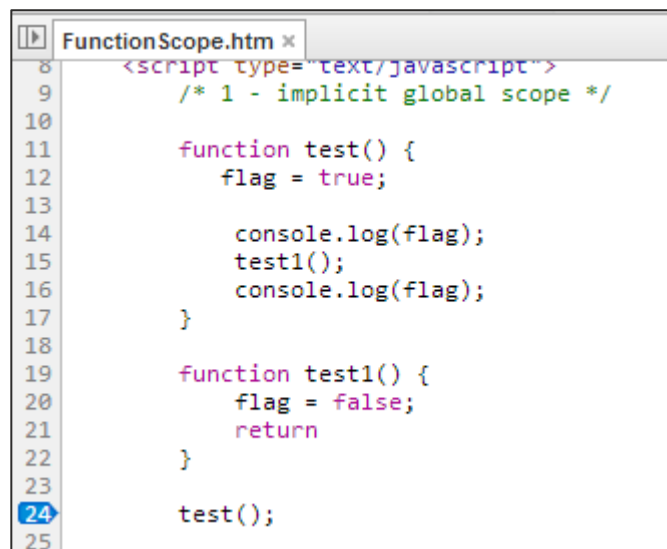
- 
16. Test the page in Chrome to verify your results.

17. Now, we are going to delve into the debug functionality of Chrome and learn a little more about JavaScript variable scope. Chrome's developer tools change very quickly; as a result, things may appear slightly different when you press F12 to bring up the developer console. For now, press F12 and go to the "Sources" panel – you should see:



18. You should be able to see the code for the **FunctionScope** page. If not, there is a folder icon in the top left corner of the panel, you can use this to select the script resource you want to debug.

19. Along the left hand side of the screen, you will see line numbers for the code. Click on top of the line number adjacent to the `test()` function call.



```
FunctionScope.htm x
8  <script type="text/javascript">
9      /* 1 - implicit global scope */
10
11     function test() {
12         flag = true;
13
14         console.log(flag);
15         test1();
16         console.log(flag);
17     }
18
19     function test1() {
20         flag = false;
21         return
22     }
23
24     test();
25
```

**You have just set a breakpoint. Breakpoints allow us to introspect the code, pausing the normal execution to examine what is happening inside the code.**

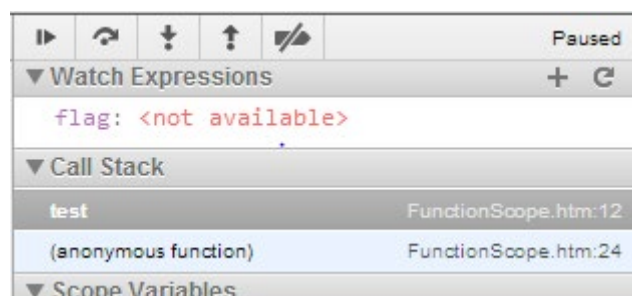
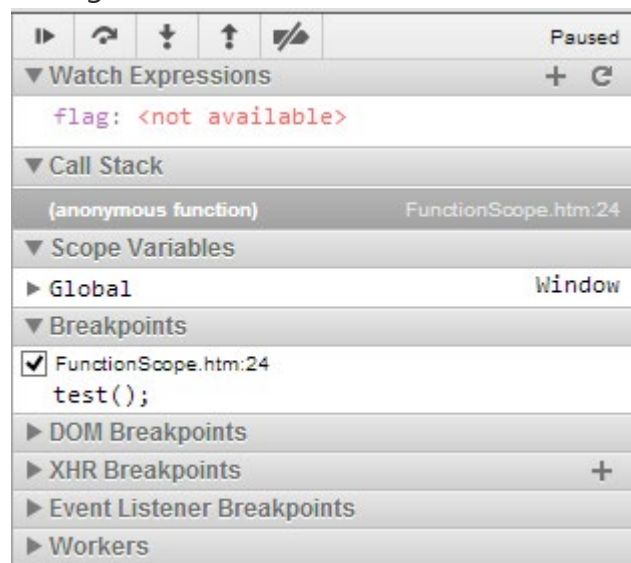
20. Refresh your page in the browser by pressing **F5**. When the code executes, it will stop at the line of code.

21. You will note there is a window to the right hand side of the screen. Locate the Watch Expressions section and click on the + sign and type in the variable name **flag**.

22. You have just created a watch. Watches are very useful debugging tools that allow us to check the value of variables to see if they behave as we expect. You'll see the benefit of this very soon. At this point, you should see it tells you flag is unavailable (early versions of Chrome may tell you there is an error but don't worry about that).

23. By pressing F11, you can move onto the next line of code in the stack. It will jump you into the test function. If you look to the right of the screen, you will see a **Call Stack** pane that shows the call stack. This

is a useful tool – the call stack is the set of instructions that are being carried out by



the program at this time. The Global object will remain alive for as long as the window is open. The functions disappear off the stack as soon as they are finished. As you press F11 (do not do that just yet), until you make it out of the program, you will see each function disappear off the stack.

24. This call stack shows that **test** was called by (anonymous function), which is Chrome's name for the global scope.
25. With the tools explained, let us start investigating. Hit F11 to move onto the next line of code. If you note what happens as you pass the **flag = true** line of code, flag will become available and have a value of true. This variable is accidentally global. It does not exist until **test()** is called, at which time it builds a new variable. This is dangerously bad practice in JavaScript. Follow through the code by pressing F11 until you leave the call stack. Once this is done, compare what happened with the assumptions you made in step 12.
26. Comment out the code in the first script block and uncomment the functions in script block 2 – explicit global scope. Note what is different. Jot down what you think will happen with two **console.log** lines to the flag variables and rerun the code with a breakpoint at the **test()** function call once again.
27. Repeat this process for blocks three and four in turn. If the results are not as expected and you do not know why, please ask your instructor for help.

## Part 3 – object literals

Object literals are one of the most powerful and useful functionalities in JavaScript. With them, we can create semantic arrays, whereby instead of using an integer index like the array you created in Part 1 of this exercise, you use a string as the key (the integer was the key in the first part of the exercise – `array[0]` is unique in the whole of the array).

28. Time to explore the difference between the two structures and then combine them together. Open **ObjectLiterals.htm** from the **Ex13 Starter Folder**.
29. Enter the following code to create an array about student data:

```
let objStudent = new Array();  
  
objStudent[0] = "Harry Potter";  
objStudent[1] = "4 Privet Drive";  
objStudent[2] = "hp@hogwarts.co.uk";
```

30. Now we have a simple indexed array. Eventually, we are going to want to hold more than one student's information. At that point, we would have an array of arrays – real programming requires a lot of these. If we were to rely on indexed integers and wanted to get the email of Harry, if he was the third entry in the **potionsClass** array, we would need to code `potionsClass[3][2]`. We can just about figure that out because we just created the `objStudent`. Imagine trying to work with that data structure if you did not know about the `objStudent` structure!
31. Right now, we'll leave the array as it is and read from it using the `for in` looping structure. The `for in` can be much better than the `for` loop, if you are reading from an array. The **in** keyword checks for the presence of a key in an array before it accesses it, so we can reduce hits to missing or sparse array structures.
32. Enter the following code and test the page in Chrome

```
for (let key in objStudent) {  
    console.log(key + " : " + objStudent[key]);  
}
```

33. This should give you every array entry and its value. Even if we switched the last index from 2 to 20, it will only read in members of the array that exist, i.e. not undefined. Note that the key refers to the numeric indexer value.

34. However, we are still left with a non-semantic data structure. So replace, in order, the numeric values with:

Numeric Indexer	String Indexer
0	name
1	address
2	email

35. Now we could access the array by using the syntax `objStudent["name"]`. Clearly, here we are asking for the name part of the student object. Feel free to try this, if you would like.
36. There is no need to change anything in the *for in* loop. The key is now a string instead of a number but behaves in the same way. Try your code and it should run exactly as before.
37. One of the other really useful functionalities is the ability to extend objects with new key values at any point, this is known as an **expando property**. Add the following code after the email declaration:

```
objStudent.schoolHouse = "Gryffindor"; //expando property
```

38. Try the loop again and you will now get the school house as well.
39. By the approach we have above, we have instantiated **objStudent** as an array. So far, this has been done to show the similarity between arrays and objects. We will not be taking advantage of any of the array types behaviour, so will change it to be a simple object. Change the **objStudent** declaration to **Object**. Test your code and, again, there will be no changes.
40. Next, create an array called **theClass** and assign **objStudent** to it as element **0**. (Hint: this is as simple as it seems).
41. To read from the array, we can access the **row** then **property** of the array we want. To test this, enter the following code:

```
console.log(theClass[0].schoolHouse);
```

42. We have defeated our nightmare of the two dimensional index array – **\*phew!\***
43. In Part 1 of the exercise, we agreed that writing less code is better for everyone. We previously used the short hand square brackets for array declaration. You could use `{ }` for object literal notational form. If you want to change the code **new object()** to **{ }** test your code once again, everything will work perfectly.

44. We will add a new student to the class array and use this to initialise an object as a single line declaration. Enter the following code (*N.B you can place this on one line if you like – the line breaks are for human readability in the manual*):

```
theClass[1] = {  
  name: "Draco Malfoy",  
  address: "Malfoy Manor",  
  email: "dm@hogwarts.ac.uk",  
  schoolHouse: "Slytherin"  
};
```

45. Change the call you made in step 41 to check for the new row in the array, it should report the value **Slytherin**.
46. Time for the final challenge in the exercise. We want to iterate over **theClass** and display every property of every object in the array. Start by adding a **for** loop that runs across the entire length of **theClass** array. Ensure the iterator is called **i**.
47. Inside the **for** loop, we will use a **for in** loop to test these values. Enter the following code:

```
for(let k in theClass[i]){  
  console.log(k + " :: " + theClass[i][k]);  
}
```

48. In the above code, **theClass[i]** gives us the current indexed entry of the array. The **for in** loop uses the variable **k** to ask for every property of the object inside that **theClass** record. It will display each property of an object before moving on to the next row.
49. Test the code in your browser to be sure it is working then you are done.