JS Basics II

Contents:

Functions II

Functions are small segments of code that perform a task. We use functions when we have some small piece of logic or functionality that we need to use a several times, and as a way to make our code clearer and more organised by breaking it into smaller segments.

It helps us to conform to one of the tenets of coding, DRY: Do not Repeat Yourself. We can write the code we need once, and then reference that code by name where it's required.

We might have a menu that we need to present to the user:

```
console.log("Hi, welcome to the app!")
console.log("Please choose from the following options:")
```

This is just an instance of something that might be repeated several times as a user moves through the app. Of course we might connect this more to the DOM, but for now it's just an example to get a sense of things.

If we wanted to use that menu several times we could repeat those lines where we need them:

```
console.log("Hi, welcome to the app!")
console.log("Please choose from the following options:")

// ...

console.log("Hi, welcome to the app!")
console.log("Please choose from the following options:")

// ...

console.log("Hi, welcome to the app!")
console.log("Please choose from the following options:")
```

But this is inefficient. We are repeating code time and again, and it's messy, prone to errors, and hard to maintain.

Defining a function

Instead, we can use a function to wrap up the code. Eventually you will see a few ways to define functions, but we will start with one way, and later show the alternatives.

In this case we are using the `function` keyword, then the name of the function, followed by the parentheses. The curly brackets hold the function code.

```
function welcome(){
    console.log("Hi, welcome to the app!")
    console.log("Please choose from the following options:")
}
```

Here we are defining a function called `welcome`. The parentheses `()` are the give away that this is a function, and these can be very useful to look for when you are trying to decypher code (although it's not failsafe because functions can exist without these curved brackets, but if you do see these parentheses, you are dealing with a function).

Because this is the function *definition*, this code is, in a sense, inert. It doesn't run now, but it is set up to run. This is an important distinction, and can take a little getting used to.

Calling a function

As a result, if we run the code above, nothing happens. The function is defined, but not used.

To execute this function we need a function *call*.

```
function welcome(){
    console.log("Hi, welcome to the app!")
    console.log("Please choose from the following options:")
}
// ...
welcome() // first call of function 'welcome'
// ...
welcome() // second call of function 'welcome'
// ...
```

Here we have defined the function, and then called it twice. In this rather basic scenario we would see the `console.log`s printing out those two lines, twice. Please run the code if you need to confirm this and get the idea.

Function arguments (inputs)

While it is great to be able to modularise your code with functions, most of the time we create a function, we have in mind a job for that function to do. Sometimes we will want that function to perform the same task, but with different inputs.

Our first function (`welcome()`) didn't take any arugments, because they weren't important for its execution. But in other functions, the inputs are vital, and we need a way to accomodate them.

We can do that through the arguments, which are the comma separated values that come inbetween the parentheses. They can be thought of as the inputs into the function. Here's an example:

```
function addThemUp(arg1, arg2) { // function definition
    let total = arg1 + arg2
    return total
}
```

Note that JS is not strict about the number of arguments accepted when calling a function. That is to say that functions can be called with fewer arguments or more arguments than the function as defined, and no error will result directly from this descrepancy. An error *may* result within the function (but not necessarily), but the function will attempt to execute regardless.

That is, we could call the function in these ways:

```
addThemUp(3)
addThemUp(67, 4, 11, 23)
```

And the function will run, but it may not run in the way we hope. You can investigate what the function returns in these cases, and we will talk more about this in the future.

`return` (the output)

We can see here in `addThemUp` that there is also the keyword `return`. *Every* function returns a value whenever it is called. This return value can be thought of as the output of the function.

Take careful note that 'returning' ends the execution of the function, meaning that any code that appears after `return` will not execute.

As before, we *call* the function (run the function) by invoking its name, and passing in the arguments that we require:

```
addThemUp(4, 5) // function call
```

We might need the value that is 'returned' from the function (the output), and if so then we need to store it somewhere. We can use a variable:

```
let totalAdded = addThemUp(4, 5)
```

Now our variable `totalAdded` will contain the 'result' of our `addThemUp` function, when called with the arguments `4`, and `5`. We can now use this value elsewhere in our code.

This is how you build up your programs. You find little tasks that need to be done, and you write functions to deal with these problems, and slowly build up the code in little pieces (as much as possible, and within reason). We will be talking more about this composition as the course goes on.

Summary so far

Again, we see here:

- Arguments are like the *inputs* into the function.
- The return value is the data that is output from the function.
- If we need to store the output (return value) of the function, we can assign it to a variable.

Also:

- Remember that when we have something to the right of an `=`, that this part is evaluated, and then *assigned* to the variable. So when you have a function call on the right, that function will be run, and the return value is then stored in the variable that you are declaring.
- Functions are a very important part of JS, so getting comfortable with the basics will be very important.

Hang on..

Some of you may have noticed that we didn't return anything from the menu printing function we used at the start, but then I also said that *every* function returns a value. So what's the deal?

In that case, the return value is `undefined`, which is a special type of value. We don't need to get into the weeds here though. It still returns a value, and that value is `undefined` if we don't explicitly specify what to return.

Also worth noting here is that the first function we created (`welcome()`) is one where the return value isn't important - the job it does is to print some text out, and then finish. We don't care about the output from this function (even though it will still return `undefined`).

In the example with `addThemUp`, we definitely do want to get hold of the return value (presumably - that's why we made it), and so we specify the return value, and capture it in a variable.

One last thing on `return`..

One thing I have also found to be a bit tricky for beginners, is the difference between `console.log` and the return value. Sometimes it can feel like the function is doing something important when we see some output as a result of `console.log`. And `console.log` is a very useful function, and I often use it to help with debugging, to see what value a variable has at a point in my code. However, it is important to keep these concepts separate. `console.log` prints to standard output (by default), and return passes a value out of the function, and back into the place in the execution of the code where the function was called.

Some example code:

```
function multiplier(num1, num2) { // function definition
    let result = num1 * num2
    return result
}
// console.log(multiplier) // You can uncomment this line if you would
like to see what is logged to screen

let resultOfMultiplier1 = multiplier(6, 7) // function call
console.log(resultOfMultiplier1)

let resultOfMultiplier2 = multiplier(1, 4) // function call
console.log(resultOfMultiplier2)

let resultOfMultiplier3 = multiplier(100, 300) // function call
console.log(resultOfMultiplier3)

function printSomething(str) { // function definition
    console.log(`printSomething ran and had argument ${str}`)
}

printSomething("Banana") // function call
printSomething("Orange") // function call
printSomething() // function call
```

We see here how we are defining the functions once, and then can use them several times. We see here also functions with explicit return values, and functions without a return. Run the code if you want to play around with it.

Edge cases

One thing that I would like to flag up early to get the concept out there is that of 'edge cases'. It's a concept that people should be trying to think about when designing code, and usually in the context of writing your functions. There is a lot to say on this topic, but here I want to just flag it up with an example, and we can talk more about it as we go through the course.

Let's write one more simple function that divides one number by another:

```
function divider(numerator, denominator) {
    let divided = numerator / denominator
    return divided
}
```

Let's think through a couple of the problems that could arise even in a function this simple. Have a think about what these problems might be, before reading on.

I can think of at least six issues that need to be dealt with before this function can be reliable:

- There are differences between doing integer and floating point division that we need to consider.

- There are issues when dividing by `0`.
- There will be issues around significant digits, and computer arithmetic.
- There are issues with very large or very small numbers.
- The arguments sent may not be numbers.
- The function may be called with fewer than two arguments.

There are likely to be more (perhaps many more), but these are a start.

All of the above problems could have nasty side effects, and they fall under the category of edge cases - cases where the inputs fall outside of the typical range, usually at the extremes. If we don't think through all the ways that inputs into this function could produce 'unusual' or somewhat unexpected behaviour, and that this might mean that a function that we trust to do a job and is then used several times throughout the project is actually introducing some strange behaviour.

Anyone interested can look into these particular issues with this function, and have a crack at making it more robust. It is worth noting that it's very important to very clearly understand what it is that you need this function to be doing, because this guides how you handle these edge cases. For exmaple, you might simply reject certain 'bad' input, and put the onus back on the user. Or you may return a value, but accompany some values with warnings. And so on. All these are important decisions to be made, and there is no right or wroing answer, because it depends on the situation.

The main point here is that we can see that even within a function that appears so simple, there are somewhat hidden complexities. Keeping an eye out for these snags is something that you will get used to as you progress through the course. We can run through this specific example, with some solutions in class.

A note on naming

Naming variables is hard. It is said that there are only two hard things in Computer Science: cache invalidation and naming things (attributed to Phil Karlton). I know virtually nothing about cache invalidation, but I have had to name very many variables and functions and other things as a dev, so I can vouch for the second element (although would argue there are many hard things in CS).

For now, err of the side of making your names too long, and try to be descriptive. Another saying I have heard is that you should write your code like your coding partner is a murderous psychopath. You want other people to be able to read your code as easily as possible, and naming things well is a big part of that. Also remember that while today everything you are writing might make sense to you, if you were to leave it aside and come back to it, you too could be the victim of your own poor choices. You may find yourself having to decode your own code.

This is similar to my request that you err on the side of clarity and simplicity more generally when writing your code. Often times code that seems 'clever' and crams a lot into a short space is actually just painful when reviewing the code, working in a team, or debugging.

Instead of the function above, I could have written:

```
function d(a,b) {
    return a/b
}
```

```
const t = d(6,7)
console.log(t)
```

It still works, so what's the problem? Well, maybe in this example we can see that what this code does, and can cope. But think ahead. Imagine you have 1500 lines of code like this - at some point you will be tearing your hair out trying to make sense of it. Or your team mate, or worse, your boss.

Here is an example of working code that no one wants to deal with:

```
(function(e){"use strict";e.fn.counterUp=function(t){var
n=e.extend({time:400,delay:10},t);return this.each(function(){var
t=e(this),r=n,i=function(){var e=[],n=r.time/r.delay,i=t.text(),s=/[0-9]+,
[0-9]+/.test(i);i=i.replace(/,/g,"");var o=/^[0-9]+$/.test(i),u=/^[0-9]+\.
[0-9]+$/.test(i),a=u?(i.split(".")[1]||[]).length:0;for(var f=n;f>=1;f--)
{var l=parseInt(i/n*f);u&&
(l=parseFloat(i/n*f).toFixed(a));if(s)while(/(\d+)
(\d{3})/.test(l.toString()))l=l.toString().replace(/(\d+)
(\d{3})/,"$1,$2");e.unshift(l)}t.data("counterup-nums",e);t.text("0");var
c=function(){t.text(t.data("counterup-
nums").shift());if(t.data("counterup-
nums").length)setTimeout(t.data("counterup-func"),r.delay);else{delete
t.data("counterup-nums");t.data("counterup-nums",null);t.data("counterup-
func",null)}};t.data("counterup-func",c);setTimeout(t.data("counterup-
func"),r.delay)};t.waypoint(i,{offset:"100%",triggerOnce:!0})})})}})
(jQuery);
```

Get in the habit of coding for others because it's good practice, but it will also help you when you need to debug you code, or return to your code in the future.

Next: Functions and the DOM