Array Methods – forEach, Map, Filter, Reduce why and when to use

A little Background

Often in the course of creation of a web application we require to use and manipulate large blocks of data. In the case of a web application this data could be read from a database and contain for example, customer addresses or sales data for regions of business. The data lends itself to the structure of an array and hence the need to iterate over such an array and perform transformations or calculations is paramount in software development. Traditionally we would primarily use one of two approaches, that of the for loop and that of the while loop. These are supported across many languages but are fundamental to Javascript which is one of the leading web development languages and the one we shall be using for our discussion henceforth.

The general structure of both **for** and **while** loops for iterating an array and performing some function is shown below:-

For loop

```
for( i = value ; Condition on i ; increment/decrement i )
  {
       function( data array );
  }
```

The basic structure is a loop centred around an indexing count i. This index can have an initial value then the loop increments or decrements that index whilst comparing to a condition. This index condition indicates when the loop will terminate, i.e. index<100 or index>=5 . The function within the loop executes for each loop iteration and performs the data manipulation required. A prerequisite of this form of loop is a known termination via the condition in order to avoid an infinite loop situation. The code though readable can become complex and requires extra variables for any required indexes, of which their can be more than one. In terms of execution speed the for loop can be very efficient in performance.

While loop

```
while( test of condition)
  {
     function(array)
     set/clear(condition)
  }
```

The **while** loop operates in a similar manner to the **for** loop but rather than using an index count to iterate the loop it uses a binary condition. This condition can be set true/false within the loop body and can use an index count or boolean equation to generate it. The function within the loop performs in the same way it does in the **for** loop. This loop method is a little easier to read than the **for** loop.

The Alternatives: forEach, Map, Filter, Reduce

Javascript offers the alternatives mentioned in the title as methods to use with array class objects. Each of the methods has unique features that are optimal for specific uses but together they can replace code often written as **for** or **while** loops and can help improve code readability and potentially reliability too. I shall give an overview of each method together with typical uses and any pro's and con's. I shall conclude by summarising the characteristics of these methods in tabular form.

ForEach

The forEach method iterates over the contents of an array element by element. The general form of the method is shown below:-

```
myArray.forEach( myCallback( element, [index],[array]));
```

The only required parameter of the callback function is element, index and array are optional. The callback function executes for each element in the array associated with the method invocation. The method does not return anything so assignment to another variable is not required. The method will modify the contents of the calling array if required as opposed to copying modified elements to a new array. This may not be the desired outcome if array data is required to be immutable. The callback function can also be represented by an arrow function. You would use forEach for example if we were to increase all of our web shop prices by 5% then we could access our database of retail prices, create an array webShopPrices and use a forEach as in pseudocode below:-

```
webShopPrices.forEach( (currentPrice) =>( currentPrice*1.05));
```

The forEach approach is much neater than the equivalent **for** loop but has the disadvantage of modifying the original data as opposed to returning a modified copy. This can sometimes not be the required result hence the next method to be discussed.

Map

If one should require to maintain the integrity of the original array data then the map method is an appropriate replacement for our **for** and **while** loops. The general form of the method is shown below:-

```
returnedArray = myArray.map( callback( element,[index],[array] ));
```

The structure of the callback function parameters are the same as that of the forEach method(including use of arrow functions). The main difference with map is in the manner in which every iteration will generate a returned result that is stored in a new array. After the map method has iterated through each element of the calling array the generated array(original array elements operated on by map callback function) will be returned to any assigned variable. The interface of the method has several forms, external callback function and inline callback function with several argument options. Use of map would be appropriate where data should be modified but the original data remain intact. An example of such could be where a user could change values universally on a web page but without changing the source data. This could be say a web store showing items that a user could change the currency type i.e GBP, USD, EUR etc and see the revised prices. This would only change the displayed data rather than the initial data returned from the central database.

Map can again replace our basic **for** and **while** loops and has the advantage over forEach of maintaining the integrity of the original source data.

Filter

The filter array method is the first of two methods that differ slightly from our previous methods. Filter, and Reduce, have a more limited scope in terms of functionality but offer an efficient way to perform commonly required functions and reduce the quantity of code. The Filter method has a interface structure in common with the previous two methods. The method has a callback function with the required element parameter and two optional index and array parameters, the general use is shown below:-

```
filterResult = myArray.filter( callback( element,[index],[array]),thisarg);
```

The parameter thisarg is used to reference other objects and for the purpose of this blog post it can be ignored. The callback functions purpose with the filter method is to test the iterated elements of the calling array against a logical condition. A simple case would be if we required to filter a numeric array to create a sub-array containing all elements that are of even value. The filter method, as with map, will not corrupt the original calling array but will return a new array with an equal or lower number of elements. In the real world our arrays are more likely to contain complex data objects that may represent perhaps customer details or utility account data. This is where Filter provides a very functional and efficient way to manipulate the data. If we had say data for a class of students we may have an array of objects such as a name and age as shown below:-

```
{ name:fred smith, age: 45 }  { name: susan johnson, age: 37 }  { name: ian cook, age:25 }
```

We could, for example, use filter to create a sub-array of names and ages for all of our students above a specific age. The structure of such a basic task would appear as shown below:-

```
resultsArray = myArray.filter( (item) => item.age>30 );
```

The callback here is an inline arrow function for brevity. This would create a sub-array returned in resultsArray and for our example would contain only 2 objects those of fred smith and susan johnson. It can be seen that for a very small number of code lines we can implement very complex filtering tasks that would be quite extensive if coding from scratch. Though it may seem simpler in function the filter method has many uses within web applications and data sorting.

Reduce

The final method to consider is the Reduce method which as its name suggest reduces an array to a single valued output. As with map and filter the calling data array remains unaltered and the result of the method is a return value. The structure of the method call is shown below:-

```
resultValue = myArray( callback( accumulator, currentval ), initialVal );
```

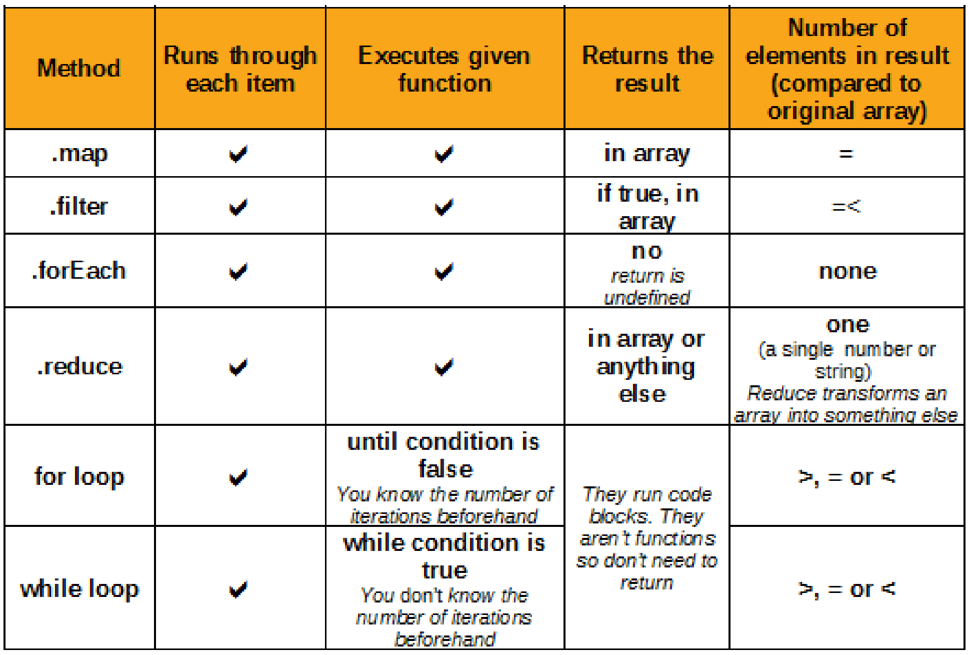
The parameter initialVal is optional and represents a starting quantity, perhaps last years profit ready for adding this years income.The method implements an accumulation of modified array elements. The method iterates over the calling arrays elements applying the callback to each and accumulating the rolling result. If an initial value is given the iteration operates from the second element, otherwise its the first. The callback can be an arrow function as well as an inline function. If we consider a typical example it could be to account the total of prices in a web cart array and factor in the VAT in the summation. This would look as below:-

```
totalCartPrice = myArray.reduce( ( total, item) => total+(item*1.2) );
```

This is a very simple example and there are more complex ways in which to use reduce but for many cases it allows a fast, clean way to iterate an arrays content to create a totalise result. The reduce method is very different in its output to the previous three methods and has a well defined purpose. It is another example of a method that can lead to cleaner more readable code.

<u>Conclusion</u>

In conclusion it can be seen that we have four method calls for array objects that allow us to replace much use of **for** or **while** loops. Each method has its own characteristic advantages and disadvantages. It is for the web programmer to decide the most suitable approach and **for** and **while** loops should not be disregarded if profiling shows a performance benefit. The table below summarises the characteristics of each method:-



Method	Runs through each item	Executes given function	Returns the result	Number of elements in result (compared to original array)
.map	✓	✓	in array	=
.filter	✓	✓	if true, in array	=<
.forEach	✓	✓	no *return is undefined*	none
.reduce	✓	✓	in array or anything else	one (a single number or string) *Reduce transforms an array into something else*
for loop	✓	until condition is false *You know the number of iterations beforehand*	*They run code blocks. They aren't functions so don't need to return*	>, = or <
while loop	✓	while condition is true *You don't know the number of iterations beforehand*		>, = or <