Iterators

Module – Advanced JS

Dr. Andrew Errity

iadt
DUN LAOGHAIRE

Outline

- `forEach()`
- `every()`
- `some()`
- `map()`
- `filter()`
- `reduce()`

Intro to Iterators

- Looping over arrays is such a common task that new features have been added to JavaScript over time to simplify this task.
- These methods are called iterators.
- The methods discussed below were added in ES5 which was published in 2009.

forEach()

- Iterates over the elements of an array:

```
1  const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];
2
3  donuts.forEach(function(donutElement) {
4    console.log(donutElement);
5  });
6
7  donuts.forEach(function(donutElement, i, donutArray) {
8    console.log(`Donut option ${i + 1} is ${donutArray[i]}`);
9  });
```

```
chocolate
red velvet
custard
jam
lemon
Donut option 1 is chocolate
Donut option 2 is red velvet
Donut option 3 is custard
Donut option 4 is jam
Donut option 5 is lemon
```

forEach()

- Note how this is used:

```
arrayName.forEach( callback )
```

- The callback takes the following parameters:

```
function(element, index, originalArray)
```

- You don't have to supply all the parameters.
- The methods on the following slides use the same syntax.

Further reading: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

Activity

- Use `forEach()` to take the array

  ['Thinking in JS', 'JS Patterns', 'JS: The Good Parts', 'ES6 and Beyond']

- And print:

  I need to read ES6 and beyond

  I need to read Thinking in JS

  I need to read JS Patterns

  I need to read JS: The Good Parts

  I need to read ES6 and Beyond

For loops

- One reason to still use for loops, when you need to break out of the loop early, e.g.

```
1    const prices = [10, 20, 40, 150, 15];
2    for(let i = 0; i < prices.length; i++) {
3      if (prices[i] > 100) {
4        doSomething();
5        break;
6      }
7    }
```

Further reading: http://thejsguy.com/2016/07/30/javascript-for-loop-vs-array-foreach.html

every()

- Returns **`true`** if the callback returns **`true`** for every element.

```
1   const words = ['speciality', 'snappy', 'funny'];
2   const words2 = ['worry', 'happy', 'tired'];
3
4   words.every(function(string) {
5     return string[string.length - 1] === 'y';
6   }); // true
7
8   words2.every(function(string) {
9     return string[string.length - 1] === 'y';
10  }); // false
```

every()

- Refactored so the callback can be written once and reused:

```
const words = ['speciality', 'snappy', 'funny'];
const words2 = ['worry', 'happy', 'tired'];

// store the function in a variable
const endsInY = function(string) {
  return string[string.length - 1] === 'y';
};

// pass in the function and it will be used to
// perform the check on every element
words.every(endsInY); // true

words2.every(endsInY); // false
```

every()

- Previous code refactored to use an arrow function:

```
1   const words = ['speciality', 'snappy', 'funny'];
2   const words2 = ['worry', 'happy', 'tired'];
3
4   // previous function rewritten as arrow function
5   const endsInY = string => string[string.length - 1] === 'y';
6
7   // pass in the arrow function and it will be used to
8   // perform the check on every element
9   words.every(endsInY); // true
10
11  words2.every(endsInY); // false
```

Activity

- Use **`every()`** to check if all the elements in an array are divisible by 5.

```
1    const nums = [5, 10, 15, 30]; // true
2    const nums2 = [6, 10, 15, 30]; // false
```

some()

- Returns **`true`** if the callback returns **`true`** for at least one element.

```
1   const words = ['speciality', 'snappy', 'funny'];
2   const words2 = ['worry', 'happy', 'tired'];
3
4   // store the function in a variable
5   const endsInY = string => string[string.length - 1] === 'y';
6
7   // pass in the function and it will be used to
8   // perform the check on every element
9   words.some(endsInY); // true
10
11  words2.some(endsInY); // true
```

Activity

- Use `some()` to check if any of the elements in an array have more than 5 characters.

```
1    const names = ['Andrew', 'Mortimor', 'Alexandria']; // true
2    const names2 = ['Jo', 'Jill', 'Alex']; // false
```

map()

- Applies a callback to each element of the array and stores the results in a new output array.

```
1    const a = [1, 2, 3, 4, 5];
2
3    const b = a.map(function(value) { return value * value; });
4    console.log(b); // [ 1, 4, 9, 16, 25 ]
```

- Refactored to use an arrow function:

```
1    const a = [1, 2, 3, 4, 5];
2
3    const b = a.map(value => value * value);
4    console.log(b); // [ 1, 4, 9, 16, 25 ]
```

map()

- Another example:

```
1   const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];
2
3   const donuts2 = donuts.map(function(donutElement) {
4     return donutElement + ' donut';
5   });
6
7   console.log(donuts2);
8   /*[ 'chocolate donut',
9     'red velvet donut',
10    'custard donut',
11    'jam donut',
12    'lemon donut' ]
13     */
```

map()

- Previous example refactored to use arrow function:

```
1   const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];
2
3   const donuts2 = donuts.map(donutElement => donutElement + ' donut');
4
5   console.log(donuts2);
6   /* [ 'chocolate donut',
7     'red velvet donut',
8     'custard donut',
9     'jam donut',
10    'lemon donut' ]
11     */
```

Activity

- Use `map()` to take the array

  ['chocolate', 'red velvet', 'custard', 'jam', 'lemon']

- And produce an array containing

['Chocolate', 'Red velvet', 'Custard', 'Jam', 'Lemon']

filter()

- The output array contains only those input elements for which callback returns **`true`**.

```
1  const dictionary = ['flabbergasted', 'outrageous', 'crazy', 'absurd'];
2
3  const isLongWord = string => string.length > 6;
4
5  const longWords = dictionary.filter(isLongWord); // [ 'flabbergasted',
•  'outrageous' ]
```

Activity

- Use `filter()` to parse an array and create a version containing only the positive values in the array.

```
1    const posNeg = [1, -1, -2, 3, 5, -7];
2    // we want [1, 3, 5]
```

reduce()

- Applies a callback against an accumulator and each element in the array (from left to right) to reduce it to a single value.

```
1  const goalsScoredInGames = [1, 1, 0, 5, 3, 0, 1];
2
3  const totalGoalsScored = goalsScoredInGames.reduce(function(sum, value) {
4    return sum += value;
5  });
6
7  console.log(totalGoalsScored); // 11
```

reduce()

- Refactored to use an arrow function:

```
1  const goalsScoredInGames = [1, 1, 0, 5, 3, 0, 1];
2
3  const totalGoalsScored = goalsScoredInGames.reduce((sum, value) => sum += value);
4
5  console.log(totalGoalsScored); // 11
```

reduce()

- The syntax differs to the previous methods:

```
arrayName.reduce( callback [, intialValue )
```

- The initial value for the accumulator is optional. If not supplied it's set equal to the first element in the array.
- The callback takes the following parameters:

```
function(accumulator, element, index, originalArray)
```

Activity

- Use `reduce()` to iterate over this array and sum the length of all the elements that are longer than 6 characters.

  ['flabbergasted', 'outrageous', 'crazy', 'absurd']

- Output should be **23**.

MapReduce

- The map() and reduce() operations are the basis of a very popular technique for processing big data.
- They are often used in distributed computing environments, e.g. Hadoop.

Further reading: https://www.simplilearn.com/what-is-mapreduce-and-why-it-is-important-article
https://www.ibm.com/analytics/us/en/technology/hadoop/mapreduce/
https://www.edureka.co/blog/mapreduce-tutorial/

Activity

- Complete part 6 - Iterators:
  - https://www.codecademy.com/learn/introduction-to-javascript
- You may need to refer to the Mozilla JS docs
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array