Iterators

The `.reduce()` Method

The .reduce() method iterates through an array and returns a single value.
In the above code example, the .reduce() method will sum up all the elements of the array. It takes a callback function with two parameters (accumulator, currentValue) as arguments. On each iteration, accumulator is the value returned by the last iteration, and the currentValue is the current element.
Optionally, a second argument can be passed which acts as the initial value of the accumulator.

```
const arrayOfNumbers = [1, 2, 3, 4];

const sum =
arrayOfNumbers.reduce((accumulator,
currentValue) => {
  return accumulator + currentValue;
});

console.log(sum); // 10
```

The `.forEach()` Method

The .forEach() method executes a callback function on each of the elements in an array in order.
In the above example code, the callback function containing a console.log() method will be executed 5 times, once for each element.

```
const numbers = [28, 77, 45, 99, 27];

numbers.forEach(number => {
  console.log(number);
});
```

The `.filter()` Method

The .filter() method executes a callback function on each element in an array. The callback function for each of the elements must return either true or false . The returned array is a new array with any elements for which the callback function returns true .
In the above code example, the array filteredArray will contain all the elements of randomNumbers but 4 .

```
const randomNumbers = [4, 11, 42, 14, 39];
const filteredArray =
randomNumbers.filter(n => {
  return n > 5;
});
```

The `.map()` Method

The `.map()` method executes a callback function on each element in an array. It returns a new array made up of the return values from the callback function.
The original array does not get altered, and the returned array may contain different elements than the original array.
In the example code above, the `.map()` method is used to add `' joined the contest.'` string at the end of each element in the `finalParticipants` array.

```
const finalParticipants = ['Taylor',
'Donald', 'Don', 'Natasha', 'Bobby'];

// add string after each final participant
const announcements =
finalParticipants.map(member => {
  return member + ' joined the contest.';
})

console.log(announcements);
```

Functions Assigned to Variables

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

```
let plusFive = (number) => {
  return number + 5;
};
// f is assigned the value of plusFive
let f = plusFive;

plusFive(3); // 8
// Since f has a function value, it can be
invoked.
f(9); // 14
```

Callback Functions

In JavaScript, a callback function is a function that is passed into another function as an argument. This function can then be invoked during the execution of that higher order function (that it is an argument of).
Since, in JavaScript, functions are objects, functions can be passed as arguments.

```
const isEven = (n) => {
  return n % 2 == 0;
}

let printMsg = (evenFunc, num) => {
  const isNumEven = evenFunc(num);
  console.log(`The number ${num} is an
even number: ${isNumEven}.`)
}

// Pass in isEven as the callback function
printMsg(isEven, 4);
// Prints: The number 4 is an even number:
True.
```

Higher-Order Functions

In Javascript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.
A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

JavaScript Functions: First-Class Objects

JavaScript functions are first-class objects. Therefore:

- They have built-in properties and methods, such as the `name` property and the `.toString()` method.
- Properties and methods can be added to them.
- They can be passed as arguments and returned from other functions.
- They can be assigned to variables, array elements, and other objects.

```
//Assign a function to a variable
originalFunc
const originalFunc = (num) => { return num
+ 2 };

//Re-assign the function to a new variable
newFunc
const newFunc = originalFunc;

//Access the function's name property
newFunc.name; //'originalFunc'

//Return the function's body as a string
newFunc.toString(); //'(num) => { return
num + 2 }'

//Add our own isMathFunction property to
the function
newFunc.isMathFunction = true;

//Pass the function as an argument
const functionNameLength = (func) => {
return func.name.length };
functionNameLength(originalFunc); //12

//Return the function
const returnFunc = () => { return newFunc
};
returnFunc(); //[Function: originalFunc]
```