Topics I’m gonna cover in this post

1. Let and Const – Let is block scoped
2. Arrow functions -

The new syntax may be confusing a little bit. But I will try to explain the syntax.

There are two parts of the syntax.

1. var newOne = ()
2. => {}
   1. let NewOneWithParameters = (a, b) => {  
       console.log(a+b); // 30  
      }  
      NewOneWithParameters(10, 20);
3. Default parameters – default parameter can e set in ECMA6
   1. let Func = (a, b = 10) => {  
       return a + b;   
      }  
      Func(20); // 20 + 10 = 30
   2. let NotWorkingFunction = (a = 10, b) => {  
       return a + b;  
      }  
      NotWorkingFunction(20); // NAN. Not gonna work.
4. for of loop –
   1. for..of is very similar to for..in with slight modification.
   2. for..of iterates through list of elements (i.e) like Array and returns the elements (not their index) one by one.
      1. let arr = [2,3,4,1];  
         for (let value of arr) {  
          console.log(value);  
         }
      2. Output:  
         2  
         3  
         4  
         1

let string = "Javascript";  
for (let char of string) {  
 console.log(char);  
}

Output:  
J  
a  
v  
a  
s  
c  
r  
i  
p  
t

1. Spread attributes –
   * 1. Example without spread attributes:
     2. let SumElements = (arr) => {  
         console.log(arr); // [10, 20, 40, 60, 90]
     3. let sum = 0;  
         for (let element of arr) {  
         sum += element;  
         }  
         console.log(sum); // 220.   
        }
     4. SumElements([10, 20, 40, 60, 90]);
     5. Above example is straightforward. We are declaring a function to accept array as parameter and returning its sum. Its simple.
     6. Now consider the same example with spread attributes
     7. let SumElements = (...arr) => {  
         console.log(arr); // [10, 20, 40, 60, 90]
     8. let sum = 0;  
         for (let element of arr) {  
         sum += element;  
         }  
         console.log(sum); // 220.   
        }
     9. SumElements(10, 20, 40, 60, 90); // Note we are not passing array here. Instead we are passing the elements as arguments.
     10. In the above example, the spread attribute converts the list of elements (i.e) the parameters to an array.
         1. Another Example:
         2. Math.max(10, 20, 60, 100, 50, 200); // returns 200.
         3. let arr = [10, 20, 60];  
            Math.max(...arr); // 60
2. Maps - Map holds key-value pairs. It’s similar to an array but we can define our own index. And indexes are unique in maps.
   1. var NewMap = new Map();  
      NewMap.set('name', 'John');   
      NewMap.set('id', 2345796);  
      NewMap.set('interest', ['js', 'ruby', 'python']);
   2. NewMap.get('name'); // John  
      NewMap.get('id'); // 2345796  
      NewMap.get('interest'); // ['js', 'ruby', 'python']

var map = new Map();  
map.set('name', 'John');  
map.set('id', 10);

map.size; // 2. Returns the size of the map.

map.keys(); // outputs only the keys.   
map.values(); // outputs only the values.

for (let key of map.keys()) {  
 console.log(key);  
}

Output:  
name  
id

var map = new Map();

for (let element of map) { // for (let [key,value] of map)  
 console.log(element);  
}

Output:  
['name', 'John']  
['id', 10]

The above example is self explanatory. The for..of loop outputs the key-value pair in array.

Sets - var sets = new Set();  
sets.add('a');  
sets.add('b');  
sets.add('a'); // We are adding duplicate value.

for (let element of sets) {  
 console.log(element);  
}

Output:  
a  
b

var sets = New Set([1,5,6,8,9]);

sets.size; // returns 5. Size of the size.  
sets.has(1); // returns true.   
sets.has(10); // returns false.

Static methods –

class Example {  
 static Callme() {  
 console.log("Static method");  
 }  
}  
Example.Callme();

Output:  
Static method

1. Getters and Setters –
2. Example with getters and setters
   * 1. class People {
     2. constructor(name) {  
         this.name = name;  
         }  
         get Name() {  
         return this.name;  
         }  
         set Name(name) {  
         this.name = name;  
         }  
        }
     3. let person = new People("Jon Snow");  
        console.log(person.Name);  
        person.Name = "Dany";  
        console.log(person.Name);

<https://stackoverflow.com/questions/11488014/asynchronous-process-inside-a-javascript-for-loop>

**Use .forEach() to iterate since it creates its own function closure**

someArray.forEach(function(item, i) {

asynchronousProcess(function(item) {

console.log(i);

});

});

**Create Your Own Function Closure Using an IIFE**

var j = 10;

for (var i = 0; i < j; i++) {

(function(cntr) {

// here the value of i was passed into as the argument cntr

// and will be captured in this function closure so each

// iteration of the loop can have it's own value

asynchronousProcess(function() {

console.log(cntr);

});

})(i);

}

**Create or Modify External Function and Pass it the Variable**

If you can modify the asynchronousProcess() function, then you could just pass the value in there and have the asynchronousProcess() function the cntr back to the callback like this:

var j = 10;

for (var i = 0; i < j; i++) {

asynchronousProcess(i, function(cntr) {

console.log(cntr);

});

}

**Use ES6 let**

If you have a Javascript execution environment that fully supports ES6, you can use let in your for loop like this:

const j = 10;

for (let i = 0; i < j; i++) {

asynchronousProcess(function() {

console.log(i);

});

}

let declared in a for loop declaration like this will create a unique value of i for each invocation of the loop (which is what you want).

**Serializing with promises and async/await**

If your async function returns a promise, and you want to serialize your async operations to run one after another instead of in parallel and you're running in a modern environment that supports asyncand await, then you have more options.

async function someFunction() {

const j = 10;

for (let i = 0; i < j; i++) {

// wait for the promise to resolve before advancing the for loop

await asynchronousProcess();

console.log(i);

}

}

This will make sure that only one call to asynchronousProcess() is in flight at a time and the forloop won't even advance until each one is done. This is different than the previous schemes that all ran your asynchronous operations in parallel so it depends entirely upon which design you want. Note: await works with a promise so your function has to return a promise that is resolved/rejected when the asynchronous operation is complete. Also, note that in order to use await, the containing function must be declared async.