



Advance Java Script

PRASHANT TOMER

Map Method

- The `map()` method is an iterative method. It calls a provided `callbackFn` function once for each element in an array and constructs a new array from the results.
- `callbackFn` is invoked only for array indexes which have assigned values. It is not invoked for empty slots in sparse arrays.
- Syntax:
 - `map(callbackFn)`
 - `map(callbackFn, thisArg)`

```
// JavaScript source code
const arr = [2, 5, 6, 7, 12, 9];
const mapArray = arr.map((x) => x * 2);
console.log(mapArray);
```

Developer PowerShell

```
PS C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1> cd '..\Advance JS'
PS C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\Advance JS> node eMap.js
[ 4, 10, 12, 14, 24, 18 ]
PS C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\Advance JS>
```

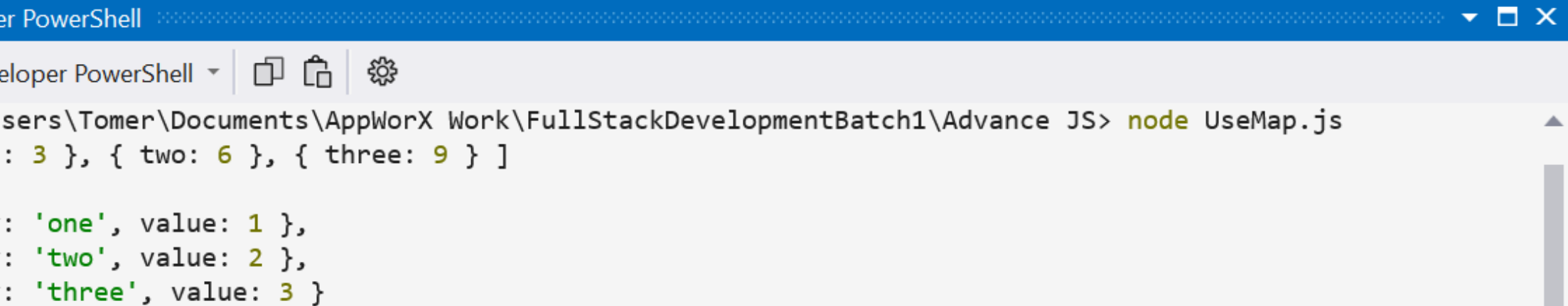
```
const numbers = [1, 4, 9];  
const roots = numbers.map((num) => Math.sqrt(num));  
  
// roots is now      [1, 2, 3]  
// numbers is still [1, 4, 9]
```

Using map to reformat objects in an

```
const kvArray = [
  { key: 'one', value: 1 },
  { key: 'two', value: 2 },
  { key: 'three', value: 3 },
];

const mappedArray = kvArray.map(({ key, value }) => ({ [key]: value*3 }));

console.log(mappedArray);
console.log(kvArray);
```



```
PowerShell
Developer PowerShell
C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\Advance JS> node UseMap.js
[ { one: 3 }, { two: 6 }, { three: 9 } ]

[ { key: 'one', value: 1 },
  { key: 'two', value: 2 },
  { key: 'three', value: 3 } ]
```

Using parseInt() with map()

```
const arr = ["3", "5", "30", "40"];
let intArray = arr.map((str) => parseInt(str));
console.log(intArray);

console.log(["1", "2", "3"].map((str) => parseInt(str))); // [1, 2, 3]

console.log(["10", "20", "30"].map(Number));
```

Developer PowerShell

+ Developer PowerShell

PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\Advance JS> node UseMap.js

[3, 5, 30, 40]

[1, 2, 3]

[10, 20, 30]

PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\Advance JS>

Important Array Methods

- The **filter()** method creates a shallow copy of a portion of a given array, filtered down to just the elements from the given array that pass the test implemented by the provided function.

Syntax:

- `filter(callbackFn)`
- `filter(callbackFn, thisArg)`

callbackFn

A function to execute for each element in the array. It should return a truthy value to keep the element in the resulting array, and a falsy value otherwise. The function is called with the following arguments:

element

The current element being processed in the array.

index

The index of the current element being processed in the array.

array

The array filter() was called upon.

thisArg Optional

A value to use as this when executing callbackFn


```
const words = ['I', 'am', 'working', 'as',  
              'a', 'freelancer', 'corporate', 'trainer'];
```

```
const result = words.filter(word => word.length > 7);
```

```
console.log(result);
```

Developer PowerShell

+ Developer PowerShell

PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\Advance JS> node UseMap.js

['freelancer', 'corporate']

PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\Advance JS>

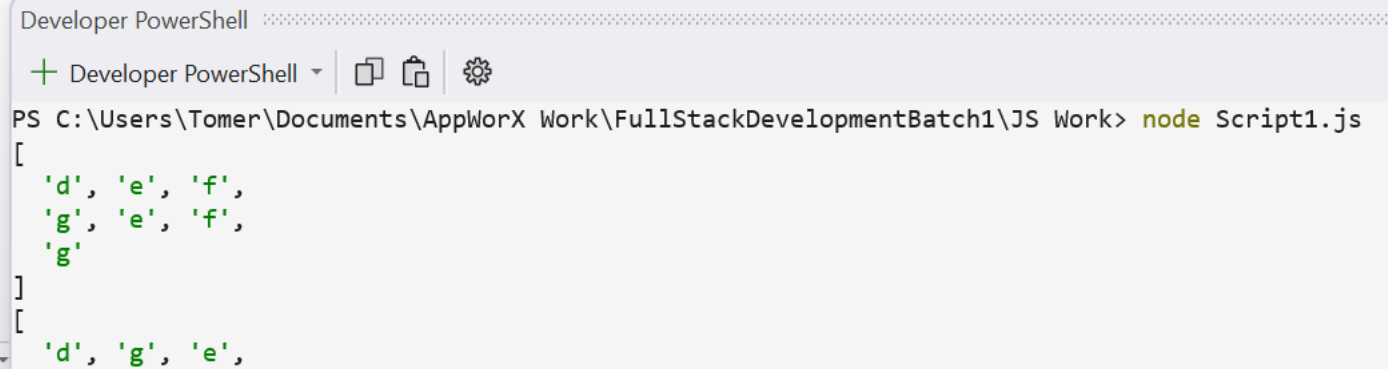
copyWithin()

The copyWithin() method shallow copies part of an array to another location in the same array and returns it without modifying its length

```
const array1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g'];

// Copy to index 0 the element at index 3, till 6-1 index
console.log(array1.copyWithin(0, 3, 6));

// Copy to index 1 all elements from index 3 to the end
console.log(array1.copyWithin(1, 3));
```



Developer PowerShell

+ Developer PowerShell | [] [] []

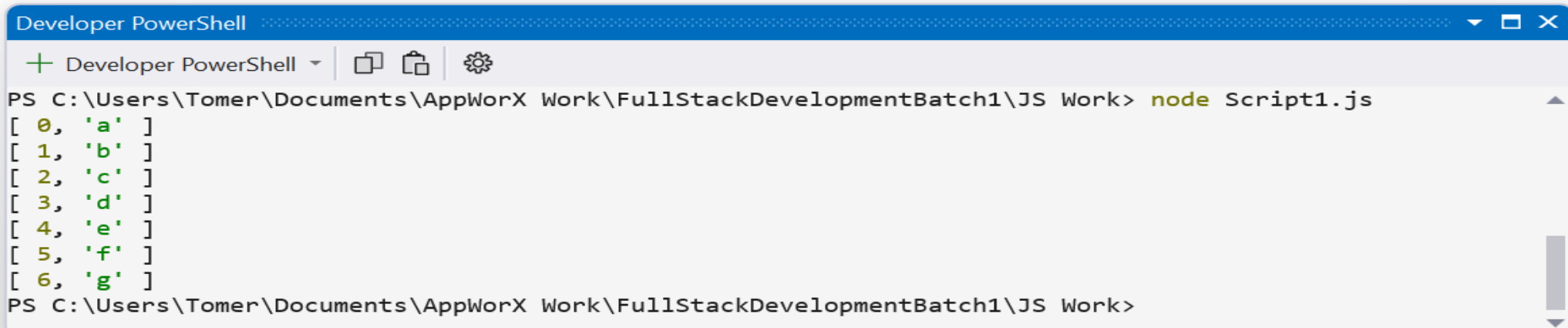
PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work> node Script1.js

```
[
  'd', 'e', 'f',
  'g', 'e', 'f',
  'g'
]
[
  'd', 'g', 'e',
```

The `entries()` method returns a new array iterator object that contains the key/value pairs for each index in the array.

```
const array1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g'];

let iterable = array1.entries();
let nextVal;
for (let i = 0; i < array1.length; i++) {
  console.log(iterable.next().value);
}
```



The screenshot shows a Windows PowerShell window titled "Developer PowerShell". The command prompt shows the execution of `node Script1.js`. The output displays seven array elements, each as an array containing an index and a character: `[0, 'a']`, `[1, 'b']`, `[2, 'c']`, `[3, 'd']`, `[4, 'e']`, `[5, 'f']`, and `[6, 'g']`. The terminal window has a blue title bar and standard Windows window controls.

```
Developer PowerShell
+ Developer PowerShell | [Icons] | [Settings]
PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work> node Script1.js
[ 0, 'a' ]
[ 1, 'b' ]
[ 2, 'c' ]
[ 3, 'd' ]
[ 4, 'e' ]
[ 5, 'f' ]
[ 6, 'g' ]
PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work>
```

Array.prototype.flat()

The flat() method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.


```
// JavaScript source code---- single line comment
```

```
const arr1 = [0, 1, 2, [3, 4]];
```

```
console.log(arr1.flat());
```

```
const arr2 = [0, 1, 2, [[[3, 4]]]];
```

```
console.log(arr2.flat(2)); // flat upto 2 layer of array
```



```
Developer PowerShell
+ Developer PowerShell
PS C:\Users\Tomer\Documents\AppWork\FullStackDevelopmentBatch1\JS Work> node Script1.js
[ 0, 1, 2, 3, 4 ]
[ 0, 1, 2, [ 3, 4 ] ]
PS C:\Users\Tomer\Documents\AppWork\FullStackDevelopmentBatch1\JS Work>
```

Array.prototype.flatMap()

The `flatMap()` method returns a new array formed by applying a given callback function to each element of the array, and then flattening the result by one level. It is identical to a `map()` followed by a `flat()` of depth 1 (`arr.map(...args).flat()`), but slightly more efficient than calling those two methods separately.

```
// JavaScript source code---- single line comment
```

```
const arr1 = [1, 2, 1];
```

```
const result = arr1.flatMap((num) => (num === 2 ? [2, 2, 5, 8] : 1));
```

```
console.log(result);
```



```
Developer PowerShell
+ Developer PowerShell
PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work> node Script1.js
[ 1, 2, 2, 5, 8, 1 ]
PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work>
```

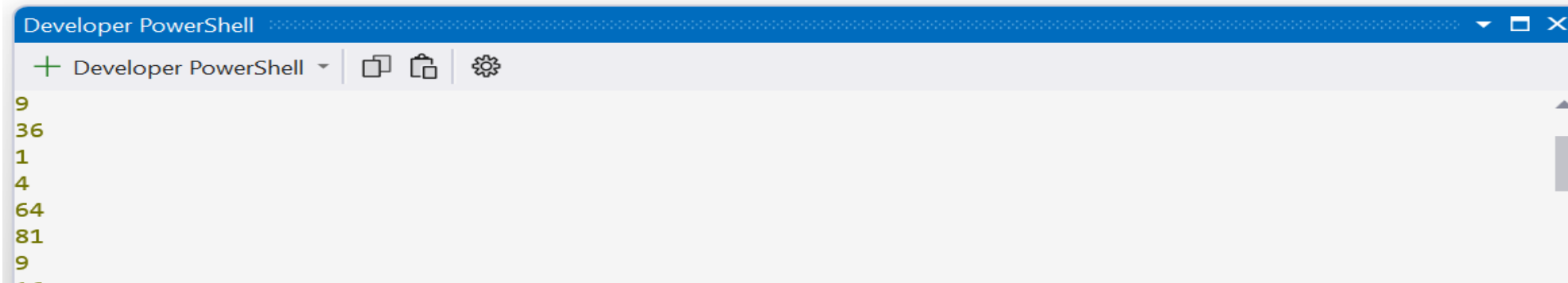
Array.prototype.forEach()

The `forEach()` method executes a provided function once for each array element.

```
// JavaScript source code---- single line comment
```

```
const arr1 = new Array(3, 6, 1, 2, 8, 9, 3, 4, 0);  
arr1.forEach(x => console.log(x)); // way to iterate the elements
```

```
arr1.map((x) => x * x).forEach(x => console.log(x));  
// map into square and iterate throug foreach
```



Array.prototype.reduce()

The reduce() method executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.

The first time that the callback is run there is no "return value of the previous calculation". If supplied, an initial value may be used in its place. Otherwise the array element at index 0 is used as the initial value and iteration starts from the next element (index 1 instead of index 0).

Perhaps the easiest-to-understand case for `reduce()` is to return the sum of all the elements in an array:

```
<script>  
  const arr = [2, 4, 6, 7];  
  let result = arr.reduce((prev, curr) => prev + curr);  
  console.log(result);  
</script>
```

O/P: 19

import/export(will be discuss with react)

The export declaration is used to export values from a JavaScript module. Exported values can then be imported into other programs with the import declaration or dynamic import. The value of an imported binding is subject to change in the module that exports it — when a module updates the value of a binding that it exports, the update will be visible in its imported value.

- Every module can have two different types of export, *named export and default export*.
- You can have multiple named exports per module but **only one default export**

JS Prototyping

Every JavaScript object has a prototype. The prototype is also an object.
All JavaScript objects inherit their properties and methods from their prototype.

- The `Object.prototype` is on the top of the prototype chain.
- All JavaScript objects (`Date`, `Array`, `RegExp`, `Function`, `Number`...) inherit from the `Object.prototype`.
- All JavaScript objects inherit the properties and methods from their prototype. In the image above `Array.prototype` inherits from `Object.prototype`.
- Objects created using an object literal, or with `new Object()`, inherit from a prototype called `Object.prototype`.
- Objects created with `new Date()` inherit the `Date.prototype`.

null



__proto__

Object.prototype

```
{  
  ...  
}
```



__proto__



__proto__

__proto__



Array.prototype

```
{  
  slice: ...,  
  other array methods  
}
```



__proto__

```
[1, 2, 3]
```

Function.prototype

```
{  
  apply: ...,  
  other function methods  
}
```



__proto__

```
function f(args) {  
  ...  
}
```

Number.prototype

```
{  
  toPrecision: ...,  
  other number methods  
}
```



__proto__

```
5
```

Template String

- String interpolation is replacing placeholders with values in a string literal.
- The string interpolation in JavaScript is done by template literals (strings wrapped in backticks `) and `${expression}` as a placeholder.

```
const number = 12;
```

```
const msg = `The number is ${number}`; \\\tilt  
console.log(msg);
```

The placeholders

A great benefit of the template strings is the ability to easily inject dynamic values into the string using placeholders. The expression inside the placeholder is evaluated during runtime, and the result is inserted into the string.

The placeholder has a special format: `${expressionToEvaluate}`. The expression inside the placeholder can be anything that evaluates to a value (but usually to a string):

- variables: `${myVar}`
- operators: `${n1 + n2}`, `${cond ? 'val 1' : 'val 2'}`
- even function calls `${myFunc('argument')}`

Here's an example:

```
let msg1 = "Welcome";  
  
let msg2 = "to template in Java script";  


---

let expression = `hey,${msg1},${msg2}`;  
  
console.log(expression);
```

Note:

`\${msg1}, \${msg2}!` is a template string having placeholders `${msg1}` and `${msg2}`.

You can put any expression inside the placeholder: either an operator, a function call, or even more complex expressions.

```
let n1 = 23;
```

```
let n2 = 100;
```

```
let sum = `sum of ${n1} and ${n2} is=${n1 + n2}`;
```

```
function sum(n1, n2) {
```

```
    return n1 + n2;
```

```
}
```

```
let fun = `sum is ${sum(2, 5)}`;
```

```
    console.log(fun);
```

Insert Array into template string:

```
const arr = [2, 5, 6, 12, 14, 20];  
const msg = `array elements are:${arr}`;  
console.log(msg);
```

The placeholder `${numbers}` contains an array of numbers. `toString()` array method executes `array.join(',')` when the array is converted to a string. The string interpolation result is 'The numbers are 1,2,3'.

Escaping placeholders

You cannot use the sequence of characters "\${someCharacters}" without escaping it because the placeholder format \${expression} has a special meaning in the template literals.

Let's create a string literal containing the sequence of characters \${abc}:

```
const msg = `Welcome to \${abc}`;
```

```
console.log(msg); // o/p:
```

```
// JavaScript source code---- single line comment
```

```
const msg = "Java Script tutorial";  
const msg1 = `welcome to \${msg}`;  
console.log(msg1); // welcome to ${msg}
```

Complex Expression:

```
const n1 = 12;
```

```
const n2 = 6;
```

```
const result = `sum=${n1 + n2}, subtraction=${n1 - n2},
```

```
multiplication=${n1 * n2}`;
```

```
console.log(result);
```

Single quotes in placeholders:

```
function getLoadingMessage(isLoading) {  
    return `Data is ${isLoading ? 'loading...' : 'done!'}`;  
}  
  
console.log(getLoadingMessage(true));
```

JS Objects

JavaScript objects can be thought of as collections of key-value pairs. As such, they are similar to:

- Dictionaries in Python.
- Hashes in Perl and Ruby.
- Hash tables in C and C++.
- HashMaps in Java.
- Associative arrays in PHP.

JavaScript objects are hashes. Unlike objects in statically typed languages, objects in JavaScript do not have fixed shapes — properties can be added, deleted, re-ordered, mutated, or dynamically queried at any time. Object keys are always strings or symbols — even array indices, which are canonically integers, are actually strings under the hood.

```
const obj = {  
  name: "Carrot",  
  for: "Max",  
  details: {  
    color: "orange",  
    size: 12,  
  },  
};
```

```
let Book = {  
  name: "Headfirst",  
  edition: 2012,  
  programming_language: "java",  
  head_office_publisher:  
    {  
      country: "India",  
      city: "New Delhi",  
      pin: 205004  
    }  
}  
  
console.log(Book.edition);  
console.log(Book.head_office_publisher);  
console.log(Book.head_office_publisher.country);
```

```
Developer PowerShell  
+ Developer PowerShell  
PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work> node  
2012  
{ country: 'India', city: 'New Delhi', pin: 205004 }  
India  
PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work>
```

Objects are always references, so unless something is explicitly copying the object, mutations to an object would be visible to the outside.

```
const obj = {};  
function doSomething(o) {  
    o.x = 1;  
}  
doSomething(obj);  
console.log(obj.x); // 1
```

Anonymous functions

JavaScript lets you create anonymous functions — that is, functions without names. In practice, anonymous functions are typically used as arguments to other functions, immediately assigned to a variable that can be used to invoke the function, or returned from another function.

// JavaScript source code---- single line comment

```
const avg = function (...args) {  
    let sum = 0;  
    for (const item of args) {  
        sum += item;  
    }  
    return sum / args.length;  
};
```

That makes the anonymous function invocable by calling avg() with some arguments — that is, it's semantically equivalent to declaring the function using function avg() {} declaration syntax.

```
let arr = [2, 4, 5, 6, 12, 8, 4];
```

```
let result = avg(...arr);  
console.log(result);
```

There's another way to define anonymous functions — using an [arrow function expression](#).

```
const avg = (...args) => {  
  let sum = 0;  
  for (i = 0; i < args.length; i++) {  
    sum += args[i];  
  }  
  return sum / args.length;  
}
```

```
let arr = [2, 4, 5, 6, 12, 8, 4];
```

```
let result = avg(...arr);  
console.log(result);
```

JS Closures

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
function init() {  
    let name = "Mozilla"; // name is a local variable created by init()  
    function displayName() {  
        // displayName() is the inner function, that forms the closure  
        console.log(name); // use variable declared in the parent function  
    }  
    displayName();  
}  
init();
```

```
function makeFunc() {  
    const name = "Java Script";  
  
    function displayName() {  
        console.log(name);  
    }  
  
    return displayName;  
}
```

```
const myFunc = makeFunc();
```

```
myFunc();
```

Running this code has exactly the same result as the previous example of the initialization above. What's different (and interesting) is that the displayName() inner function is returned from the outer function before being executed.

Here's a slightly more interesting example—a makeAdder function

```
// javascript source code----- single line comment  
function createAdder(x) {  
    return function (y) {  
        return x + y;  
    };  
}
```

```
let oneadd = createAdder(5);
```

```
let result = oneadd(10);
```

```
console.log(result)
```

```
function changeBy(val) {  
    privateCounter += val;  
}  
  
return {  
    increment() {  
        changeBy(1);  
    },  
  
    decrement() {  
        changeBy(-1);  
    },  
  
    value() {  
        return privateCounter;  
    },  
};  
})();
```

```
console.log(counter.value()); // 0.
```

```
counter.increment();  
counter.increment();  
console.log(counter.value()); // 2.
```

```
counter.decrement();  
console.log(counter.value()); // 1.
```

Promises in JS

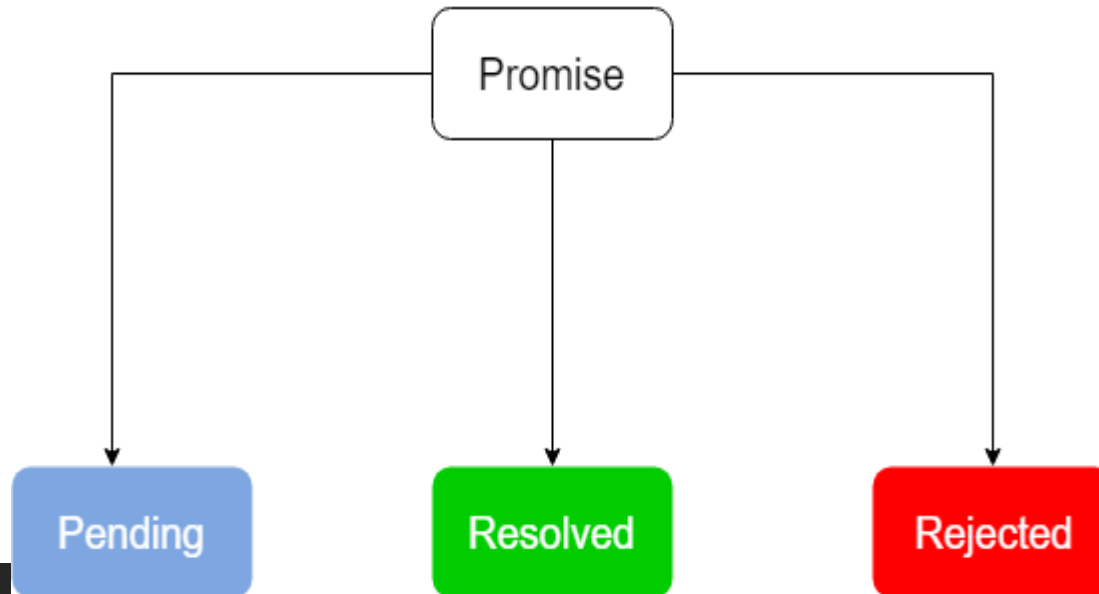
A Promise is an object that holds the future value of an **async operation**. For example, if we are requesting some data from a server, the promise promises us to get that data which we can use in future.

Before we get into all the technical stuff, let's understand the terminology of a Promise.

States of a Promise

A Promise in JavaScript just like a promise in real-world has 3 states. It can be 1) unresolved (pending), 2) resolved (fulfilled), or 3) rejected.

States of a Promise



- **Unresolved or Pending** —

- A Promise is pending if the result is not ready. That is, it's waiting

- ---

- for something to finish (for example, an async operation).

Resolved or Fulfilled —

A Promise is resolved if the result is available. That is, something finished (for example, an async operation) and all went well.

Rejected — A Promise is rejected if an error happened.

Creating a Promise

Most of the times you will be consuming promises rather than creating them, but it's still important to know how to create them.

```
//Creating a promise
```

```
const promise = new Promise((resolve, reject) = {
```

```
//body
```

```
});
```

We create a new promise using the Promise constructor it takes a single argument, a callback, also known as executor function which takes two callbacks, resolve and reject.

The executor function is immediately executed when a promise is created. The promise is resolved by calling the `resolve()` and rejected by calling `reject()`. For example:

```
— const promise = new Promise((resolve, reject) => { —  
    if (allWentWell) {  
        resolve('All things went well!');  
    } else {  
        reject('Something went wrong');  
    }  
});
```

Example Code

```
- const promise = new Promise((resolve, reject) => {  
  - let a = 12;  
  - if (a < 0) {  
    - resolve('Problem resolved');  
  }  
  - else {  
    - reject('something went wrong');  
  }  
- });
```

Consuming a Promise

Now that we know how to create a promise, let's understand how to consume an already created promise. We consume a promise by calling **then()** and **catch()** methods on the promise.

For example, requesting data from an API using `fetch` which returns a promise.

.then() syntax: `promise.then(successCallback, failureCallback)`

The `successCallback` is called when a promise is resolved. It takes one argument which is the value passed to `resolve()`.

The `failureCallback` is called when a promise is rejected. It takes one argument which is the value passed to `reject()`.

```

[-] const promise = new Promise((resolve, reject) => {
    |   let a = 12;
    |   if (a < 20) {
    |       |   resolve('Problem resolved');
    |       |   }
    |   else {
    |       |   reject('something went wrong');
    |       |   }
    |   });
[-] promise.then((msg) => {
    |   console.log("calling resolve:" + msg);
[-] }).catch((msg) => {
    |   console.log("calling reject:"+msg);
    |   })

```

Developer PowerShell

+ Developer PowerShell ▾ |   

S C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\JS Work> node Script1.js

calling reject: something went wrong

S C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\JS Work> node Script1.js

calling resolve: Problem resolved

Promise Chaining

- The `then()` and `catch()` methods can also return a new promise which can be handled by chaining another `then()` at the end of the previous `then()` method.

We use promise chaining when we want to resolve promises in a sequence.

```
const promise1 = new Promise((resolve, reject) => {
  resolve('Promise1 resolved');
});

const promise2 = new Promise((resolve, reject) => {
  resolve('Promise2 resolved');
});

const promise3 = new Promise((resolve, reject) => {
  reject('Promise3 rejected');
});
```

+ Developer PowerShell



```
PS C:\Users\Tomer\Documents\AppData\Local\Temp\
Promise1 resolved
Promise2 resolved
Promise3 rejected
PS C:\Users\Tomer\Documents\AppData\Local\Temp\
```

```
promise1
  .then((data) => {
    console.log(data); // Promise1 resolved
    return promise2;
  })
  .then((data) => {
    console.log(data); // Promise2 resolved
    return promise3;
  })
  .then((data) => {
    console.log(data);
  })
  .catch((error) => {
    console.log(error); // Promise3 rejected
  });
```


So what's happening here?

- When `promise1` is resolved, the `then()` method is called which returns `promise2`.
- The next `then()` is called when `promise2` is resolved which returns `promise3`.
- Since `promise3` is rejected, the next `then()` is not called instead `catch()` is called which handles the `promise3` rejection.

Promise.all()

This method takes an array of promises as input and returns a new promise that fulfills when all of the promises inside the input array have fulfilled or rejects as soon as one of the promises in the array.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise1 resolved');
  }, 2000);
}); const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise2 resolved');
  }, 1500);
}); Promise.all([promise1, promise2])
  .then((data) => console.log(data[0], data[1]))
  .catch((error) => console.log(error));
```

Developer PowerShell

```
PS C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\JS Work> node Script1.js
Promise1 resolved Promise2 resolved
```

Here we have two promises where one is resolved after 2 seconds, and the other is rejected after 1.5 seconds.

As soon as the second promise is rejected after 1.5 seconds, the returned promise from Promise.all() is rejected without waiting for the first promise to be resolved.

Promises remaining

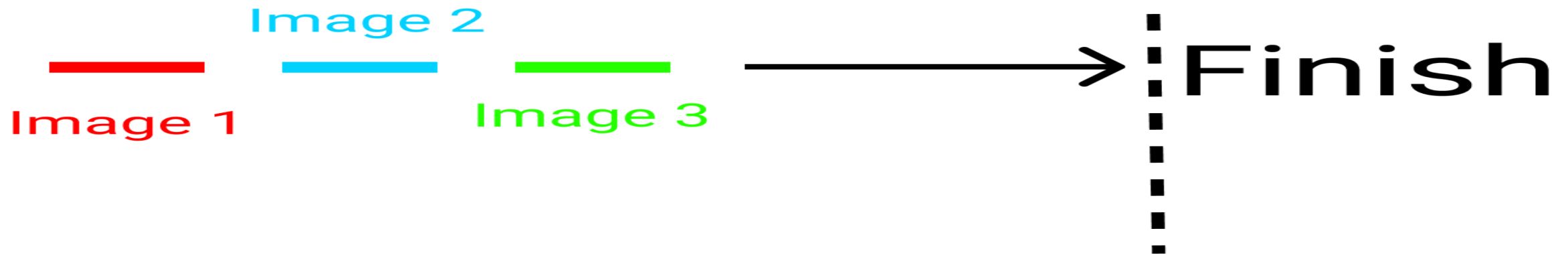
Async/Await in JavaScript

- This is supposed to be the better way to write promises and it helps us keep our code simple and clean.
- All you have to do is write the word `async` before any regular function and it becomes a promise.
- The keyword `await` makes JavaScript wait until a promise settles and returns its result.

Synchronous Flow

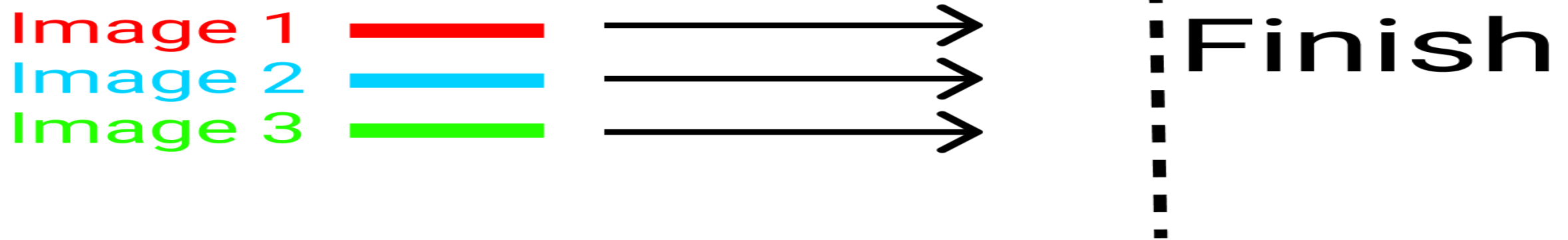
Synchronous system, three images are in the same lane. One can't overtake the other. The race is finished one by one. If image number 2 stops, the following image stops.

Synchronous




Asynchronous system, the three images are in different lanes. They'll finish the race on their own pace. Nobody stops for anybody:

Asynchronous




```
console.log("I");
```

```
// This will be shown after 2 seconds
```

```
 setTimeout(() => {  
  console.log("eat");  
}, 2000)
```

```
console.log("Ice Cream")
```

Promises and `async/await`

- We can either use Promises or `async/await` to develop projects in java script.
- The first things you have to understand that `async/await` syntax is just syntactic sugar which is meant to augment promises.
- the return value of an `async` function is a promise.
- **`async/await`** syntax gives us the possibility of writing asynchronous in a synchronous manner.
- **`async/await`** might be more readable than promise chaining.
- This is especially true when the amount of promises which we are using increases.
- Both Promise chaining and `async/await` solve the problem of callback hell and which method you choose is matter of personal preference.

Code: promises Vs async/await

```
function logFetch(url) {  
  return fetch(url)  
    .then(response => response.text())  
    .then(text => {  
      console.log(text);  
    }).catch(err => {  
      console.error('fetch failed', err);  
    });  
}
```

```
async function logFetch(url) {  
  try {  
    const response = await fetch(url);  
    console.log(await response.text());  
  }  
  catch (err) {  
    console.log('fetch failed', err);  
  }  
}
```

Why is async/await preferred over promises?

Concise and clean - We don't have to write `.then` and create an anonymous function to handle the response, or give a name data to a variable that we don't need to use. We also avoided nesting our code. `async/await` is a lot cleaner.

Error handling - `Async/await` makes it finally possible to handle both synchronous and asynchronous errors with the same `try/catch` format.

. **Debugging** - A really good advantage when using async/await is that it's much easier to debug than promises for 2 reasons: 1) you can't set breakpoints in arrow functions that return expressions (no body). 2) if you set a breakpoint inside a .then block and use debug shortcuts like step-over, the debugger will not move to the following .then because it only "steps" through synchronous code.

Error stacks - The error stack returned from the promises chain gives us no idea of where the error occurred and can be misleading. async/await gives us the error stack from async/await points to the function that contains the error which is a really big advantage

Final Result:

Async/await can help make your code cleaner and more readable in cases where you need complicated control flow. It also produces more debug-friendly code. And makes it possible to handle both synchronous and asynchronous errors with just try/catch.

Note: it is up to you in which concept you are comfortable either promises or Async/await.

Important Point

An async function always
returns a promise =>

```
const fun = async () => {  
    return "Asynchronous call";  
}  
  
fun().then(  
    console.log  
);
```

Developer PowerShell

+ Developer PowerShell

PS C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\JS Work> node Script1.js
Asynchronous call

```

const delayAndGetRandom = (ms) => {
  return new Promise(resolve => setTimeout(
    () => {
      const val = parseInt(Math.random() * 100);
      resolve(val);
    }, ms
  ));
};

```

```

async function fn() {
  const a = await 9;
  const b = await delayAndGetRandom(1000);
  const c = await 5;
  await delayAndGetRandom(1000);
  console.log("a=" + a);
  console.log("a=" + b);
  console.log("a=" + c);
  return a + b * c;
}

fn().then(console.log);

```

PS C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\JS Work> node Script1.js

a=9

a=14

a=5

79

PS C:\Users\Tomer\Documents\AppWork\Work\FullStackDevelopmentBatch1\JS Work>

Code Explanation:

- When `fn` is executed, the first line to be evaluated is `const a = await 9;`. It is internally transformed into `const a = await Promise.resolve(9);`.
- ~~➤ Since we are using `await`, `fn` pauses until the variable `a` gets a value. In this case, the promise resolves it to 9.~~
- `delayAndGetRandom(1000)` causes `fn` to pause until the function `delayAndGetRandom` is resolved which is after 1 second. So, `fn` effectively pauses for 1 second.
- Also, `delayAndGetRandom` resolves with a random value. Whatever is passed in the resolve function, that is assigned to the variable `b`.
- `c` gets the value of 5 similarly and we delay for 1 second again using `await delayAndGetRandom(1000)`. We don't use the resolved value in this case.
- Finally, we compute the result `a + b * c` which is wrapped in a Promise using `Promise.resolve`. This wrapped promise is returned.

Let Create a simple Ice Cream Maker

Steps to make ICE Cream once you receive order

- #1 Place Order
- #2 Cut The Fruit
- #3 Add water and ice
- #4 Start the machine
- #5 Select Container
- #6 Select Toppings
- #7 Serve Ice Cream

Let's store our data:

1. We can store gradients inside the objects like this:

```
let stocks = {  
  Fruits: ["strawberry", "grapes", "banana", "apple"]  
}  
  
console.log(stocks.Fruits[2]);
```

2. Other ingredients like:

Holder (Cone, Cup , Stick)

Toppings (Choclates, Sprinkles)

Add these too in stocks object:

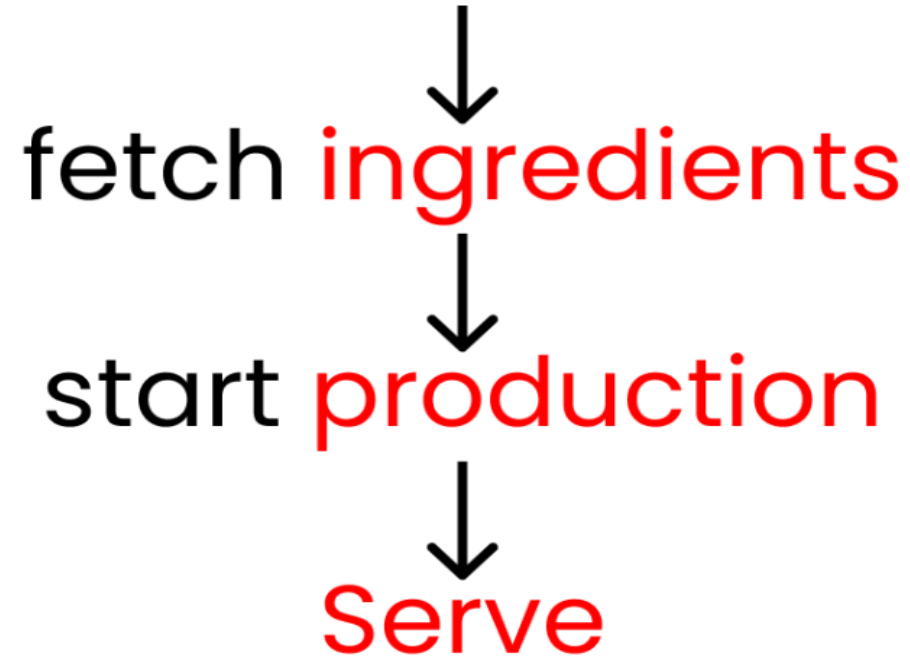
```
let stocks = {  
  Fruits: ["strawberry", "grapes", "banana", "apple"],  
  liquid: ["water", "ice"],  
  holder: ["cone", "cup", "stick"],  
  toppings: ["chocolate", "peanuts"],  
};
```

The entire business depends on what a customer **orders**. Once we have an order, we start production and then we serve ice cream. So, we'll create two functions :

- **Time**=> it provides the time delay among every steps of ice cream making

- **Production**=> it handle the complete process of ice cream production

- **Flow of data** => **order** from customer



Time(seconds)

#1 Place Order	→ 2
#2 Cut The Fruit	→ 2
#3 Add water and ice	→ 1
#4 Start the machine	→ 1
#5 Select Container	→ 2
#6 Select Toppings	→ 3
#7 Serve Ice Cream	→ 2

Chart contains steps to make ice cream

Time function

```
- function time(ms) {  
-  
-   return new Promise((resolve, reject) => {  
-  
-       if (is_shop_open) {  
-           setTimeout(resolve, ms);  
-       }  
-  
-       else {  
-           reject(console.log("Shop is closed"))  
-       }  
-   });  
- }
```

```
async function production() {  
  try {  
    await time(2000)  
    console.log(`${stocks.Fruits[0]} was selected`)  
  
    await time(1000)  
    console.log("production has started")  
  
    await time(2000)  
    console.log("fruit has been chopped")  
  
    await time(1000)  
    console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} added`)  
  
    await time(1000)  
    console.log("start the machine")  
  }  
}
```



```
await time(2000)
console.log(`ice cream placed on ${stocks.holder[1]}`)
```

```
await time(3000)
console.log(`${stocks.toppings[0]} as toppings`)
```

```
await time(2000)
console.log("Serve Ice Cream")
```

```
}
```

```
catch (error) {
  console.log("customer left")
}
```

```
}
```

```
... ..
```

Call the function: production()

node Script1.js

strawberry was selected

production has started

fruit has been chopped

water and ice added

start the machine

ice cream placed on cup

chocolate as toppings

Serve Ice Cream

PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\JS Work>

Rest API

There are a few different types of REST APIs. Let's look at the ones you will use in most cases.

GET—Get data from the API. For example, get a twitter user based on their username.

POST—Push data to the API. For example, create a new user record with name, age, and email address.

PUT—Update an existing record with new data. For example, update a user's email address.

DELETE—Remove a record. For example, delete a user from the database.

There are three elements in every REST API. The request, response, and headers.

Request—This is the data you send to the API, like an order id to fetch the order details.

Response—Any data you get back from the server after a successful / failed request.

Headers—Additional metadata passed to the API to help the server understand what type of request it is dealing with, for example “content-type”.

Advantages REST API

The real advantage of using a REST API is that you can build a single API layer for multiple applications to work with.

If you have a database that you want to manage using a web, mobile, and desktop application, all you need is a single REST API Layer.

XMLHttpRequest

- Before [JSON](#) took over the world, the primary format of data exchange was XML. XMLHttpRequest() is a JavaScript function that made it possible to fetch data from APIs that returned XML data.
- XMLHttpRequest gave us the option to fetch XML data from the backend without reloading the entire page.

Fetch API

- The Fetch API provides a JavaScript interface for accessing and manipulating parts of the protocol, such as requests and responses. It also provides a global `fetch()` method that provides an easy, logical way to fetch resources asynchronously across the network.
- Unlike XMLHttpRequest that is a callback-based API, Fetch is promise-based and provides a better alternative that can be easily used in service workers. Fetch also integrates advanced HTTP concepts such as CORS and other extensions to HTTP.

A basic fetch request looks like this:

```
[- async function logJSONData() {  
  :   const response = await fetch("http://example.com/movies.json");  
  :   const jsonData = await response.json();  
  :   console.log(jsonData);  
  : }  
[-
```


➤ Here we are fetching a JSON file across the network and printing it to the console. The simplest use of `fetch()` takes one argument — the path to the resource you want to fetch — and does not directly return the JSON response body but instead returns a promise that resolves with a `Response` object.

➤ The `Response` object, in turn, does not directly contain the actual JSON response body but is instead a representation of the entire HTTP response. So, to extract the JSON body content from the `Response` object, we use the `json()` method, which returns a second promise that resolves with the result of parsing the response body text as JSON.

The function returns a Response object that contains useful functions and information about the HTTP response, such as:

text() - returns the response body as a string

json() - parses the response body into a JSON object, and throws an error if the body can't be parsed

status and statusText - contain information about the HTTP status code

ok - equals true if status is a 2xx status code (a successful request)

headers - an object containing response headers, a specific header can be accessed using the get() function.

Add fetch API to node

In Visual Studio:

View → terminal:

npm install node-fetch

Add:

Npm install node-fetch@2.0

Create new Folder: UseFetchAPI

Go inside this folder using terminal:

cd UseFetchAPI

Now :

Initialize Node Project:

Now you are here: ../UseFetchAPI>

Write command:

npm init -y // it will create a package.json file inside your project folder(UseFetchAPI)

Open package.json(Add type:module and see here is index.js is default js file)

```
{
  "name": "usefetchapi",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "Tomer",
  "license": "ISC",
  "dependencies": {
    "node-fetch": "2.0"
  }
}
```

```
// javascript source code
```

```
import fetch from 'node-fetch';  
fetch('https://data.binance.com/api/v3/ticker/24hr')  
  .then(res => res.text())  
  .then(text => console.log(text));
```

Developer PowerShell

+ Developer PowerShell ▾



```
000","openTime":1681651464575,"closeTime":1681737864575,"firstId":-1,"lastId":-1,"count":0},{ "symbol":"VOXELETH","priceChange":"0.00000000",  
ightedAvgPrice":"0.00000000","prevClosePrice":"0.00020220","lastPrice":"0.00000000","lastQty":"0.00000000","bidPrice":"0.00000000","bidQty"  
000","askQty":"0.00000000","openPrice":"0.00000000","highPrice":"0.00000000","lowPrice":"0.00000000","volume":"0.00000000","quoteVolume":"0.  
75,"closeTime":1681737864575,"firstId":-1,"lastId":-1,"count":0},{ "symbol":"ALICEBNB","priceChange":"0.00000000","priceChangePercent":"0.0  
","prevClosePrice":"0.00524000","lastPrice":"0.00000000","lastQty":"0.00000000","bidPrice":"0.00000000","bidQty":"0.00000000","askPrice":"0.  
"openPrice":"0.00000000","highPrice":"0.00000000","lowPrice":"0.00000000","volume":"0.00000000","quoteVolume":"0.00000000","openTime":1681  
5,"firstId":-1,"lastId":-1,"count":0},{ "symbol":"ATOMTRY","priceChange":"7.40000000","priceChangePercent":"3.308","weightedAvgPrice":"227.  
000000","lastPrice":"231.10000000","lastQty":"6.72600000","bidPrice":"231.10000000","bidQty":"62.48000000","askPrice":"231.20000000","askQ  
.70000000","highPrice":"231.70000000","lowPrice":"222.40000000","volume":"10528.81400000","quoteVolume":"2393409.79320000","openTime":1682  
6,"firstId":1641610,"lastId":1643279,"count":1670},{ "symbol":"ETHUST","priceChange":"0.00000000","priceChangePercent":"0.000","weightedAvg
```

Fetch Data in JSON Format

// JavaScript source code

```
import fetch from 'node-fetch';  
fetch('https://data.binance.com/api/v3/ticker/24hr')  
  .then(res => res.json())  
  .then(text => console.log(text))  
  .catch(err => console.log('Request Failed', err))
```

Developer PowerShell

+ Developer PowerShell | [icon] [icon] [icon]

```
symbol: 'EOSBTC',  
priceChange: '-0.00000100',  
priceChangePercent: '-2.625',  
weightedAvgPrice: '0.00003736',  
prevClosePrice: '0.00003800',  
lastPrice: '0.00003710',  
lastQty: '243.60000000',  
bidPrice: '0.00003700',  
bidQty: '3416.60000000',  
askPrice: '0.00003710',  
askQty: '17346.00000000',  
openPrice: '0.00003810',  
highPrice: '0.00003810',  
lowPrice: '0.00003690',
```

Headers

```
.then(res => res.status)
```

Miscellaneous

then() callback

```
// JavaScript source code
```

```
import fetch from 'node-fetch';
fetch('https://data.binance.com/api/v3/ticker/24hr', {
  method: "GET",
  headers: { "Content-type": "application/json;charset=UTF-8" }
})
  .then(res => res.headers)
  .then(res => console.log(res))
  .catch(err => console.log('Request Failed',err))
```

Developer PowerShell

+ Developer PowerShell

PS C:\Users\Tomer\Documents\AppWorX Work\FullStackDevelopmentBatch1\UseFetchAPI> node index.js

```
Headers {
  [Symbol(map)]: [Object: null prototype] {
    'content-type': [ 'application/json;charset=UTF-8' ],
    'content-length': [ '161075' ],
    connection: [ 'close' ],
    date: [ 'Wed, 26 Apr 2023 16:04:55 GMT' ],
    server: [ 'nginx' ],
    'x-mbx-uuid': [ 'deb7502e-5687-4925-8ba9-adb9347fc36d' ],
```


Test

Your best quote that reflects your approach... “It’s one small step for man, one giant leap for mankind.”

- NEIL ARMSTRONG