Functional Programming Paradigm

Topics:

- Modules
- Functional Programming Paradigm

Modules:

Modules are different files of JavaScript. Earlier, there was very small interactivity, but as the internet usage increased, they came up with ECMA Script module.

Lets create a .js file such that everything start from it. So, either I can could wirte everything in one file, to the point of the 1000 lines. But this will create a problem if there is a bug inside it.

```
function add (...args) {
    let sum = 0
    for (const iterator of args) {
        sum += number
    }
    return sum
}

function cube(num) {
    return num**3
}

const result = cube(add(10, 52, 11))

const elem = document.createElement('h1')
elem.innerHTML = result

document.body.appendChild(elem)
```

This is a basic example. Instead of putting everything in one file, we can put the DOM related things in dom-manipulation.js file and then in math-operations.js we can put all the maths computations. This actually eases the work as we will split the one big file into smaller modules.

In the html page, we need to put them in the right order to get the result as we expected.

```
    <script src="dom-manipulation.js"></script>
    <script src="math-operators.js"></script>
```

The DOM will be ready, but it is not able to find the cube because we have put the math-operators.js below the dom-manipulation.js

```
    <script src="math-operators.js"></script>
    <script src="dom-manipulation.js"></script>
```

It is not easy to manage the entire code figuring out the order in which we have set the .js files.

Let's say math-operation.js is a module. Image it is a closure and so has its own scope. So, first we have to use some syntax so that it can be used. Export this function cube to anyone who is importing it.

We have only exported and did not import it. So,

```
import { cube } from 'math-operations.js'
```

There are two ways to create a module. So instead of .js file, we can rename them as .mjs file; this signifies to the javascript that the specified file is a module and not a simple .js file. With this done, if we write the same line again,

```
import { cube } from 'math-operations.mjs'
```

The other way is, in the html file, in the script tag, we can write it in the type attribute as "text/module". We can solve this by using the live server extension in the vs code. This file will not show the same error. So, we need to change the path in the dom-manipulation.mjs file.

```
import { cube, add } from './math-operations.mjs'
```

Functionality of Modules:

Digging deeper into the functionalities of modules. Let's add a variable in dom-manipulation.mjs file and in math-operations.mjs file we will declare another variable and console.log at the end of the file. Every module has its own scope so there will not be any collision. So, as we have use the same "myVariable" it will print both the values in console.log

In dom-manipulation.mjs

```
let myVariable = 456
```

In math-operations.mjs

```
let myVariable = 123
```

The scope is different for each file. So, first in dom-manipulation, it would declare the variable and then again in math-operations file, it would get another value for the same variable. The reason value is changed is because the math-operations does not know any declarations done in the parent file, basically different module.

The two files do not know implicitly internally. All modules have individual scope unless and until you import something from another file, there won't be any name collision. We need to use export to get the value from different modules. So,

```
export let myVariable = 123
```

If we put the value in the import as such,

```
import { cube, add, myVariable } from './math-operations.mjs'
```

This makes the dom-manipulation aware of myVariable coming from math-operations file, which will give us an error.

Currently, there is no default export. So instead of

```
import { cube, add, myVariable } from './math-operations.mjs'
```

If we write

```
import cube from './math-operations.mjs'
```

The output shows that it does not provide an export named 'default'. So, if we use write export default abc in the file that has the values and import abc from './file.js' whatever is the default value, will be imported. So, abc is getting the value of cube function itself. Since we have named abe, it is saying not defined. It shows that add is not defined as it is not exported as a default so we can write it in the curly braces beside cube in the import

```
import cube, { add } from './math-operations.mjs'
```

If we use export default {} (exporting a default object) with cube: cube so that the first cube is property name and the second cube is the actual function reference.

```
export default {
    cube: cube,
    add: add
}
```

Instead of doing de-structuring, we can just export or import the entire file.

```
let myVariable = 123;

import test from './math-operations.mjs'
console.log(test)
const result = test.cube(test.add(10,52,11))

const element = document.createElement('h1')
elem.innerText = result

document.body.appendChild(elem)
```

We can use this if we come across a situation where we need to export some part and not all the information.

In case you have the property name and function name same, then we can do the short-hand definition in objects. For example, instead of writing

```
export default {
    cube: cube,
    add: add
}
```

We can simply write it down as

```
export default {
    cube,
    add
}
```

And JavaScript would read it as cube: cube, add: add.

localStorage & sessionStorage:

In order to save details persistently, we need to have a database. But for small applications as to-do applications, if we are able to save the details in some place then it would be easier or better for us.

Two-types:

- localStorage
- sessionStorage

localStorage → stores data permanently as per User’s wish

sessionStorage → stores data only for the current session

The read-only localStorage property allows you to access a Storage object for the Document's origin; the stored data is saved across browser sessions. localStorage is similar to sessionStorage, except that while data stored in localStorage has no expiration time, data stored in sessionStorage gets cleared when the page session ends — that is, when the page is closed. (Data in a localStorage object created in a "private browsing" or "incognito" session is cleared when the last "private" tab is closed.)

Data stored in either localStorage is specific to the protocol of the page. In particular, data stored by a script on a site accessed with HTTP (e.g., http://example.com) is put in a different localStorage object from the same site accessed with HTTPS (e.g., https://example.com).

We can different functions in order to set data. We can wirte,

```
localStorage.setItem('computedValue', result)
```

The result will be type casted, as the localStorage and sessionStorage, store data only in string form.

```
localStorage.setItem('computedValue', result)
localStorage.setItem('userName','Unnamed')

const value = localStorage.getItem('userName')

alert(value)
```

localStorage.length will give 2 as we have two data entries in our local storage.

localStorage.clear will all the application data from the local storage.

For sessionStorage

```
sessionStorage.setItem('title', 'Sinha')
```

Using the alert here,

```
alert(sessionStorage.getItem('title'))
```

This will give us an alert box every time we refresh the page. But once we close the page and reopen the tab, all the data of sessionStorage is lost.