JavaScript –ES6

Day 3

Agenda

- New ES6 syntax
- Arrow Functions
- Destructuring
- ES6 Modules
- ES6 Classes
- Promises
- ES6 collections

Agenda

- Array extensions
- Object extensions
- String extensions

Day 2 -Recap

- Arrow Functions
- Destructuring
- ES6 Classes

Day 3

- ES6 Modules
- Promises

ES-6 modules

- An ES6 module is a JavaScript file that executes in strict mode only. It means that any variables or functions declared in the module won't be added automatically to the global scope.
- A module is just a file. One script is one module. As simple as that.

- Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one:

- export keyword labels variables and functions that should be accessible from outside the current module.
- import allows the import of functionality from other modules.

example

- // sayHi.js

```
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

- As modules support special keywords and features, we must tell the browser that a script should be treated as a module, by using the attribute <script type="module">.

```
<script type="module" src="./say.js">
  console.log(sayHi(john));
</script>
```

Note: Modules work only via HTTP(s), not locally

If you try to open a web-page locally, via file:// protocol, you'll find that import/export directives don't work. Use a local web-server, such as static-server or use the "live server" capability of your editor, such as VS Code Live Server Extension to test modules.

- Everything inside an ES6 module is private by default, and runs in strict mode (there’s no need for 'use strict').
- Public variables, functions and classes are exposed using export.
- Exposed modules are called into other modules using import
- Modules must be included in your HTML with type="module", which can be an inline or external script tag.

Modules Export-Syntax 1

- //using multiple export keyword

export component1

export component2

...

...

export componentN

Modules Export-Syntax 1 Example

- In some.js file

```
export let company = "tekisky"

export let getCompany = function(){
  return company.toUpperCase()
}

export let setCompany = function(newValue){
  company = newValue
}
```

Modules Export –Syntax 2

- //using single export keyword

```
export {component1,component2,....,component}
Example
 let company = "tekisky"
 let getCompany = function(){
   return company.toUpperCase()
}
 let setCompany = function(newValue){
   company = newValue
}
export {company,getCompany,setCompany}
```

Modules Export –Syntax 3

- export default component_name

Example

```
 let company = "tekisky";
 let getCompany = function(){
   return company.toUpperCase()
}
 let setCompany = function(newValue){
   company = newValue
}
export default company;
```

Modules Import –Syntax

- import {component1,component2..componentN} from module_name

eg import {company ,getCompany,setCompany} from './some.js'

- However, while importing named exports, they can be renamed using the as keyword. Use the syntax given below −

import {original_component_name as new_component_name }

eg import {company as x}

- All named exports can be imported onto an object by using the asterisk * operator.

import * as variable_name from module_name

eg : import * as myCompany from './company1.js'

console.log(myCompany.getCompany())

console.log(myCompany.company)

Callback

- A Callback is a way to handle the function execution after the completion of the execution of another function.
- A Callback would be helpful in working with events. In Callback, a function can be passed as a parameter to another function.

setTimeout()

- The setTimeout() method executes a block of code after the specified time. The method executes the code only once.
- The commonly used syntax of JavaScript setTimeout is:

- setTimeout(function, milliseconds);
- Its parameters are:

- function - a function containing a block of code
- milliseconds - the time after which the function is executed
- The setTimeout() method returns an intervalID, which is a positive integer.

program to display a text using setTimeout method

```
// function greet() {
    console.log('Hello world');
}
setTimeout(greet, 3000);
console.log('This message is shown first')
```

- In the above program, the setTimeout() method calls the greet() function after 3000 milliseconds (3 second).
- Hence, the program displays the text Hello world only once after 3 seconds.

Display Time Every 3 Second

```
// program to display time every 3 seconds
function showTime() {
    // return new date and time
    let dateTime= new Date();
    // returns the current local time
    let time = dateTime.toLocaleTimeString();
    console.log(time)
    // display the time after 3 seconds
     setTimeout(showTime, 3000);
}
// calling the function
showTime();
```

- You can also pass additional arguments to the setTimeout() method. The syntax is:
- setTimeout(function, milliseconds, parameter1, ...paramenterN);
- When you pass additional parameters to the setTimeout() method, these parameters (parameter1, parameter2, etc.) will be passed to the specified function.

For example

- // program to display a name

```
function greet(name, lastName) {
    console.log('Hello' + ' ' + name + ' ' + lastName);
}
// passing argument to setTimeout
setTimeout(greet, 1000, 'John', 'Doe');
```

In the above program, two parameters John and Doe are passed to the setTimeout() method. These two parameters are the arguments that will be passed to the function (here, greet() function) that is defined inside the setTimeout() method.

clearTimeout()

- The syntax of clearTimeout() method is:
- clearTimeout(intervalID);
- Here, the intervalID is the return value of the setTimeout() method.

clearTimeout() Method example

- // program to stop the setTimeout() method

```
let count = 0;
// function creation
function increaseCount(){
   // increasing the count by 1
   count += 1;
   console.log(count)
}
let id = setTimeout(increaseCount, 3000);
```

- // clearTimeout

```
clearTimeout(id);
console.log('setTimeout is stopped.');
```

ES6 Promises

- A Promise represents something that is eventually fulfilled. A Promise can either be rejected or resolved based on the operation outcome.

- ES6 Promise is the easiest way to work with asynchronous programming in JavaScript. Asynchronous programming includes the running of processes individually from the main thread and notifies the main thread when it gets complete. Prior to the Promises, Callbacks were used to perform asynchronous programming.

How Does Promise work?

- The Promise represents the completion of an **asynchronous operation**. It returns a single value based on the operation being **rejected** or **resolved**. There are mainly three stages of the Promise, which are shown below

Pending - It is the initial state of each Promise. It represents that the result has not been computed yet.

Fulfilled - It means that the operation has completed.

Rejected - It represents a failure that occurs during computation.

- Once a Promise is fulfilled or rejected The **Promise()** constructor takes two arguments that are **rejected** function and a **resolve** function. Based on the asynchronous operation, it returns either the first argument or second argument.

- **A promise can be created using Promise constructor.**

```
var promise = new Promise(function(resolve, reject){
    //do something
});
```

Parameters

Promise constructor takes only one argument which is a callback function (and that callback function is also referred as anonymous function too).

Callback function takes two arguments, resolve and reject

Perform operations inside the callback function and if everything went well then call resolve.

If desired operations do not go well then call reject.

Syntax- ex-1

```
let p = new Promise(function(resolve,reject){
  let workDone = true; // some time consuming work
    if(workDone){
    //invoke resolve function passed
    resolve('success promise completed')
  }
  else{
    reject('ERROR , work could not be completed')
  }
})
//result value is coming from resolve parameter
p.then((result)=>{console.log(result)});
//result value is coming from reject parameter
p.catch((result)=>{console.log(result)});
```

Chaining then ex-2

```
let countValue = new Promise(function (resolve, reject) {
  resolve("Promise resolved");
});//without condition always fullfil

// executes when promise is resolved successfully

countValue
  .then(function successValue(result) { //result value coming from resolve parameter
    console.log(result);
  })

  .then(function successValue1() {
    console.log("You can call multiple functions this way.");
  });
```

Eg-3

```
 // returns a promise
let countValue = new Promise(function (resolve, reject) {
  reject('Promise rejected');
}); //it will always reject as no condition calling always reject
// executes when promise is resolved successfully
countValue.then(
   function successValue(result) {
      console.log(result);
   },
 )
// executes if there is an error
.catch(
   function errorValue(result) {
      console.log(result);
   }
);
```

JavaScript finally() method

The finally() method gets executed when the promise is either resolved successfully or rejected. For example,

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
    // could be resolved or rejected
    resolve('Promise resolved');
});
// add other blocks of code
countValue.finally(
    function greet() {
        console.log('This code is executed.');
    }
);
```