Introduction to JavaScript

Teaching Faculty: Umur INAN

Prepared by Muhyieddin **AL-TARAWNEH**, Umur **INAN**

ECMAScript

- ECMAScript (or ES) is a JavaScript standard meant to ensure the interoperability of web pages across different web browsers.

- It is standardized by Ecma International according to the document ECMA-262.

- ECMAScript is commonly used for client-side scripting on the World Wide Web, and it is increasingly being used for writing server applications and services using Node.js.

ECMAScript

- ECMAScript is the standard JS is based on since there are many engines.
- JS Engine
  - is a software component that executes JavaScript code.
  - JavaScript engines are typically developed by web browser vendors, and every major browser has one.

DESTRUCTURING

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car, truck, suv] = vehicles;

let person = {
    firstName: 'Umur',
    lastName: 'Inan'
};
let { firstName, lastName } = person;
```

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Arrow functions

- Arrow functions were introduced in ES6.
- Arrow functions allow us to write shorter function syntax.
- The handling of this is also different in arrow functions compared to regular functions.

Arrow functions

- There are no binding of this.
- In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.
- With arrow functions the this keyword always represents the object that defined the arrow function.

Bind()

- The bind() method creates a new function that, when called, has its this keyword set to the provided value.
- It just returns a new version of the function whose this sets to thisArg argument.
- `fn.bind(thisArg[, arg1[, arg2[, ...]]])`

Bind()

```
const person = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}

const member = {
  firstName:"Umur",
  lastName: "Inan",
}
let fullName = person.fullName.bind(member);
```

call()

- The call() method calls a function with a given this value and arguments provided individually.

- It can be used to invoke (call) a method with an owner object as an argument (parameter).

call()

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}

person.fullName.call(person1);
```

EXPORTS

- The export statement is used when creating JavaScript modules to export live bindings to functions, objects, or primitive values from the module so they can be used by other programs with the import statement.

- There are two types of exports
    - Named Exports (Zero or more exports per module)
    - Default Exports (One per module)

Named Exports

- Named exports are useful to export several values.
- During the import, it is mandatory to use the same name of the corresponding object.

Default Exports

- Default exports are created by including a default tag in the export.
- One can have only one default export per file.

Export

```
export const myNumbers = [1, 2, 3, 4];
const animals = ['Panda', 'Bear', 'Eagle'];

export default function myLogger() {
  console.log(myNumbers, pets);
}

export class Alligator {
  constructor() {
    // ...
  }
}
```

IMPORT

- Import statement is used to import bindings that are exported by another module.

```
import defaultExport from 'module_name';
import * as name from 'module_name';
import { content } from 'module_name';
import { content as alias } from 'module_name';
import { content , content2 } from 'module_name';
```

CLASS

- Classes are a template for creating objects. They encapsulate data with code to work on that data.
- Classes in JS are built on prototypes. Prototypes are the mechanism by which JavaScript objects inherit features from one another.
- Classes are in fact "special functions", and just as you can define a function.
- An important difference between function declarations and class declarations is that function declarations are hoisted and class declarations are not.

CLASS

```
class Student extends Person {
constructor(name){
   super();
   this.name = name;
}
sayHi = () => 'Hi ' + this.name;
}

const student1 = new Student('Bob');
console.log(student1.sayHi());
```

Constructor

- The constructor method is a special method for creating and initializing an object created with a class.
- There can only be one special method with the name "constructor" in a class.
- A constructor can use the super keyword to call the constructor of the super class.

Extend

- The extends keyword is used in class declarations or class expressions to create a class as a child of another class.
- If there is a constructor present in the subclass, it needs to first call super() before using "this".

SPREAD OPERATOR

- Used to split up array elements or object properties.
- Used to can expand, copy an array, or clone an object.

```
let obj1 = { foo: 'bar', x: 42 };

let clonedObj = { ...obj1 };

let mergedObj = { ...obj1, ...obj2 };
```

Callback hell

```
go.addEventListener("click", function(e) {
  const el = e.currentTarget;
  setTimeout(function() {
    el.classList.add("circle");
    setTimeout(function() {
      el.classList.add("red");
      setTimeout(function() {
        el.classList.remove("circle");
        setTimeout(function() {
          el.classList.remove("red");
          setTimeout(function() {
            el.classList.add('fadeOut');
          })
        }, 300);
      }, 300);
    }, 500);
  }, 2000);
});
```

PROMISES

- A promise is a special JavaScript object that links the “producing code” and the “consuming code” together.
- A promise is defined as a proxy for a value that will eventually become available.
- Promises are one way to deal with code, without getting stuck in callback hell.

STATES

- Fulfilled
    - Action related to the promise succeeded.
- Rejected
    - Action related to the promise failed.
- Pending
    - Promise is still pending i.e. not fulfilled or rejected yet.

BENEFITS

- Improves Code Readability.
- Better handling of asynchronous operations.
- Better flow of control definition in asynchronous logic.
- Better Error Handling.

Creating a promise object

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

- Promise constructor takes only one argument which is a callback function.
- 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.

Consuming a promise object

```
promise.then(successFn)
       .catch(errorFn)
       .finally(alwaysFn)
```

Async await

- Async/Await is just syntactic sugar for .then() calls on a promise.
- Async and await make promises easier to write.
- Async makes a function return a Promise.
- Await makes a function wait for a Promise.
- The await keyword is only valid inside async functions.

Async

- It simply allows us to write promises based code as if it was synchronous and it checks that we are not breaking the execution thread.
- It operates asynchronously via the event-loop.
- Async functions will always return a value.
- It makes sure that a promise is returned and if it is not returned then JavaScript automatically wraps it in a promise which is resolved with its value.

Async

```
const getData = async() => {
    let data = "Hello World";
    return data;
}

getData().then(data => console.log(data));
```

AWAIT

```
let promise = new Promise(function (resolve, reject) {
    resolve('Promise resolved')}, 4000);
});
async function asyncFunc() {
    try{
       let result = await promise;
    }catch(err){
      // handle it
   }
}
asyncFunc();
```