Advanced Javascript concepts

BCA III

ES6 js

- ECMAScript 2015 was the second major revision to JavaScript.
- ECMAScript 2015 is also known as ES6 and ECMAScript 6.
- This chapter describes the most important features of ES6.

ES6+ Features

- Scopes = block level scope
- const to define for constants
- template strings are used to define long and lengthy strings
- While using template strings we can do variable substitution
- Destructuring assign the array or objects properties to variables
- New for loop for-of
- data structure like map and set (map is used to store key value pairs and set can store any number of values)
- classes are used to follow prototype patterns
- module support (Export modules and use them using import)

Let = Block level scope

```
<script>
function letDemo(){
    var i = 10;

    for(var i=0;i<=20;i++){
        console.log(i);
    }

    console.log(i);
}

letDemo();
</script>
```

```
<script>
function letDemo(){
    var i = 10;

    for(let i=0;i<=20;i++){
        console.log(i);
    }

    console.log(i);
}

letDemo();
</script>
```

Const

```
<body>
<script>
const pi = 3.14;
pi = 3.79;
console.log(pi);
</script>
</body>
</html>
```

```
const product = {};
product={};
</script>
</body>
```

```
const product = {};
product['name']="Iphone";
console.log(product);
</script>
```

```
<script>
"use strict";
const pi = 3.14;
//pi = 3.79;
console.log(pi);

const product = Object.freeze({})
product['name']="Iphone";
console.log(product);
</script>
```

Template string using back tick

```
<script>
let quote = "You are the\n"+
"Creator of\n"+
"Your Destiny\n";
</script>
```

```
<script>
let quote = `You are the
creator
of your
destiny`;

console.log(quote);
</script>
```

Variable substitution

```
<script>
let name ='John';
let quote = `You are the
creator
of your
destiny ${name}`;

console.log(quote);
</script>
```

Arrow Functions

Arrow function

- Arrow functions allows a short syntax for writing function expressions.
- You don't need the function keyword, the return keyword, and the curly brackets.

Arrow function

```
// ES5

var x = function(x, y) {

   return x * y;

}

// ES6

const x = (x, y) => x * y;
```

First arrow function

```
<script>
let x = ()=>console.log("Hello");
x();
</script>
```

Arrow function passing parameters

```
<script>
let add = (num1,num2)=>{
    return num1+num2;
}

let div = (num1,num2)=>{
    return num1/num2;
}


console.log(add(2,3));
console.log(div(10,5));
</script>
```

Characteristics of arrow function

- Arrow functions are written with the => symbol, which makes them compact.
- They don’t have their own this. They inherit this from the surrounding context.
- For functions with a single expression, the return is implicit, making the code more concise.
- Arrow functions do not have access to the arguments object, which is available in regular functions.

Objects

Objects

```
const student = {
  name: "Ram kumar",
  address : "Dhanhadhi kailali",
  roll     :12,
  concat : function() {
    return this.name + " " + this.address;
  }
};
```

Promises

Introduction

A promise in real life is like a placeholder for something that we are going to do for somebody in the future.

It's the same in case of Es6 version of javascript. Promise helps us with asynchronous programming.

connect()
dbCall()
restCall()

First Promise

```
<script>

let promise = new Promise(()=>{
    setTimeout(()=>{
        console.log("Working asynchronously")}
        ,1000);
});

</script>
```

Promise in function with resolve and reject

```
<script>

function myAsyncFun(){
let promise = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        console.log("Working asynchronously")
        resolve();
    }
        ,1000);
});

return promise;
}
```

Promise Error handling

```
function myAsyncFun(){
let promise = new Promise((resolve,reject)=>{
    let error = true;
    setTimeout(()=>{
        console.log("Working asynchronously")
        if(error){
            reject();
        }else{
        resolve();

        }
    }
        ,2000);
});

return promise;
}

myAsyncFun().then(
    ()=>console.log("Work Done"),
    ()=>console.log("Error")
    );
```

Passing values

```
let checkEven = new Promise((resolve, reject) => {
    let number = 4;
    if (number % 2 === 0) resolve("The number is even!");
    else reject("The number is odd!");
});
checkEven
    .then((message) => console.log(message)) // On success
    .catch((error) => console.error(error)); // On failure
```

Promise States

Promise can be in one of three states

- **Pending**: The task is in the initial state.
- **Fulfilled**: The task was completed successfully, and the result is available.
- **Rejected**: The task failed, and an error is provided.

Advanced Promise Methods

- Promise.all() Method
- Promise.allSettled() Method
- Promise.race() Method
- Promise.any() Method
- Promise.resolve() Method
- Promise.reject() Method
- Promise.finally() Method

Promise.all() Method

Waits for all promises to resolve and returns their results as an array. If any promise is rejected, it immediately rejects.

```
Promise.all([
    Promise.resolve("Task 1 completed"),
    Promise.resolve("Task 2 completed"),
    Promise.reject("Task 3 failed")
])
    .then((results) => console.log(results))
    .catch((error) => console.error(error));
```

Promise.allSettled() Method

Waits for all promises to settle (fulfilled or rejected) Method and returns an array of their outcomes.

```
Promise.allSettled([
    Promise.resolve("Task 1 completed"),
    Promise.reject("Task 2 failed"),
    Promise.resolve("Task 3 completed")
])
    .then((results) => console.log(results));
```

Promise.race() Method

Promise.race() Method resolves or rejects as soon as the first promise settles.

```
Promise.race([
    new Promise((resolve) =>
        setTimeout(() =>
            resolve("Task 1 finished"), 1000)),
    new Promise((resolve) =>
        setTimeout(() =>
            resolve("Task 2 finished"), 500)),
]).then((result) =>
    console.log(result));
```

Promise.any() Method

Promise.any() Method resolves with the first fulfilled promise. If all are rejected, it rejects with an AggregateError.

```
Promise.any([
    Promise.reject("Task 1 failed"),
    Promise.resolve("Task 4 completed"),
    Promise.resolve("Task 3 completed")
])
    .then((result) => console.log(result))
    .catch((error) => console.error(error));
```

Promise.finally() Method

Promise.finally() Method runs a cleanup or final code block regardless of the promise's result (fulfilled or rejected).

```
Promise.resolve("Task completed")
    .then((result) => console.log(result))
    .catch((error) => console.error(error))
    .finally(() => console.log("Cleanup completed"));
```

Regular expression

Introduction

A regular expression is a special sequence of characters that defines a search pattern, typically used for pattern matching within text.

It’s often used for tasks such as validating email addresses, phone numbers, or checking if a string contains certain patterns (like dates, specific words, etc.).

```
const pattern = /pusp/i;
const name = "pusp joshi";
const address = "dhangadhi";

console.log(pattern.test(name));
console.log(pattern.test(address));
```

Creating a RegExp in JavaScript

There are two primary ways to create a RegExp in JavaScript

1) Using the RegExp Constructor
2) Using Regular Expression Literal

Using constructor

```
let pattern = "hello";  // Pattern to match
let flags = "i";  // Case-insensitive flag
let regex = new RegExp(pattern, flags);

let s = "Hello world";

console.log(regex.test(s));
```

Using Regular expression literal

```
let regex = /hello/i;

let s = "Hello world";

console.log(regex.test(s));
```

Expression Modifiers

Expression	Description
g	Find character globally
i	Case insensitive matching
m	Multiline matching
[abc]	Find any of the characters inside the brackets
[^abc]	Find any character, not inside the brackets
[0-9]	Find any of the digits between the brackets 0 to 9
[^0-9]	Find any digit not in between the brackets
(x|y)	Find any of the alternatives between x or y separated with |

Validating Email address

```
let regex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
console.log(regex.test("user@example.com"));
```

Async/Await

Async

Async and Await in JavaScript is used to simplify handling asynchronous operations using promises.

By enabling asynchronous code to appear synchronous, they enhance code readability and make it easier to manage complex asynchronous flows.

Async

```
<script>

    async function testFunction() {
        return "Hello";
    }
    // Or
    function testFunction() {
       return Promise.resolve("Hello");
    }
    //Above two functions are identical

    testFunction().then(
        (value) =>{ console.log("here"); },
        (error) =>{ console.log("there"); }
    );
</script>
```

Await

The await keyword can only be used inside an **async** function.

The await keyword makes the function pause the execution and wait for a resolved promise before it continues:

Advantages

Improved Readability: Async and Await allow asynchronous code to be written in a synchronous style, making it easier to read and understand.

Error Handling: Using try/catch blocks with async/await simplifies error handling.

Avoids Callback Hell: Async and Await prevent nested callbacks and complex promise chains, making the code more linear and readable.

Better Debugging: Debugging async/await code is more intuitive since it behaves similarly to synchronous code.

Fetch api

Introduction

The fetch() method in JavaScript is a modern and powerful way to retrieve resources from a server. It returns a Promise that resolves to a Response object, encapsulating the server’s response to your request.

Fetch API comes with the fetch() method, which is used to fetch data from different sources.

Syntax

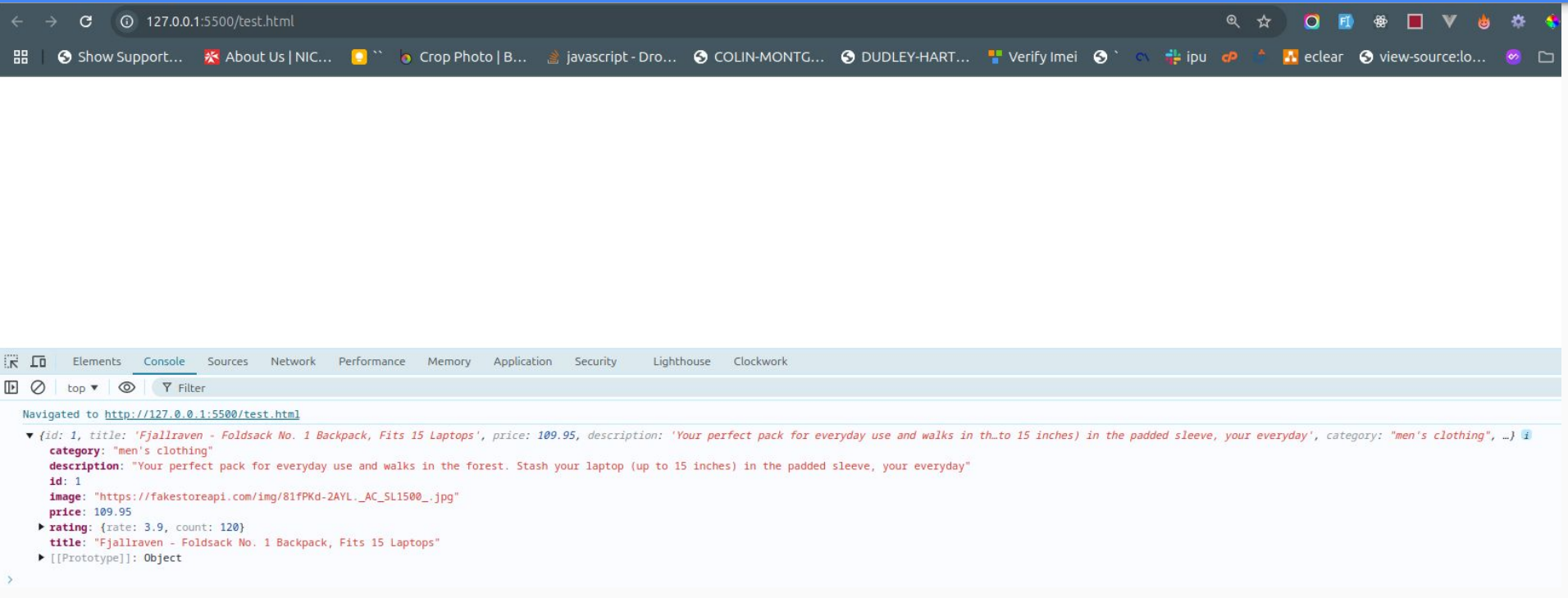
```
fetch('url')
    .then(response => response.json())
    .then(data => console.log(data));
```

Fetch from fake API

```
function fetchData() {
    fetch('https://fakestoreapi.com/products/1')
    .then(res=>res.json())
    .then(json=>console.log(json))

}
fetchData();
```

Fetch from fake API



Get request using fetch()

```
function fetchData() {
    fetch('https://fakestoreapi.com/products')
    .then(res=>res.json())
    .then(json=>console.log(json))

}
fetchData();
```

Post request using fetch()

```
<script>
    const jsonData = { key1: 'value1', key2: 'value2' };

    const options = {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(jsonData)
    };

    fetch('https://api.example.com/upload', options)
    .then(response => {
        if (!response.ok) {
        throw new Error('Network response was not ok');
        }
        return response.json();
    })
    .then(data => {
        console.log(data);
    })
    .catch(error => {
        console.error('Fetch error:', error);
    });

</script>
```

Error handling

Error handling

JavaScript uses throw to create custom errors and try…catch to handle them, preventing the program from crashing.

The finally block ensures that code runs after error handling, regardless of success or failure.

throw: Used to create custom errors and stop code execution.
try…catch: Allows you to catch and handle errors, preventing the program from crashing.
try block: Contains code that may throw an error.
catch block: Catches and handles the error.
finally: Executes code after the try and catch blocks, regardless of an error occurring.
Custom Errors: You can create your own error types by extending the Error class.

Syntax

```
try {
    console.log('try');
} catch (e) {
    console.log('catch');
} finally {
    console.log('finally');
}
```

Introduction to JSON

JSON

- JSON stands for JavaScript Object Notation.
- It is a format for structuring data.
- JSON is the replacement of the XML data exchange format in JSON.
- It is easy to struct the data compare to XML.
- It supports data structures like arrays and objects and the JSON documents that are rapidly executed on the server.
- It is also a Language-Independent format that is derived from JavaScript.
- The official media type for the JSON is application/json and to save those file .json extension.

Features of JSON

- Easy to understand: JSON is easy to read and write.
- Format: It is a text-based interchange format. It can store any kind of data in an array of video, audio, and image anything that you required.
- Support: It is light-weighted and supported by almost every language and OS. It has a wide range of support for the browsers approx each browser supported by JSON.
- Dependency: It is an Independent language that is text-based. It is much faster compared to other text-based structured data.

Simple json structure of file.

```
{
        "Courses": [
        {
                "Name" : "BCA",
                "fee" : "500000",
                "colleges" : [ "Nast", "La-grandi","NCIT", "Oxford" ]
        },
        {
                 "Name" : "BE-Computer",
                 "fee" : "600000",
                "Colleges" : [ "NCIT", "Central college", "Nast" ]
         }
        ]

}
```

Introduction to AJAX

Introduction

Asynchronous JavaScript And XML.

AJAX is not a programming language.

AJAX just uses a combination of:

- A browser built-in XMLHttpRequest object (to request data from a web server)
- JavaScript and HTML DOM (to display or use the data)

Working mechanism

- An event occurs in a web page (the page is loaded, a button is clicked)
- An XMLHttpRequest object is created by JavaScript
- The XMLHttpRequest object sends a request to a web server
- The server processes the request
- The server sends a response back to the web page
- The response is read by JavaScript
- Proper action (like page update) is performed by JavaScript

How to make ajax call from JavaScript ?

Making an Ajax call from JavaScript means sending an asynchronous request to a server to fetch or send data without reloading the web page. This allows dynamic content updates, enhancing user experience by making the web application more interactive and responsive.

1: Using the XMLHttpRequest object

In this approach, we will use the XMLHttpRequest object to make an Ajax call. The XMLHttpRequest() method creates an XMLHttpRequest object which is used to make a request with the server.

Syntax:

let xhttp = new XMLHttpRequest();

How to make ajax call from JavaScript ?

2: Using jQuery

In this approach, we will use jQuery to make an Ajax call. The ajax() method is used in jQuery to make ajax calls. It is used as a replacement for all approaches which are not working to make ajax calls.

3: Fetch Api

In this approach, we will use fetch() API which is used to make XMLHttpRequest with the server. Because of its flexible structure, it is easy to use. This API makes a request to the server and gets the result as a promise which is resolved to the string.

syntax:
fetch(url, {config}).then().catch();