Error Handling

Error handling is crucial for building robust and fault-tolerant JavaScript applications. It ensures that unexpected issues, such as invalid input or network failures, don't crash your program and provides a way to gracefully recover from errors.

try...catch Statement

```
try {
    // Code that might throw an error
    let result = riskyOperation();
    console.log(result);
} catch (error) {
    // Code to handle the error
    console.error("An error occurred:", error.message);
}
```

Throwing Errors

```
function divide(a, b) {
    if (b === 0) {
        throw new Error("Division by zero is not allowed");
    }
    return a / b;
}

try {
    let result = divide(10, 0);
} catch (error) {
    console.error(error.message);  // Outputs: "Division by zero is not allowed"
}
```

finally Block

```
try {
    // Code that may throw an error
    let result = riskyOperation();
} catch (error) {
    console.error("Error occurred:", error.message);
} finally {
    console.log("This will always run, whether an error occurred or not.");
}
```

Error Object

- **message:** A human-readable description of the error.
- **name:** The name/type of the error (e.g., Error, TypeError).
- **stack:** A stack trace showing where the error occurred in the code.

```
try {
    let num = x / y;  // ReferenceError: x is not defined
} catch (error) {
    console.log(error.name);      // Outputs: "ReferenceError"
    console.log(error.message);  // Outputs: "x is not defined"
    console.log(error.stack);    // Outputs: stack trace
}
```

Custom Errors
```
class ValidationError extends Error {
    constructor(message) {
        super(message);  // Call the parent class (Error) constructor
        this.name = "ValidationError";
    }
}

try {
    throw new ValidationError("Invalid input detected");
} catch (error) {
    console.error(`${error.name}: ${error.message}`);
// Outputs: "ValidationError: Invalid input detected"
}
```

Promise Error Handling
```
fetch("https://api.example.com/data")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error("Fetch error:", error.message));
```

Async/Await Error Handling
```
async function fetchData() {
    try {
        const response = await fetch("https://api.example.com/data");
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Error fetching data:", error.message);
    }
}
```

(Object-Oriented Programming)

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to represent data and methods. In JavaScript, OOP enables you to create reusable and organized code.

Objects

```
const car = {
    make: 'Toyota',
    model: 'Corolla',
    year: 2020,
    start: function() {
        console.log('Car started');
    }
};

// Accessing object properties
console.log(car.make); // Outputs: Toyota
car.start(); // Outputs: Car started
```

Constructor Functions

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        console.log(`Hello, my name is ${this.name}`);
    };
}

// Creating instances
const person1 = new Person('Alice', 30);
const person2 = new Person('Bob', 25);

person1.greet(); // Outputs: Hello, my name is Alice
```

Classes (ES6)

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}

const dog = new Dog('Rex');
dog.speak(); // Outputs: Rex barks.
```

Inheritance

```
class Vehicle {
    constructor(brand) {
        this.brand = brand;
    }

    honk() {
        console.log('Beep beep!');
    }
}

class Car extends Vehicle {
    constructor(brand, model) {
        super(brand); // Call the parent constructor
        this.model = model;
    }

    info() {
        console.log(`This car is a ${this.brand}
${this.model}.`);
    }
}

const myCar = new Car('Honda', 'Civic');
myCar.honk(); // Outputs: Beep beep!
myCar.info(); // Outputs: This car is a Honda Civic.
```

Encapsulation

```
function createCounter() {
    let count = 0;

    return {
        increment: function() {
            count++;
            console.log(count);
        },
        getCount: function() {
            return count;
        }
    };
}

const counter = createCounter();
counter.increment(); // Outputs: 1
console.log(counter.getCount());
// Outputs: 1
```

Polymorphism

```
class Cat extends Animal {
    speak() {
        console.log(`${this.name}
meows.`);
    }
}

const animals = [new Dog('Rex'), new
Cat('Whiskers')];

animals.forEach(animal => {
    animal.speak();
// Outputs: Rex barks. and Whiskers
meows.
});
```

Timers

JavaScript timers allow you to execute code after a specific time interval or repeatedly at regular intervals. They are often used for animations, delaying actions, or scheduling repeated actions.

setTimeout()

```
setTimeout(() => {
    console.log("Hello, after 2 seconds!");
}, 2000);  // Executes after 2000ms (2 seconds)
```

setInterval()

```
setInterval(() => {
    console.log("Repeats every 1 second!");
}, 1000);  // Repeats every 1000ms (1 second)
```

Clearing Timers

- **clearTimeout():** Stops the function scheduled by setTimeout().
- **clearInterval():** Stops the function set by setInterval().

```
let timeoutId = setTimeout(() => {
    console.log("This won't run!");
}, 3000);

clearTimeout(timeoutId);  // Cancels the timeout before it can execute

let intervalId = setInterval(() => {
    console.log("Repeating...");
}, 1000);

clearInterval(intervalId);  // Stops the repeating action
```

Regular Expressions

Regular Expressions (RegEx) are patterns used to match character combinations in strings. In JavaScript, they are used for string searching and manipulation. You can create regular expressions using literal notation or the **RegExp** constructor.

1. Creating a Regular Expression

- **Literal Notation:** Uses slashes (/pattern/).
- **RegExp Constructor:** Uses new RegExp('pattern').

```
let regexLiteral = /hello/;
let regexConstructor = new
RegExp('hello');
```

2. Common Methods with Regular Expressions

- **test():** Tests whether a pattern exists in a string. Returns true or false.

```
let regexLiteral = /hello/;
let regexConstructor = new
RegExp('hello');
```

- **exec():** Executes a search for a match in a string. Returns an array of results or null if no match is found.

```
let pattern = /world/;
let result = pattern.exec("Hello
world!");
console.log(result);
// Output: ["world", index: 6, input:
"Hello world!", groups: undefined]
```

match(): Returns an array of all matches or null if no match is found. Works with the string's match() method.

```
let text = "hello hello";
let matches = text.match(/hello/g);
// Output: ["hello", "hello"]
```

3. match(): Returns an array of all matches or null if no match is found. Works with the string's match() method.

```
let text = "hello hello";
let matches = text.match(/hello/g);
// Output: ["hello", "hello"]
```

4. replace(): Replaces a matched pattern in a string.

```
let text = "foo bar";
let result = text.replace(/foo/,
"baz");
// Output: "baz bar"
```

5. search(): Searches for a match and returns the index of the first match, or -1 if not found.

```
let text = "JavaScript is fun";
let index = text.search(/fun/);
// Output: 14
```

6. split(): Splits a string into an array of substrings based on a pattern.

```
let text = "apple, banana, cherry";
let fruits = text.split(/,\s*/);
// Output: ["apple", "banana", "cherry"]
```

3. Common Patterns

.: Matches any single character except a newline.

```
/h.llo/.test("hello");   // true
```

^: Matches the beginning of a string.

```
/^hello/.test("hello world");   // true
```

$: Matches the end of a string.

```
/world$/.test("hello world");   // true
```

***:** Matches 0 or more of the preceding element.

```
/ho*/.test("hooo");   // true
```

+: Matches 1 or more of the preceding element.

```
/ho+/.test("hooo");   // true
```

[]: Matches any one of the characters inside the brackets (character class).

```
/[aeiou]/.test("hello");   // true
```

\d: Matches any digit (0-9).

```
/\d/.test("abc123");   // true
```

\w: Matches any alphanumeric character (letters, digits, or underscore).

```
/\w/.test("hello_123");   // true
```

Flags

- **g:** Global search (find all matches, not just the first).
- **i:** Case-insensitive search.
- **m:** Multiline search.

Local Storage & Session Storage

Local Storage and Session Storage are web storage objects that allow you to store key-value pairs in the browser. They provide a way to persist data across browser sessions (for local storage) or just within a session (for session storage).

Local Storage

- Data stored in Local Storage persists even after the browser is closed and reopened.
- **Storage limit:** typically around 5-10 MB per domain.

```
localStorage.setItem('name', 'John');
```

localStorage.getItem(key): Retrieves the value of the specified key.

```
let name =
localStorage.getItem('name');
// "John"
```

localStorage.removeItem(key): Removes the specified key and its value.

```
localStorage.removeItem('name');
```

localStorage.clear(): Clears all keys and values in local storage.

```
localStorage.clear();
```

Example:

```
// Storing data
localStorage.setItem('username',
'alice');

// Retrieving data
let user =
localStorage.getItem('username');
// "alice"

// Removing data
localStorage.removeItem('username');

// Clearing all local storage data
localStorage.clear();
```

Session Storage

- Data stored in Session Storage persists only until the browser (or tab) is closed.
- **Storage limit:** typically similar to local storage, but session-based.

Basic Methods

sessionStorage.setItem(key, value): Stores a key-value pair.

```
sessionStorage.setItem('sessionUser',
'John');
```

sessionStorage.getItem(key): Retrieves the value of the specified key.

```
sessionStorage.removeItem('sessionUser'
);
```

sessionStorage.clear(): Clears all keys and values in session storage.

```
sessionStorage.clear();
```

Example:

```
// Storing session data
sessionStorage.setItem('sessionToken',
'abc123');

// Retrieving session data
let token =
sessionStorage.getItem('sessionToken');
// "abc123"

// Removing session data
sessionStorage.removeItem('sessionToken
');

// Clearing all session storage data
sessionStorage.clear();
```