1. ***try, catch and finally used for error handling***

when we know there can be runtime error in the program at that time we use these error handling methods.

***Try*** :- If any error occurs in within this try block then javascript stops running or executing the try block and to catch block

***Catch*** :- The catch block is where you handle any errors that occur in the try block. It contains the code that handles the error you can access the error object as an argument in the catch block

***Finally*** :- this is an optional block, this contains the code that will execute any how in any situation whether there is an error or not.

try {

let result = riskyOperation();

console.log("Operation successful:", result);

} catch (error) {

console.error("An error occurred:", error.message);

} finally {

console.log("Cleanup code executed.");

}

**OUTPUT**:- An error occurred: riskyOperation is not defined

Cleanup code executed.

**EXPLANATION OF THE ABOVE EXAMPLE**

Here the function riskyOperation() is not defined that shows an error and the catch and finally block has been executed with the error message.

1. ***Lexial scope with example***

It is a concept in programming languages where the scope of variable is determined by its position in source code. In simple words visibility of variables and functions is based on their location within the nested structure of code.

Here the nested functions have access to variables defined in outer (enclosing) scopes this relation is established at the compile time and remains constant throughout the execution of the code.

Function outer() {

Let outervariable = “Hey Guys”;

Function inner() {

Let innervariable= “tata dosto”;

Console.log(outervariable);

}

Inner();

}

Outer();

**OUTPUT:- Hey Guys**

*EXPLANATION FOR THE LEXICAL SCOPE EXAMPLE..*

**Defining `outer()` Function**:- here the `outer` function is defined, it creates a variable name `outerVariable` with a value "Hey Guys".

**Defining `inner()` Function:-** Inside `outer`, another function call `inner` is defined this function has its own local variable `innervariable` with a value "tata dosto".

**Accessing `outerVariable`:-** The `inner` function is called within `outer`. At this point, the code inside `inner` executes and prints `outerVariable` to the console. The output will be "Hey Guys".

1. ***Closures***

A closure in JavaScript is when a function retains access to its outer (lexical) environment, even after the outer function has finished executing. This allows the inner function to remember and use variables from its outer scope.

function outerFunction() {

let outerVariable = 'I am outside!';

function innerFunction() {

console.log(outerVariable);

}

return innerFunction;

}

const myClosure = outerFunction();

myClosure();

***EXPLANATION:-***

1. `outerFunction` is called and executed.
2. Inside `outerFunction`, there's a variable `outerVariable` and a function `innerFunction`.
3. `innerFunction` is returned from `outerFunction` and assigned to `myClosure`.
4. Even though outerFunction has finished executing, `myClosure` (which is `innerFunction`) still remembers outerVariable because of the closure.

This is a closure: `innerFunction` keeps access to `outerVariable` even after `outerFunction` is done running.

1. ***Callback***

Callback is function that is passed as an argument to another function and is executed after some operation has been completed.

Callback is used to handle asynchronous operation like fetching from the server or reading files.

***KEY POINTS***

***Synchronous Callbacks***: Executed immediately after the function they're passed into completes.

***Asynchronous Callbacks***: Executed after a certain operation (like a timer, API call, or file reading) completes.

***Synchronous Callbacks example:-***

function calculate(a, b, callback) {

let result = a + b;

callback(result);

}

function displayResult(result) {

console.log(`The result is ${result}`);

}

// Using the functions

calculate(5, 3, displayResult);

**EXPLANATION FOR SYNCHRONOUS CALLBACK:-**

* ***calculate Function***: Takes two numbers (a and b) and a `callback` function as arguments. It adds the two numbers together and then calls the `callback` function, passing the result as an argument.
* ***displayResult Function***: A simple function that logs the result to the console.
* ***Execution***:
  + When calculate(5, 3, displayResult) is called, it adds 5 and 3 to get 8.
  + Then, it immediately calls displayResult(8), which logs "The result is 8" to the console.

***Asynchronous Callbacks example:-***

function fetchData(callback) {

console.log("Fetching data, please wait...");

setTimeout(() => {

let data = {

id: 1,

name: "Arman Shaikh",

email: "armanshaikh@gmail.com"

};

callback(data);

}, 2000);

}

function processData(data) {

console.log(`Data received: ${data.name}, ${data.email}`);

}

fetchData(processData);

***Explanation:***

1. fetchData Function: Simulates fetching data asynchronously. It takes a `callback` function as an argument.
2. `setTimeout` is used to simulate a 2-second delay before the data is "fetched."
3. After 2 seconds, the `callback` function is called with the fetched data as *an* argument.
4. processData Function: Logs the fetched data to the console.
5. Execution:
6. When `fetchData(*processData*)` is called, it first logs "Fetching data, please wait...".
7. After 2 seconds, the data is "fetched," and `processData` is called with the data, logging "Data received: Arman Shaikh, armanshaikh@gmail.com" to the console.

***5) PROMISE***

A Promise in JavaScript is like a promise you make in real life. It's a way to handle things that will happen in the future.

**Promises** are a modern way to handle asynchronous operations in JavaScript, making the code more readable and easier to manage compared to callback-based approaches.

They allow you to define what happens when an operation is successful (`then`), when it fails (`catch`), and what should happen regardless of the outcome (`finally`).

METHODS OD PROMISE:-

* then()
* catch()
* finally()

EXAMPLE OF PROMISE :-

function userAuthentication(username, password) {

return new Promise((resolve, reject) => {

console.log("Checking user Authentication...");

setTimeout(() => {

if (username === "Arman" && password === "password") {

resolve("user Authentication successful");

} else {

reject("user Authentication failed");

}

}, 3000);

});

}

userAuthentication("Arman", "password")

.then((message) => {

console.log("user Authentication successful");

})

.catch(() => {

console.log("user Authentication failed");

})

.finally(() => {

setTimeout(() => {

console.log("Authentication Process Complete");

}, 2000);

});

**OUTPUT:-**

Checking user Authentication...

user Authentication successful

Authentication Process Complete

EXPLANATION FOR THE ABOVE PROMISE EXAMPLE:-

* Here the function is created by the name of `userAuthentication` that takes the parameter `username` and `password`
* Inside the function a new `promise` object is created and this object tells that the task will complete in future also stimulates user authentication and return a promise that resolves or rejects on the credentials
* Inside the Promise:
* Logs "Checking user Authentication..." to indicate the start of the authentication process.
* Uses `setTimeout` to simulate a delay of 3 seconds (3000 *milliseconds*). After the delay:
* If `username` is "Arman" and `password` is "password", the promise is resolved with the message "user Authentication successful".
* If the credentials do not match, the promise is rejected with the message "user Authentication failed".
* Using `userAuthentication`(“Arman”, “password”)

This initiates the authentication process with the give credentials. The function returns a promise.

**.then((message) => { ... }):**

* This is called if the promise is **resolved** (i.e., authentication is successful).
* It logs "user Authentication successful". Note that the message parameter, which contains "user Authentication successful", is not used here, but could be if needed.

**.catch(() => { ... }):**

* This is called if the promise is **rejected** (i.e., authentication fails).
* It logs "user Authentication failed". No parameters are used in the catch block, but you could modify it to accept and use an error message if needed.

**.finally(() => { ... }):**

* This is called after the promise has been settled (either resolved or rejected), regardless of the outcome.
* Inside the finally block, there is a setTimeout with a 2-second delay (2000 milliseconds) before logging "Authentication Process Complete". This ensures that the completion message is logged after the final delay.

***6) async/await***

**Async**: A keyword used to declare a function as asynchronous. An `async` function always returns a promise, which resolves with the value returned from the function or rejects if an error is thrown.

**Await**: A keyword used inside `async` functions to pause the execution of the function until a promise is resolved or rejected. It allows you to write code that waits for asynchronous operations to complete without blocking the main thread.

**EXAMPLE OF async/await**:-

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Data fetched");

}, 2000);

});

}

async function fetchDataAndLog() {

try {

let data = await fetchData(); // Waits for fetchData to resolve

console.log(data); // Logs the resolved data

} catch (error) {

console.error("Error:", error); // Catches and logs any error

}

}

fetchDataAndLog();

output:- Data fetched

EXPLANATION FOR async/await EXAMPLE…

* **`fetchData`**: A function that returns a promise, simulating a delay of 2 seconds before resolving with the string "Data fetched".
* **`fetchDataAndLog`**: An `async` function that uses `await` to wait for the fetchData promise to resolve.
* **`await fetchData()`**: Pauses the execution of fetchDataAndLog until the promise from fetchData is resolved.
* **`try...catch`**: Handles any errors that may occur during the await operation.