Error Handling

- Error handling refers to the process of catching and managing errors that occur during code execution.
- JavaScript provides built-in mechanisms for capturing and responding to errors to prevent them from crashing the program or creating undesired behaviour.
- In JavaScript, errors can occur due to various reasons like invalid input, network issues, or logical errors in code.
- JavaScript provides a built-in mechanism for error handling using the try...catch block.

Key Concepts of Error Handling

a) **Errors:** Errors in JavaScript typically occur when the program encounters something unexpected. Common types of errors include:

- **Syntax Errors:** Errors in the syntax of the code, such as missing brackets or misspelled keywords.

```
function sayHello() {
console.log("Hello, World!");
}
sayHello;  // SyntaxError: Function statements require a function name
```

- **Reference Errors:** Occur when a variable or object is referenced that does not exist.

```
console.log(x);  // ReferenceError: x is not defined
```

- **Type Errors:** When a value is of an unexpected type, like calling a method on undefined.

```
let num = 10;
num.toUpperCase();  // TypeError: num.toUpperCase is not a function
```

- **Range Errors:** When a value is not within a valid range, like trying to create an array with a negative length.

```
let num = new Array(-1);  // RangeError: Invalid array length
```

b) **Try-Catch-Finally:** JavaScript provides the try-catch-finally construct to handle errors.

- **try block:** Contains code that might throw an error.
- **catch block:** Executes when an error occurs in the try block. It captures the error and prevents the program from crashing.
- **finally block:** Executes regardless of whether an error occurred or not, usualy used for cleanup operations.

Syntax

```
try {
    // Code that might throw an error
} catch (error) {
    // Code to handle the error
    console.log(error.message);
} finally {
    // Code that runs no matter whether error is thrown or not
    console.log("Execution complete.");
}
```

Example

```
try {
    let result = myFunction(); // This will throw an error if there is no function
} catch (error) {
        console.error("An error occurred: ", error.message);
} finally {
        console.log("Clean-up code or finalise the requirements.");
}
```

Example: Handling Division by Zero

```
function divide(firstVal, secondVal) {
  try {
    if (secondVal === 0) {
      throw new Error("Division by zero is not allowed.");
    }
    console.log("Result:", firstVal / secondVal);
  } catch (error) {
    console.error("Error:", error.message);
  } finally {
    console.log("Division operation completed.");
  }
}
divide(10, 2);  // Output: Result: 5
divide(10, 0);  // Output: Error: Division by zero is not allowed.
```

Example: Handling Invalid JSON Parsing

```
const jsonString = '{ "name": "Nisha", "age": 24 }';  // Correct JSON
const badJsonString = '{ name: Karan, age: 19 }';    // Invalid JSON (missing quotes)

function parseJSON(jsonStr) {
  try {
    const obj = JSON.parse(jsonStr);
    console.log("Parsed JSON:", obj);
  } catch (error) {
    console.error("Invalid JSON:", error.message);
  } finally {
    console.log("JSON parsing attempt completed.");
  }
}
parseJSON(jsonString);     // Correct JSON parsing
parseJSON(badJsonString);  // Output: Invalid JSON
```

Throwing Errors: You can also create your own errors using the throw statement. This is helpful for handling custom error conditions.

Example

```
function calculate(a, b) {
    if (b === 0) {
        throw new Error("Division by zero is not allowed.");
    }
    return a / b;
}
try {
            let result = calculate (10, 0);
} catch (error) {
            console.error(error.message);
}
```

Example: File Handling Simulation (Using Throw)

```
function readFile(filename) {
  try {
    if (filename === "") {
      throw new Error("Filename cannot be empty.");
    }
    // Simulate file reading
    console.log(`Reading file: ${filename}`);
  } catch (error) {
    console.error("File Error:", error.message);
  } finally {
    console.log("File operation completed.");
  }
}
readFile("data.txt");  // Simulated successful file reading
readFile("");          // Output: File Error: Filename cannot be empty.
```

Example: Handling Array Index Out of Bounds

```
const arr = [1, 2, 3];
function accessArrayElement(index) {
  try {
    if (index >= arr.length) {
      throw new RangeError("Index is out of bounds.");
    }
    console.log(`Element at index ${index}:`, arr[index]);
  } catch (error) {
    console.error(error.name + ":", error.message);
  } finally {
    console.log("Array access completed.");
  }
}
accessArrayElement(1);  // Output: Element at index 1: 2
accessArrayElement(5);  // Output: RangeError: Index is out of bounds.
```

Example: Handling Invalid Function Arguments

```
function calculateSquareRoot(number) {
  try {
    if (number < 0) {
      throw new RangeError("Square root of negative numbers is not allowed.");
    }
    console.log("Square root:", Math.sqrt(number));
  } catch (error) {
    console.error(error.name + ":", error.message);
  } finally {
    console.log("Square root calculation completed.");
  }
}

calculateSquareRoot(16);   // Output: Square root: 4
calculateSquareRoot(-4);  // Output: RangeError: Square root of negative numbers
                            is not allowed.
```

Custom Errors: JavaScript allows creating custom error classes to differentiate different types of application-specific errors.

Example

```
class CustomError extends Error {
    constructor(message) {
                                    super(message);
                                    this.name = "Here is the CustomError";
    }
}

try {
        throw new CustomError("This is a custom error.");
} catch (error) {
        console.error(`${error.name}: ${error.message}`);
}
```

Exception Handling

- Exception handling in JavaScript is essentially the same as error handling.
- In JavaScript, errors (both built-in and custom) are treated as exceptions, which can be "thrown" using the throw keyword and "caught" using the try...catch mechanism.

Types of Exceptions in JavaScript

- **System-defined exceptions:** These are built-in errors like ReferenceError, TypeError, SyntaxError, etc.
- **User-defined exceptions:** You can create custom exceptions using the throw statement.

Example of Exception Handling

```
function checkAge(age) {
  try {
    if (typeof age !== "number") {
                throw new TypeError("Age must be a number.");
    }
    if (age < 18) {
                throw new RangeError("You must be at least 18 years old.");
    }
                console.log("Age is valid:", age);
  } catch (exception) {
    if (exception instanceof TypeError) {
                console.error("Type Error:", exception.message);
    } else if (exception instanceof RangeError) {
                console.error("Range Error:", exception.message);
    } else {
                console.error("Unknown Error:", exception.message);
    }
  }
}
checkAge("twenty");  // Throws TypeError
checkAge(15);        // Throws RangeError
checkAge(21);        // Valid age
```

Throwing Custom Exceptions: You can create and throw your own custom errors using the throw statement, and these will be handled in the same way as built-in errors.

Example

```
function withdraw(amount) {
  const balance = 500;
  try {
    if (amount > balance) {
      throw {
        name: "InsufficientFundsError",
        message: `Requested amount exceeds available balance. Current balance is
${balance}.`,
      };
    }
    console.log(`Withdrawal successful. You withdrew ${amount}`);
  } catch (error) {
    console.error(`${error.name}: ${error.message}`);
  }
}
withdraw(600);  // Custom error for insufficient funds
```

Why JavaScript Doesn't Use throws

- JavaScript only has unchecked exceptions, meaning there's no need to declare exceptions a function might throw.
- JavaScript's error handling is dynamic and flexible, allowing errors to be handled at runtime.
- Adding throws would impose unnecessary static constraints on a language that is designed to be more fluid and dynamic.
- JavaScript allows developers to handle exceptions through the try-catch mechanism without the need for function declarations regarding exceptions.

JavaScript uses unchecked exceptions, meaning that the language does not enforce explicit declaration of which exceptions a function might throw. You can throw exceptions using the throw statement, but you are not required to declare them in the function signature, unlike in languages like Java that use the throws keyword.

Difference Between Error Handling and Exception Handling

The terms "error handling" and "exception handling" are often used interchangeably, but there is a subtle difference:

Error Handling

- Errors are typically used to describe unexpected issues in the code itself, such as syntax errors, type errors, or reference errors. These errors often indicate bugs or problems with the code.
- Errors can be recoverable (i.e., the program can continue running by handling them appropriately) or non-recoverable (i.e., they cause the program to crash or stop execution).
  Example: Catching an error when a file is not found or when the user input is invalid.

Exception Handling

- Exceptions are specific error-like conditions that are intentionally thrown to signal that something unusual happened. The term "exception" usually implies that the code is designed to handle this situation and recover.
- Exception handling is more of a structured mechanism to deal with specific exceptional cases, such as custom application logic that detects and throws an exception.
- Exceptions in JavaScript are thrown using the throw statement and can be caught and handled in a try-catch block.
  Example: Throwing an exception when the result of a function is invalid for business logic (e.g., throwing an exception when a user tries to withdraw more money than available in their account).