**Tasks:**

**1.  JS-ERROR-T-ERROR-VS-IF-1 (1 point)**

In your operations logic, when should you use if statements and when error handling? Give a justified explanation.

**1. If Statements:**

When to use:

Scenario: I want a program to make decisions based on certain conditions.

Example: If it's raining, take an umbrella; otherwise, go out without it.

Explanation: If statements help a program choose between different actions based on whether a condition is true or false.

if (isRaining) {

takeUmbrella();

} else {

goOutWithoutUmbrella();

}

**2. Error Handling:**

When to use:

Scenario: You want to prepare your program for unexpected problems.

Example: You're baking cookies, and the recipe says to use 2 eggs. What if you only have 1 egg?

Explanation: Error handling helps your program deal with unexpected situations, like when something goes wrong or isn't what you expect.

try {

bakeCookies();

} catch (error) {

handleMissingIngredients(error);

}

**Conclusion:**

Use If Statements when you want your program to make decisions based on conditions.

Use Error Handling when you want your program to be ready for unexpected problems.

In simple terms, If Statements help your program decide what to do in different situations, while Error Handling helps it deal with problems that might pop up unexpectedly. Think of If Statements as making choices and Error Handling as preparing for the unexpected.

**2. JS-ERROR-P-LEAPYEAR-1 (2 points)**

In the practical exercise 2 the task 1 (JS-FUNCTION-P-LEAPYEAR-1) was to program a function isLeapYear. The function accepted one argument year.

Develop this function further so that it can throw two different kinds of errors when called incorrectly. If the function is called without a parameter, then an error with a message “Missing argument year error” is thrown. If the function is called with non-integer argument, then an error with a message “Non-integer argument year error” is thrown. In this case, write the try-catch statement inside the function and output the error message on the console.

Please note that an argument value 0 should not be considered as an error.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Leap Year Checker</title>

</head>

<body>

    <label for="yearInput">Enter a year:</label>

    <input type="text" id="yearInput" placeholder="Enter a year">

    <button onclick="checkLeapYear()">Check year</button>

    <div id="result"></div>

    <script>

        function isLeapYear(year) {

            try {

                // Check if the year is not provided

                if (year === undefined) {

                    throw new Error("Missing argument year error");

                }

                // Check if the year is not an integer

                if (!Number.isInteger(year) && year !== 0) {

                    throw new Error("Non-integer argument year error");

                }

                // Leap year conditions

                return (year % 4 === 0 && (year % 100 !== 0 || year % 400 === 0));

            } catch (error) {

                // Output the error message to the console

                console.error(error.message);

            }

        }

        function checkLeapYear() {

            var yearInput = document.getElementById('yearInput').value;

            var resultDiv = document.getElementById('result');

            if (!isNaN(yearInput) && yearInput !== '') {

                var year = parseInt(yearInput);

                var isLeap = isLeapYear(year);

                resultDiv.innerHTML = "Year " + (isLeap ? "is a leap year." : "is not a leap year.");

            } else {

                resultDiv.innerHTML = "Please enter a valid year.";

            }

        }

    </script>

</body>

</html>

**3. JS-ERROR-P-LEAPYEAR-2 (1 point)**

Return to the task above. Change the implementation so that the try-catch is moved to an enclosing operation tryIsLeapYear. The errors are still thrown inside the isLeapYear operation.  
  
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Leap Year Checker</title>

</head>

<body>

    <label for="yearInput">Enter a year:</label>

    <input type="text" id="yearInput" placeholder="Enter a year">

    <button onclick="tryIsLeapYear()">Check year</button>

    <div id="result"></div>

    <script>

        function isLeapYear(year) {

            // Check if the year is not provided

            if (year === undefined) {

                throw new Error("Missing argument year error");

            }

            // Check if the year is not an integer

            if (!Number.isInteger(year) && year !== 0) {

                throw new Error("Non-integer argument year error");

            }

            // Leap year conditions

            return (year % 4 === 0 && (year % 100 !== 0 || year % 400 === 0));

        }

        function tryIsLeapYear() {

            var yearInput = document.getElementById('yearInput').value;

            var resultDiv = document.getElementById('result');

            try {

                if (!isNaN(yearInput) && yearInput !== '') {

                    var year = parseInt(yearInput);

                    var isLeap = isLeapYear(year);

                    resultDiv.innerHTML = "Year " + (isLeap ? "is a leap year." : "is not a leap year.");

                } else {

                    resultDiv.innerHTML = "Please enter a valid year.";

                }

            } catch (error) {

                console.error(error.message);

            }

        }

    </script>

</body>

</html>

**4. JS-ERROR-P-LEAPYEAR-3 (1 point)**

Return to the task above and change the implementation so that also throwing the error is done in the enclosing operation tryIsLeapYear.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Leap Year Checker</title>

</head>

<body>

    <label for="yearInput">Enter a year:</label>

    <input type="text" id="yearInput" placeholder="Enter a year">

    <button onclick="tryIsLeapYear()">Check year</button>

    <div id="result"></div>

    <script>

        function isLeapYear(year) {

            // Leap year conditions

            return (year % 4 === 0 && (year % 100 !== 0 || year % 400 === 0));

        }

        function tryIsLeapYear() {

            var yearInput = document.getElementById('yearInput').value;

            var resultDiv = document.getElementById('result');

            try {

                // Check if the year is not provided

                if (yearInput === '') {

                    throw new Error("Missing argument year error");

                }

                // Check if the year is not a valid number

                var year = parseInt(yearInput);

                if (isNaN(year)) {

                    throw new Error("Non-integer argument year error");

                }

                var isLeap = isLeapYear(year);

                resultDiv.innerHTML = "Year " + (isLeap ? "is a leap year." : "is not a leap year.");

            } catch (error) {

                console.error(error.message);

                resultDiv.innerHTML = "Error: " + error.message;

            }

        }

    </script>

</body>

</html>

**5.  JS-ERROR-T-ERROR-COMPARE-FUNCTIONS-1 (1 point)**

Compare the ways of handling errors above. What are main differences when writing and using them.

1. function isLeapYear(year) {

    try {

        // Check if the year is not provided

        if (year === undefined) {

            throw new Error("Missing argument year error");

        }

        // Check if the year is not an integer

        if (!Number.isInteger(year) && year !== 0) {

            throw new Error("Non-integer argument year error");

        }

        // Leap year conditions

        return (year % 4 === 0 && (year % 100 !== 0 || year % 400 === 0));

    } catch (error) {

        // Output the error message to the console

        console.error(error.message);

    }

}

2.

function isLeapYear(year) {

    // Leap year conditions

    return (year % 4 === 0 && (year % 100 !== 0 || year % 400 === 0));

}

function tryIsLeapYear() {

    var yearInput = document.getElementById('yearInput').value;

    var resultDiv = document.getElementById('result');

    try {

        if (!isNaN(yearInput) && yearInput !== '') {

            var year = parseInt(yearInput);

            var isLeap = isLeapYear(year);

            resultDiv.innerHTML = "Year " + (isLeap ? "is a leap year." : "is not a leap year.");

        } else {

            resultDiv.innerHTML = "Please enter a valid year.";

        }

    } catch (error) {

        console.error(error.message);

    }

}

Differences:

Error Throwing Location:

Original: The isLeapYear function itself throws errors within its own Logic using throw.

Modified: The tryIsLeapYear function is responsible for validation and error throwing. The isLeapYear function only focuses on the core Logic without error handling.

Error Handling Responsibility:

Original: The isLeapYear function handles its own errors by catching them within its try-catch block. It also logs the error message to the console.

Modified: The tryIsLeapYear function handles errors thrown by isLeapYear. It catches errors and logs the message to the console. It also updates the resultDiv with an error message.

Clarity of Responsibilities:

Original: The responsibility for checking leap year conditions and error handling is mixed in the isLeapYear function.

Modified: The responsibilities are separated more clearly. isLeapYear focuses only on leap year conditions, and tryIsLeapYear handles validation and error handling.

Output to User:

Original: The original implementation logs error messages to the Console but doesn't directly inform the user about errors.

Modified: The modified implementation updates the resultDiv with appropriate messages, both for successful checks and errors, making it more user-friendly.

Both approaches have their merits, and the choice depends on the structure and complexity of your application. The modified approach with a separate error-handling function (tryIsLeapYear) can be more modular and easier to understand, especially as the codebase grows.

**6. JS-REGEX-T-EXPLAIN-1 (2 points)**

Explain shortly:

1. How regular expressions consume their input? (What is the mechanism of matching?)

Mechanism of Matching in Regular Expressions:

Regular expressions in JavaScript use a pattern to match against input strings. The matching mechanism involves traversing the input string and attempting to match the characters based on the specified pattern. The engine tries to find the leftmost, longest match in the input string. It proceeds character by character, attempting to satisfy the pattern requirements.

1. What do parentheses around any part of the regular expression cause?

Parentheses in Regular Expressions:

Parentheses in a regular expression serve two main purposes:

Grouping: Parentheses group expressions together, allowing you to apply quantifiers or other operators to a group of characters. For example, (abc)+ means one or more occurrences of the sequence "abc."

Capturing: Parentheses also create capture groups, which are used to remember parts of the matched text. The content matched by a group can be referenced later in the regex or in the result of a match.

1. What is the difference between lazy matching and greedy matching in regular expressions?

Lazy Matching vs. Greedy Matching:

Greedy Matching: By default, quantifiers (such as \*, +, ?, {}) in regular expressions are greedy, meaning they match as much of the input as possible. For example, in the regex a+, it will match one or more consecutive 'a' characters, trying to maximize the number of 'a's.

Lazy Matching (Non-greedy): Adding a ? after a quantifier makes it lazy, causing it to match as little as possible. For example, in the regex a+?, it will match the smallest possible sequence of 'a' characters.

1. When do you need to use RegExp constructor like new RegExp() instead of regular expression literal like / /?

Using RegExp Constructor vs. Regular Expression Literal:

Regular Expression Literal (e.g., /pattern/): This is a concise and common way to create regular expressions. It is suitable when the pattern is known at the time of writing the code.

RegExp Constructor (e.g., new RegExp("pattern")): Use the constructor when the pattern needs to be dynamically generated at runtime, perhaps based on user input or other variables. The constructor allows you to build a regular expression from a string.

**7. JS-REGEX-P-BUILD-REGISTER-NUMBER-1 (2 points)**

Develop a function buildRegisterNumber so that it can throw two different kinds of errors when called with incorrect argument values. The function takes two arguments: theLetters and theDigits. If the value of theLetters argument is not valid, then an error with a message “Invalid register number letters” is thrown. If the value of theDigits argument is not valid, then an error with a message “Invalid register number digits” is thrown. In case the arguments are valid, then a valid register number is returned.

Use regular expressions to validate the argument values.

Let’s agree that a valid register number obeys the following rules

·        there are from two to three uppercase letters before a dash

·        the letter W is not allowed

·        a dash

·        there are from one to three digits after the dash

·        no leading zeros are allowed (no zeros before first non-zero digit)

Examples of valid register numbers:

AX-12

UUI-6

GFS-200

Examples of invalid register numbers:

X-100

YUT-020

WWW-100

function buildRegisterNumber(theLetters, theDigits) {

    // Define regular expressions for letters and digits validation

    const lettersRegex = /^[A-VX-Z]{2,3}$/; // Two to three uppercase letters excluding 'W'

    const digitsRegex = /^[1-9]\d{0,2}$/;   // One to three digits with no leading zeros

    // Validate theLetters

    if (!lettersRegex.test(theLetters) || theLetters.includes('W')) {

      throw new Error("Invalid register number letters");

    }

    // Validate theDigits

    if (!digitsRegex.test(theDigits)) {

      throw new Error("Invalid register number digits");

    }

    // If both validations pass, construct and return the register number

    return `${theLetters}-${theDigits}`;

  }

  // Examples

  try {

    const validRegisterNumber = buildRegisterNumber("AX", "12");

    console.log(validRegisterNumber); // Output: AX-12

    // Uncomment the following lines to see how errors are thrown for invalid inputs

    // const invalidLetters = buildRegisterNumber("X", "100"); // Throws "Invalid register number letters" error

    // const invalidDigits = buildRegisterNumber("AX", "020");  // Throws "Invalid register number digits" error

  } catch (error) {

    console.error(error.message);

  }