Name: **Dev Gaonkar** Div/Roll no.: **12**

Lab Exercise 5

Title: Assignment on Advanced Javascript

Objectives:

1. Write a javascript code to implement a basic calculator that performs addition, subtraction, multiplication, and division using promises. The calculator should accept two numbers and an operation and return the result as a promise. In case of invalid operation (wrong operator or divide by zero) let it reject with an error message
2. Creating a custom iterator for an array of numbers. The iterator should produce squares of all numbers in the array.
3. Execute generator function that generates prime numbers using JavaScript generators. The generator should produce prime numbers up to a specified limit.

Theory

1. **What is a Promise in JavaScript, and what problem does it solve?**
   A Promise in JavaScript is an object that represents the eventual completion or failure of an asynchronous operation. It provides a way to handle asynchronous tasks like API requests or file operations without resorting to callback functions, which can lead to "callback hell" or difficult-to-maintain code. Promises have three states: *pending*, *fulfilled*, and *rejected*. By using `.then()` and `.catch()`, developers can manage asynchronous results in a more readable and structured way, solving the problem of managing multiple levels of nested callbacks.

2. **What is an iterator in JavaScript, and what is its primary purpose?**
   An iterator in JavaScript is an object that allows sequential access to elements in a collection, such as an array or object. It implements the `next()` method, which returns an object with two properties: `value` (the current element) and `done` (a boolean indicating if iteration is complete). Iterators enable looping over data structures in a controlled manner, especially for custom data types. They are primarily used with constructs like `for...of` loops or within generator functions, enabling cleaner and more modular data traversal.

3. **What is the purpose of the yield keyword in a generator function?**
   The `yield` keyword in a JavaScript generator function pauses the execution of the function, returning a value to the caller while preserving the function's state. When the generator is resumed (using the `next()` method), execution continues from where it left off. This makes `yield` useful for generating sequences of values lazily, handling asynchronous operations, or creating complex iterator logic. The purpose of `yield` is to provide a more flexible, resumable execution model compared to traditional functions, allowing the function to yield control back to the calling code when needed.

Code with Screenshot:

1. **Calculator using Promises**

Code:

```
function calculator(num1, num2, operation) {
  return new Promise((resolve, reject) => {
    if (operation === '/' && num2 === 0) {
      reject("Error: Division by zero is not allowed.");
    } else {
      switch (operation) {
        case '+':
          resolve(num1 + num2);
          break;
        case '-':
          resolve(num1 - num2);
          break;
        case '*':
          resolve(num1 * num2);
          break;
        case '/':
          resolve(num1 / num2);
          break;
        default:
          reject("Error: Invalid operation. Use +, -, *, or /.");
      }
    }
  });
}

function getUserInput() {
  const num1 = parseFloat(prompt("Enter the first number:"));
  const num2 = parseFloat(prompt("Enter the second number:"));
  const operation = prompt("Enter the operation (+, -, *, /):");

  calculator(num1, num2, operation)
    .then(result => console.log(`Result: ${result}`))
    .catch(error => console.log(`Error: ${error}`));
}

getUserInput();
```

Output:

```
node /tmp/SE32wVcu79.js
Enter the first number:12
Enter the second number:0
Enter the operation (+, -, *, /):/
ERROR!
Error: Error: Division by zero is not allowed.
```

2. **Iterator for an Array**

Code:

```
function squareIterator(arr) {
  let index = 0;

  return {
    next() {
      if (index < arr.length) {
        const value = arr[index] * arr[index];
        index++;
        return { value, done: false };
      } else {
        return { done: true };
      }
    }
  };
}

const numbers = [1, 2, 3, 4, 5];

const squares = squareIterator(numbers);

let result = squares.next();
while (!result.done) {
  console.log(result.value);
  result = squares.next();
}
```

Output:

```
node /tmp/zPLloXBncl.js
1
4
9
16
25
```

3. **Generator function for Prime numbers**

Code:

```
function* generatePrimes(limit) {
  function isPrime(num) {
    if (num <= 1) return false;
    for (let i = 2; i <= Math.sqrt(num); i++) {
      if (num % i === 0) return false;
    }
    return true;
  }

  for (let num = 2; num <= limit; num++) {
    if (isPrime(num)) {
      yield num;
    }
  }
}

function getUserInput() {
  const limit = parseInt(prompt("Enter the limit for prime numbers:"));

  if (isNaN(limit) || limit < 2) {
    console.log("Please enter a valid number greater than or equal to 2.");
    return;
  }

  const primeGenerator = generatePrimes(limit);

  console.log(`Prime numbers up to ${limit}:`);
  for (let prime of primeGenerator) {
    console.log(prime);
  }
}

getUserInput();
```

Output:

```
node /tmp/xEbG2gBfjF.js
Enter the limit for prime numbers:12
Prime numbers up to 12:
2
3
5
7
11
```