Assignment 4B

Aim: Write a program on Inheritance, Iterators and Generators.

Theory:

1. Inheritance-

   Inheritance is a fundamental concept in object-oriented programming that allows you to create a new class (sub-class or derived class) that inherits properties and behaviours (methods) from an existing class (superclass or base class). This promotes code reusability and hierarchy in your codebase.

   In JavaScript, inheritance is achieved through the **class** keyword and the **extends** keyword.

2. Iterators-

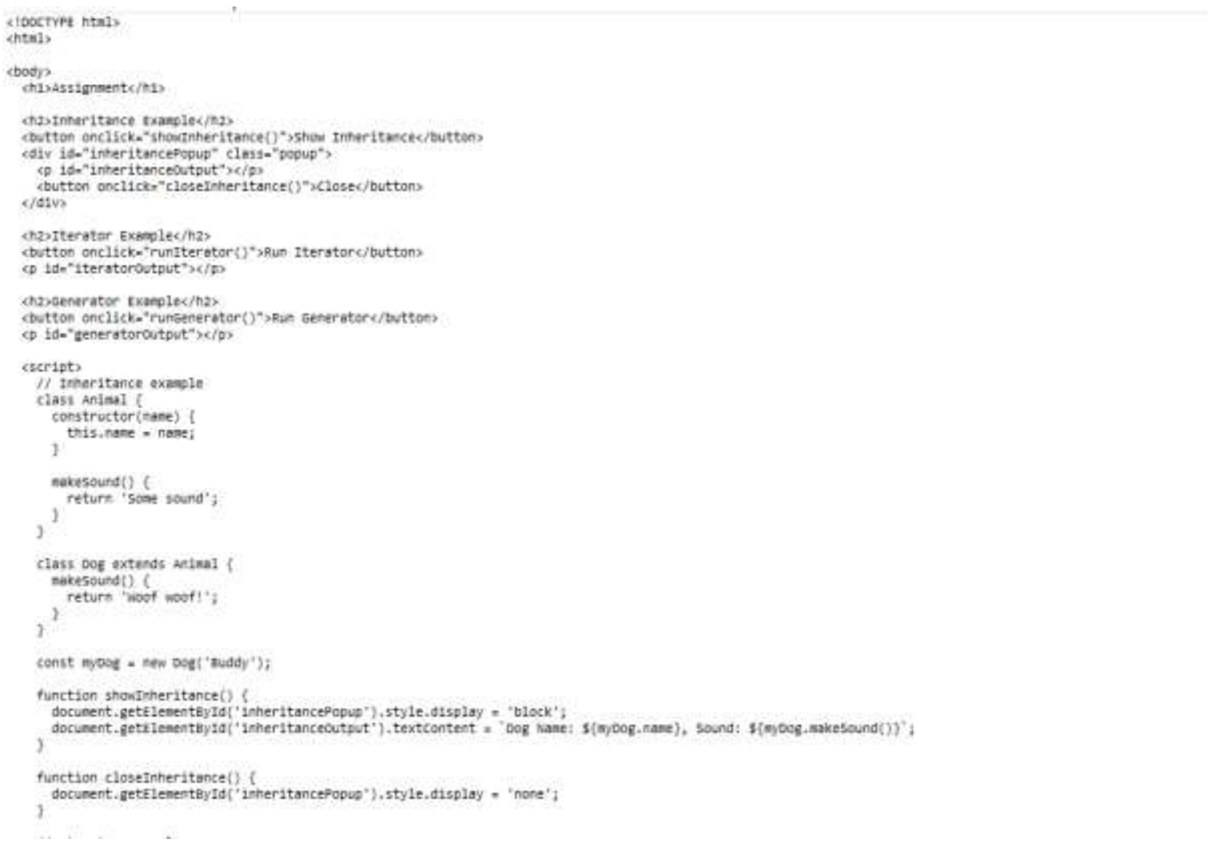
   Iterators are objects that allow you to traverse through a sequence of values, often used with collections like arrays or custom data structures. They provide a way to access elements one by one in a sequential manner.

   In JavaScript, the iterator protocol is implemented using the **Symbol.iterator** symbol and the **.next()** method.

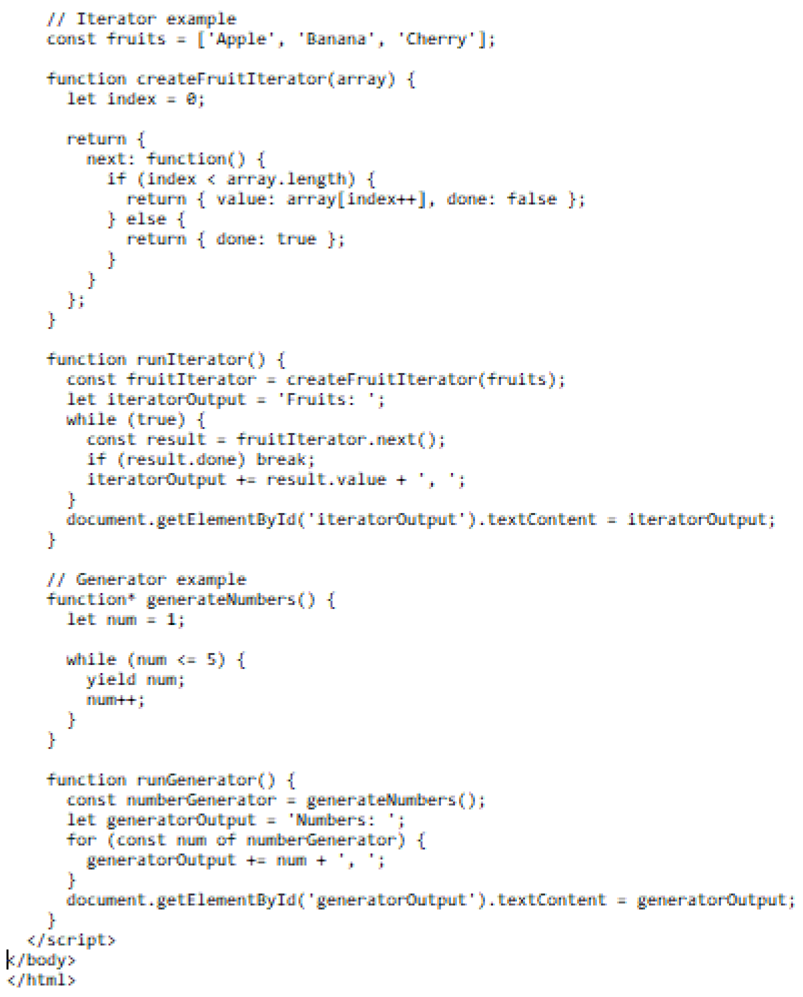
3. Generators-

   Generators are a special type of function in JavaScript that can be paused and resumed. They are defined using the **function*** syntax and utilize the **yield** keyword to produce a sequence of values lazily. This is particularly useful for creating iterators and handling asynchronous operations.

Code :

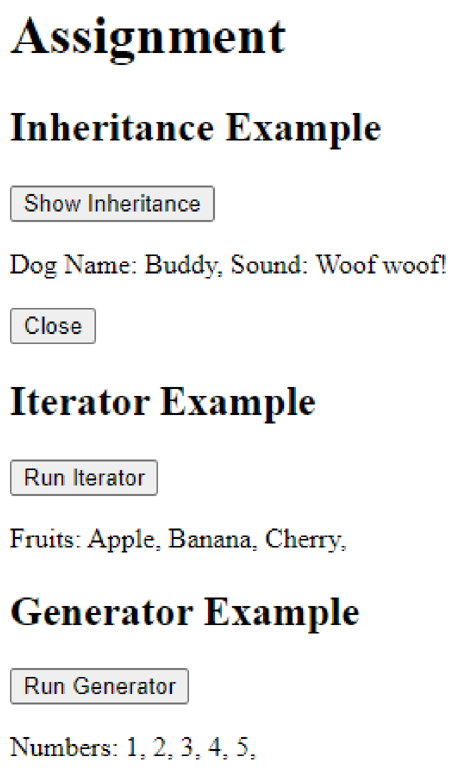


```
<!DOCTYPE html>
<html>

<body>
  <h1>Assignment</h1>

  <h2>Inheritance Example</h2>
  <button onclick="showInheritance()">Show Inheritance</button>
  <div id="inheritancePopup" class="popup">
    <p id="inheritanceOutput"></p>
    <button onclick="closeInheritance()">Close</button>
  </div>

  <h2>Iterator Example</h2>
  <button onclick="runIterator()">Run Iterator</button>
  <p id="iteratorOutput"></p>

  <h2>Generator Example</h2>
  <button onclick="runGenerator()">Run Generator</button>
  <p id="generatorOutput"></p>

  <script>
    // Inheritance example
    class Animal {
      constructor(name) {
        this.name = name;
      }

      makeSound() {
        return 'Some sound';
      }
    }

    class Dog extends Animal {
      makeSound() {
        return 'Woof woof!';
      }
    }

    const myDog = new Dog('Buddy');

    function showInheritance() {
      document.getElementById('inheritancePopup').style.display = 'block';
      document.getElementById('inheritanceOutput').textContent = `Dog Name: ${myDog.name}, Sound: ${myDog.makeSound()}`;
    }

    function closeInheritance() {
      document.getElementById('inheritancePopup').style.display = 'none';
    }
```



```
    // Iterator example
    const fruits = ['Apple', 'Banana', 'Cherry'];

    function createFruitIterator(array) {
      let index = 0;

      return {
        next: function() {
          if (index < array.length) {
            return { value: array[index++], done: false };
          } else {
            return { done: true };
          }
        }
      };
    }

    function runIterator() {
      const fruitIterator = createFruitIterator(fruits);
      let iteratorOutput = 'Fruits: ';
      while (true) {
        const result = fruitIterator.next();
        if (result.done) break;
        iteratorOutput += result.value + ', ';
      }
      document.getElementById('iteratorOutput').textContent = iteratorOutput;
    }

    // Generator example
    function* generateNumbers() {
      let num = 1;

      while (num <= 5) {
        yield num;
        num++;
      }
    }

    function runGenerator() {
      const numberGenerator = generateNumbers();
      let generatorOutput = 'Numbers: ';
      for (const num of numberGenerator) {
        generatorOutput += num + ', ';
      }
      document.getElementById('generatorOutput').textContent = generatorOutput;
    }
  </script>
</body>
</html>
```

Output



```
Assignment

Inheritance Example

Dog Name: Buddy, Sound: Woof woof!

Iterator Example

Fruits: Apple, Banana, Cherry,

Generator Example

Numbers: 1, 2, 3, 4, 5,
```

Conclusion:

In conclusion, inheritance in JavaScript enables the creation of class hierarchies, allowing classes to inherit properties and behaviours from parent classes.

Iterators provide a systematic way to traverse through sequences of data, enhancing the control and readability of iteration processes.

Generators introduce a unique approach to creating iterable sequences, allowing functions to yield values in a paused and resumed manner, thereby facilitating efficient lazy generation and asynchronous handling.

Together, these concepts empower developers to structure code more effectively, navigate data collections seamlessly, and implement dynamic value generation with elegance and flexibility in JavaScript programming.