Mastering For Loops in JavaScript

This document focuses on understanding and effectively using `for` loops in JavaScript. A `for` loop provides a way to repeatedly execute a block of code while a specific condition remains true. It's especially powerful for iterating over arrays, performing repetitive tasks with conditional logic, and handling multi-dimensional data structures.

Syntax of a For Loop

The basic syntax of a `for` loop includes three components:

1. **Initialization:** Set up a counter.
2. **Condition:** Continue looping as long as the condition is true.
3. **Final-expression:** Update the counter after each iteration.

```
for (initialization; condition; final-expression) {
    // Code to execute during each iteration
}
```

Simple For Loop Example

Here's a simple example that prints numbers from 1 to 5:

```
for (let i = 1; i <= 5; i++) {
    console.log(i);
}
```

Example: Sum of an Array

This example sums the numbers in an array using a `for` loop:

```
const numbers = [10, 20, 30, 40, 50];
let sum = 0;

for (let i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}
console.log("Sum:", sum);
```

Example: Finding the Maximum Value

To find the maximum value in an array, you can use a `for` loop to compare each element:

```
const values = [45, 72, 10, 60, 30];
let max = values[0];

for (let i = 1; i < values.length; i++) {
    if (values[i] > max) {
        max = values[i];
    }
}
console.log("Maximum Value:", max);
```

Example: Reversing an Array

Reversing an array without using built-in methods demonstrates control over array indexing:

```
const original = [1, 2, 3, 4, 5];
const reversed = [];

for (let i = original.length - 1; i >= 0; i--) {
    reversed.push(original[i]);
}
console.log("Reversed:", reversed);
```

Example: Nested For Loops - Multiplication Table

Nested `for` loops are useful when dealing with multi-dimensional data. Here's how you can generate a multiplication table:

```
const size = 10; // Size of the multiplication table
let row;

for (let i = 1; i <= size; i++) {
  row = "";
  for (let j = 1; j <= size; j++) {
    row += `${i * j}\t`; // Append each product and a tab
to the row string
  }
  console.log(row); // Output the row string
}
```

Example: Finding Unique Elements in an Array

This example demonstrates how to find unique elements in an array using nested `for` loops:

```
let items = [1, 2, 4, 2, 7, 4, 7, 9, 1, 3];
let uniqueItems = [];

for (let i = 0; i < items.length; i++) {
    let isUnique = true;
    for (let j = 0; j < i; j++) {
        if (items[i] === items[j]) {
            isUnique = false;
            break;
        }
    }
    if (isUnique) {
        uniqueItems.push(items[i]);
    }
}
console.log("Unique Items:", uniqueItems);
```

Example: Sorting an Array with Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm is named for the way smaller or larger elements "bubble" to the top of the list.

Pseudo Code for Bubble Sort Algorithm

```
BEGIN BubbleSort
    SET array = [64, 34, 25, 12, 22, 11, 90]

    FOR i FROM 0 TO length of array - 1
        // Start each pass of the array
        FOR j FROM 0 TO length of array - i - 1
            // Compare each pair of adjacent elements
            IF array[j] > array[j + 1] THEN
                // Swap if they are in the wrong order
                SET temp = array[j]
                SET array[j] = array[j + 1]
                SET array[j + 1] = temp
            END IF
        END FOR
    END FOR

    PRINT "Sorted Array:", array
END BubbleSort
```

Explanation of Each Stage

1. **Initialization:**
    - The array is initialized with a set of unsorted numbers.
2. **Outer Loop (i-loop):**
    - This loop controls the number of times the array is passed through. The variable `i` represents the pass number starting from 0 up to the length of the array minus one.
3. **Inner Loop (j-loop):**
    - This loop goes through the array from the first element to the position determined by subtracting the pass number (`i`) and one from the array's length. This avoids unnecessary comparisons with elements that have already been sorted in previous passes.
4. **Comparison and Swapping:**

- Inside the inner loop, each element at position `j` is compared with the next element (`j + 1`). If the current element (`array[j]`) is greater than the next element (`array[j + 1]`), a swap is performed using a temporary variable `temp` to hold the value during the swap.

5. **Output:**
    - Once all passes are completed and no more swaps are necessary (indicating that the array is sorted), the sorted array is printed.

Visual Representation

- **Before Sorting:** [64, 34, 25, 12, 22, 11, 90]
- **First Pass:** Compare and swap, leading to [34, 25, 12, 22, 11, 64, 90]
- **Second Pass:** Compare and swap, leading to [25, 12, 22, 11, 34, 64, 90]
- **Continuing:** The process repeats, gradually moving the largest unsorted element to its correct position at the end of the list.
- **After Sorting:** [11, 12, 22, 25, 34, 64, 90]

Code Example with Comments

Here is the Bubble Sort algorithm implemented in JavaScript, annotated with detailed comments for clarity:

```
// Initialize the array to be sorted
const array = [64, 34, 25, 12, 22, 11, 90];

// Outer loop: Runs `n-1` times where `n` is the length
of the array
// This loop controls the number of passes through the
array
for (let i = 0; i < array.length - 1; i++) {
    // Inner loop: Goes through the array from the start
to `n-i-1`
    // As the largest element of the remaining unsorted
section "bubbles up" to the correct position
    // after each pass, we reduce the end point of the
inner loop by 1 each time (`i` is the number
    // of passes already completed)
    for (let j = 0; j < array.length - i - 1; j++) {
        // Compare adjacent elements
        if (array[j] > array[j + 1]) {
            // Swap elements if they are in the wrong
order
            let temp = array[j];
            array[j] = array[j + 1];
            array[j + 1] = temp;
        }
    }
}

// Output the sorted array
console.log("Sorted Array:", array);
```

Explanation of Key Concepts and Steps

1. **Outer Loop (Passes):**
    - **Purpose:** Controls the number of complete passes through the array necessary to sort it.
    - **Termination Condition:** `(i < array.length - 1)` ensures that the loop iterates one less than the number of elements in the array since the last pass is not needed (the elements are already sorted).

2. **Inner Loop (Comparisons):**
    - **Purpose:** Performs the actual comparisons and potential swaps between adjacent elements in the unsorted portion of the array.
    - **Range:** From the beginning of the array to `(array.length - i - 1)`. The variable `i` indicates how many elements at the end of the array have been sorted and do not need to be included in further comparisons.
3. **Comparison and Swapping:**
    - **Condition:** `(array[j] > array[j + 1])` checks if the current element is greater than the next element.
    - **Action:** If the condition is true, the elements are swapped. This is done using a temporary variable `temp` to hold one of the values during the swap.

Advantages and Disadvantages of Bubble Sort

Advantages:

- Simplicity: The algorithm is straightforward to understand and implement.
- Detection of sorted list: The algorithm can be easily modified to stop early if it finds that the list is already sorted.

Disadvantages:

- Inefficiency: Bubble Sort has a worst-case and average complexity of (O(n^2)) where (n) is the number of items being sorted. Thus, it is not suitable for large data sets.