Loops and Iterators

Week 3

Adib Sakhawat
IUT CSE ‘21

Loops are a fundamental concept in programming that allow you to execute a block of code repeatedly based on a condition. JavaScript provides several types of loops to handle different scenarios efficiently. In this guide, we'll explore:

- `for` loops
- `while` loops
- `do...while` loops
- `for...of` loops

We'll delve into how each loop operates, their syntax, and provide code examples to illustrate their usage.

---

1. The `for` Loop

The `for` loop is commonly used when you know in advance how many times you want to execute a statement or a block of code.

Syntax

```
for (initialization; condition; increment) {
    // Code to execute in each iteration
}
```

- **Initialization**: Initializes the loop variable and is executed once before the loop starts.
- **Condition**: Evaluated before each iteration; if it's `true`, the loop continues; if `false`, the loop stops.

- **Increment**: Updates the loop variable after each iteration.

Example

Print numbers from 1 to 5:

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

Explanation:

- Starts with `i = 1`.
- Checks if `i <= 5`; if `true`, executes `console.log(i)`.
- Increments `i` by 1 after each iteration.

Output:

```
1
2
3
4
5
```

Nested `for` Loop

Loops can be nested to handle multidimensional data structures.

```
for (let i = 1; i <= 3; i++) {
    for (let j = 1; j <= 2; j++) {
        console.log(`i = ${i}, j = ${j}`);
    }
}
```

Output:

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
```

```
i = 3, j = 1
i = 3, j = 2
```

---

2. The `while` Loop

The `while` loop executes a block of code **as long as** a specified condition is `true`. It's ideal when the number of iterations is not known beforehand.

Syntax

```
while (condition) {
    // Code to execute in each iteration
}
```

Example

Print numbers from 1 to 5:

```
let i = 1;

while (i <= 5) {
    console.log(i);
    i++;
}
```

Explanation:

- Initializes `i` outside the loop.
- Checks if `i <= 5` before each iteration.
- Increments `i` within the loop.

Output:

```
1
2
3
4
5
```

Infinite Loop Caution

Ensure that the loop condition will eventually become `false`; otherwise, you'll create an infinite loop.

---

3. The `do...while` Loop

The `do...while` loop is similar to the `while` loop, but it guarantees that the code block executes **at least once** before the condition is evaluated.

Syntax

```
do {
    // Code to execute
} while (condition);
```

Example

Prompt the user until they enter a number greater than 10:

```
let number;

do {
    number = parseInt(prompt("Enter a number greater than 10:"));
} while (number <= 10);

console.log(`You entered ${number}.`);
```

Explanation:

- Prompts the user at least once.
- Continues to prompt until `number > 10`.

---

4. The `for...of` Loop

The `for...of` loop iterates over iterable objects like arrays, strings, maps, and sets. It's a concise and readable way to loop through the elements of an iterable.

Syntax

```
for (variable of iterable) {
    // Code to execute for each element
}
```

Example with Arrays

Iterate over an array of fruits:

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

for (const fruit of fruits) {
    console.log(fruit);
}
```

Output:

```
Apple
Banana
Cherry
```

Example with Strings

Iterate over each character in a string:

```
const word = 'JavaScript';

for (const letter of word) {
    console.log(letter);
}
```

Output:

```
J
a
v
a
S
```

```
c
r
i
p
t
```

Loop Control Statements

`break` Statement

The `break` statement exits the loop immediately, skipping any remaining iterations.

Example:

```
for (let i = 1; i <= 10; i++) {
    if (i === 5) {
        break;
    }
    console.log(i);
}
```

Output:

```
1
2
3
4
```

`continue` Statement

The `continue` statement skips the current iteration and moves to the next one.

Example:

```
for (let i = 1; i <= 5; i++) {
    if (i === 3) {
        continue;
    }
```

```
    console.log(i);
}
```

Output:

```
1
2
4
5
```

Practical Examples

Summing Numbers in an Array

Using a `for` loop:

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

for (let i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}

console.log(`Total sum: ${sum}`);
```

Output:

```
Total sum: 100
```

Using a `for...of` loop:

```
let sum = 0;

for (const num of numbers) {
    sum += num;
}
```

```
console.log(`Total sum: ${sum}`);
```

Searching for an Element

Using a `while` loop:

```
const items = ['Book', 'Pen', 'Notebook', 'Pencil'];
let index = 0;
let found = false;

while (index < items.length && !found) {
    if (items[index] === 'Notebook') {
        console.log(`Found 'Notebook' at index ${index}`);
        found = true;
    }
    index++;
}
```

Output:

```
Found 'Notebook' at index 2
```

Best Practices

- **Choose the Right Loop**: Use the loop that best fits your scenario.
  - Use `for` loops when you know the number of iterations.
  - Use `while` loops when the number of iterations is uncertain.
  - Use `for...of` loops for iterating over iterable objects.
- **Avoid Infinite Loops**: Ensure that your loop condition will eventually become `false`.
- **Use Descriptive Variable Names**: Improves code readability.
- **Minimize Side Effects**: Avoid modifying the iterable within the loop to prevent unexpected behavior.

Common Pitfalls

Off-by-One Errors

Be careful with loop conditions to ensure you don't miss the first or last iteration.

Incorrect:

```
for (let i = 0; i <= array.length; i++) {
    // May cause an undefined error when i equals array.len
gth
}
```

Correct:

```
for (let i = 0; i < array.length; i++) {
    // Safe iteration over array indices
}
```

Using `for...in` with Arrays

The `for...in` loop is intended for objects, not arrays. It iterates over enumerable properties, which can lead to unexpected results with arrays.

Example:

```
const arr = ['a', 'b', 'c'];

for (const index in arr) {
    console.log(arr[index]);
}
```

Potential Issues:

- Iterates over inherited properties.
- The order is not guaranteed.

Recommendation: Use `for...of` or traditional `for` loops for arrays.

Summary

- `for` **Loop**: Ideal for a known number of iterations.
- `while` **Loop**: Suitable when the number of iterations is unknown.
- `do...while` **Loop**: Guarantees the loop runs at least once.
- `for...of` **Loop**: Best for iterating over iterable objects like arrays and strings.
- **Control Statements**: `break` exits the loop; `continue` skips to the next iteration.