Loops in JavaScript

Loops in JavaScript

Sure! Here’s a quick overview of loops in JavaScript:

---

1. for loop

Used when you know how many times you want to run the loop.

```js
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

- `i = 0`: initialization
- `i < 5`: condition to keep looping
- `i++`: increment after each iteration

---

2. while loop

Runs as long as a condition is true.

```js
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

---

3. do...while loop

Runs at least once, then repeats while the condition is true.

```js
```

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

---

4. for...in loop

Loops over the **keys** of an object.

```js
const obj = { a: 1, b: 2, c: 3 };

for (let key in obj) {
  console.log(key, obj[key]);
}
```

---

5. for...of loop

Loops over **values** of iterable objects like arrays, strings, etc.

```js
const arr = [10, 20, 30];

for (let value of arr) {
  console.log(value);
}
```

If you want, I can show you some examples or help with a specific use case. What kind of loops are you working with?

Provide conceptual use cases or examples (also complex)

Absolutely! Here are some conceptual use cases and examples for each type of loop in JavaScript — including some more complex scenarios that show their real power.

---

1. for loop — Use Case: Processing a fixed number of items (e.g., pagination)

Example: Paginate through search results, displaying 10 results per page

```js
const results = [...Array(53).keys()]; // dummy array: [0,1,2,...,52]
const resultsPerPage = 10;
const totalPages = Math.ceil(results.length / resultsPerPage);

for (let page = 1; page <= totalPages; page++) {
  console.log(`Page ${page}:`);
  for (let i = (page - 1) * resultsPerPage; i < page * resultsPerPage && i < results.length; i++) {
    console.log(`Result ${results[i]}`);
  }
}
```

Concept: Nested loops to handle chunks of data (pagination).

---

2. while loop — Use Case: Wait for a condition dynamically (e.g., retry until success)

Example: Retry a function until it returns true or max attempts reached

```js
function unreliableOperation() {
  return Math.random() > 0.7; // 30% success rate
}

let attempts = 0;
const maxAttempts = 5;
let success = false;

while (!success && attempts < maxAttempts) {
  success = unreliableOperation();
  attempts++;
  console.log(`Attempt ${attempts}: ${success ? 'Success' : 'Fail'}`);
}

if (!success) {
  console.log('Operation failed after maximum attempts.');
}
```

Concept: Looping until a dynamic condition is met or limit is reached.

---

3. do...while loop — Use Case: Prompt user input at least once

Example: Simulate user input prompt until input is valid

js

```js
let input;
do {
  input = prompt("Enter a number greater than 10:");
} while (isNaN(input) || Number(input) <= 10);

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

Concept: Guaranteed execution at least once, then continue if invalid.

---

4. for...in loop — Use Case: Iterate over object properties, including nested objects

Example: Deep clone an object (shallow for now, recursive deeper cloning can use recursion)

js

```js
const original = {
  name: "Alice",
  age: 30,
  address: {
    city: "Wonderland",
    zip: 12345,
  },
};

const clone = {};
for (let key in original) {
  if (typeof original[key] === 'object') {
    clone[key] = { ...original[key] }; // shallow clone for nested object
  } else {
    clone[key] = original[key];
  }
}

console.log(clone);
```

Concept: Loop over keys of objects for cloning or property manipulation.

---

5. for...of loop — Use Case: Iterate over iterable data structures (arrays, strings, maps, sets)

Example: Processing a list of orders, filtering by status

```js
const orders = [
  { id: 1, status: 'pending' },
  { id: 2, status: 'shipped' },
  { id: 3, status: 'delivered' },
  { id: 4, status: 'pending' },
];

for (const order of orders) {
  if (order.status === 'pending') {
    console.log(`Order ${order.id} needs to be processed.`);
  }
}
```

Concept: Cleaner looping over arrays and other iterable collections.

Bonus: Complex Use Case Combining Loops

Use Case: Find prime numbers between 2 and 50

```js
for (let num = 2; num <= 50; num++) {
 let isPrime = true;
 for (let i = 2; i <= Math.sqrt(num); i++) {
   if (num % i === 0) {
     isPrime = false;
     break;
   }
 }
 if (isPrime) {
   console.log(num);
 }
}
```

Concept: Nested loops, break statement, and algorithm optimization (looping up to sqrt of num).

Want me to build or explain a specific example for you? Maybe something you're working on?

2. Loop Control Statements

break — exit the loop immediately.

continue — skip current iteration, continue next.

Labelled loops (rare but useful).

3. Looping Over Arrays and Objects

Looping arrays with:

Classic for loop

for...of (ES6) — for iterable objects like arrays.

forEach method (array method).

Looping objects with:

for...in (to iterate over object keys).

Object.keys(), Object.values(), Object.entries() + loops.

4. Nested Loops

Understand how loops inside loops work.

Common in matrix or grid problems.

Great! Let’s break down those points one by one with explanations and examples to deepen your understanding.

---

2. Loop Control Statements

break

- Immediately exits the nearest enclosing loop.
- Useful when you want to stop looping once a condition is met.

```js
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;  // Exit loop when i is 5
  }
  console.log(i);  // Prints 0,1,2,3,4
}
```

continue

- Skips the current iteration and moves to the next iteration of the loop.

```js
for (let i = 0; i < 10; i++) {
  if (i % 2 === 0) {
    continue;  // Skip even numbers
  }
  console.log(i);  // Prints only odd numbers: 1,3,5,7,9
}
```

Labelled loops

- Labels let you control which loop you break or continue from, especially in nested loops.

```js
outerLoop: for (let i = 0; i < 3; i++) {
  innerLoop: for (let j = 0; j < 3; j++) {
    if (i === 1 && j === 1) {
      break outerLoop;  // Breaks the outer loop directly
    }
    console.log(`i=${i}, j=${j}`);
  }
}
```

---

3. Looping Over Arrays and Objects

Looping arrays

- **Classic for loop**

```js
const arr = ['a', 'b', 'c'];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

- **for...of** (ES6)

```js
for (const val of arr) {
  console.log(val);
}
```

- **forEach method**

```js
```

```
arr.forEach((val, index) => {
  console.log(index, val);
});
```

Looping objects

- **for...in** (iterates over keys)

```js
const obj = { name: 'Alice', age: 25 };
for (const key in obj) {
  console.log(key, obj[key]);
}
```

- **Object.keys(), Object.values(), Object.entries()**

```js
const keys = Object.keys(obj);      // ['name', 'age']
const values = Object.values(obj);  // ['Alice', 25]
const entries = Object.entries(obj);// [['name','Alice'], ['age',25]]

// Loop keys
for (const key of keys) {
  console.log(key, obj[key]);
}

// Loop entries
for (const [key, value] of entries) {
  console.log(key, value);
}
```

---

4. Nested Loops

Loops inside loops, useful for working with grids, matrices, or multi-dimensional data.

Example: Print a multiplication table (5x5)

```js
for (let row = 1; row <= 5; row++) {
  let line = '';
  for (let col = 1; col <= 5; col++) {
    line += (row * col) + '\t';
  }
  console.log(line);
}
```

Output:

```
1  2  3  4  5
2  4  6  8  10
3  6  9  12   15
4  8  12   16   20
5  10   15   20   25
```

Example: Working with 2D array (matrix)

```js
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];

for (let i = 0; i < matrix.length; i++) {
  for (let j = 0; j < matrix[i].length; j++) {
    console.log(`matrix[${i}][${j}] = ${matrix[i][j]}`);
  }
}
```

If you want, I can help you with specific complex examples using these concepts or create exercises for you. How’s this so far?

9. Asynchronous Loops

Using loops with async/await (advanced, less common in interviews but good to know).

Promise.all with loops.

Great topic! Asynchronous loops are super important in modern JavaScript because a lot of operations (API calls, file reads, timers) are async. Let’s go through two main ways to work with async code inside loops:

---

9. Asynchronous Loops

A) Using `async/await` inside loops

---

1. `for` / `for...of` with `await`

You can **await** inside loops, but be careful: this makes the loop **run sequentially** — each iteration waits for the previous one.

```js
async function processItems(items) {
  for (const item of items) {
    const result = await fakeAsyncOperation(item);
    console.log(result);
  }
  console.log('All done sequentially');
}

function fakeAsyncOperation(item) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(`Processed ${item}`);
    }, 1000);
  });
}

processItems([1, 2, 3]);
```

Output:

```scss
(after 1 sec) Processed 1
(after 2 sec) Processed 2
(after 3 sec) Processed 3
All done sequentially
```

---

2. `forEach` and async/await

Be careful: `forEach` **does not** work well with async/await because it doesn't await promises inside the callback!

```js
[1, 2, 3].forEach(async item => {
  const result = await fakeAsyncOperation(item);
  console.log(result);
});
console.log('This logs before async ops complete');
```

This prints the last line before async ops finish, which can cause bugs.

B) Running async operations in parallel using `Promise.all`

If you want to process many items **at the same time** (in parallel), use `Promise.all` with `.map()` or `.forEach()` to collect promises, then await them all.

```js
async function processAll(items) {
  const promises = items.map(item => fakeAsyncOperation(item));
  const results = await Promise.all(promises);
  results.forEach(res => console.log(res));
  console.log('All done in parallel');
}

processAll([1, 2, 3]);
```

Output (after ~1 second):

```bash
Processed 1
Processed 2
Processed 3
All done in parallel
```

Summary:

Method	Runs sequentially?	Code example	When to use
`for` / `for...of` + `await`	Yes	`for (const x of arr) await`	When order matters
`forEach` + `async/await`	No (doesn't wait)	Not recommended	Avoid for async logic
`Promise.all` + `.map()`	No (parallel)	`Promise.all(arr.map(...))`	When operations can run in parallel

---

Bonus: Advanced — Async generators + `for await...of`

You can even loop asynchronously over data streams or async sources:

```js
async function* asyncGenerator() {
  yield await Promise.resolve(1);
  yield await Promise.resolve(2);
  yield await Promise.resolve(3);
}

(async () => {
  for await (const num of asyncGenerator()) {
    console.log(num);
  }
})();
```

Want me to show you a real-world example using async loops? Or dive deeper into any of these?