Notes - JavaScript - Gang of Four - Behavioral - Iterator

Dr Nick Hayward

A brief introduction to the Iterator pattern in JavaScript/TypeScript.

What is the Iterator pattern?

Iterator provides a uniform way to traverse elements of a collection without exposing its internal representation. In JavaScript, the pattern is baked into the language: any object implementing `Symbol.iterator` (sync) or `Symbol.asyncIterator` (async) can be consumed by `for...of` , spread syntax, array destructuring, or `for await...of` .

Benefits:

- decouples traversal logic from container implementation
- enables lazy evaluation (generators, on-demand pagination)
- supports composition (pipelines of generators)
- allows uniform consumption (same loop constructs for arrays, sets, maps, custom iterables)

Common use cases:

- streaming large datasets chunk by chunk
- file or network pagination
- building lazy transformation pipelines (map/filter without intermediate arrays)
- creating domain-specific sequence abstractions (e.g., calendar ranges)

Worked example 1 — custom iterable via `Symbol.iterator`

```
class Catalog {
    constructor(items) { this.items = items; }
    [Symbol.iterator]() {
        let index = 0; const data = this.items;
        return {
            next() {
                if (index < data.length) {
                    return { value: data[index++], done: false };
                }
                return { value: undefined, done: true };
            }
        };
    }
}

const members = [ ...Array.from({ length: 5 }, (_, i) => `book${i+1}`), 'author1', 'author2' ];
for (const item of new Catalog(members)) {
    console.log(item);
}
```

Why it works: The object returns an iterator with a `next()` method returning `{ value, done }` . `for...of` drives the iteration until `done: true` .

Worked example 2 — generator-based iterable

```
class CatalogGen {
    constructor(items) { this.items = items; }
```

```
    *[Symbol.iterator]() {
        for (const it of this.items) {
            yield it; // pause + resume lazily
        }
    }
}

for (const item of new CatalogGen(['A','B','C'])) {
    console.log(item);
}
```

Why it works: Generators simplify iterator creation— `yield` handles `next()` state management.

Worked example 3 — lazy transform pipeline

```
function *range(start, end) {
    for (let i = start; i <= end; i++) yield i;
}

function *map(iterable, fn) {
    for (const x of iterable) yield fn(x);
}

function *filter(iterable, pred) {
    for (const x of iterable) { if (pred(x)) yield x; }
}

// pipeline: squares of even numbers from 1..10
const pipeline = filter(map(range(1, 10), n => n * n), n => n % 2 === 0);
console.log([...pipeline]); // [4, 16, 36, 64, 100]
```

Why it works: Each stage yields values lazily — no intermediate arrays, enabling efficient large-range processing.

Worked example 4 — async iterator (paginated fetch)

```
// Simulated paginated API
async function fetchPage(page) {
    await new Promise(r => setTimeout(r, 50));
    if (page > 3) return { items: [], next: null };
    return { items: [`item${page}-1`, `item${page}-2`], next: page + 1 };
}

async function *paginate(start = 1) {
    let page = start;
    while (page) {
        const { items, next } = await fetchPage(page);
        for (const item of items) yield item;
        page = next;
    }
}

(async () => {
```

```
    for await (const item of paginate()) {
        console.log('async item:', item);
    }
})();
```

Why it works: The async generator implements `Symbol.asyncIterator`; `for await...of` pulls values as Promises resolve.

Edge cases and trade-offs

- premature materialization: spreading or `[...iter]` turns lazy sequences into arrays — avoid for large datasets
- resource cleanup: use `finally` blocks in generators if external resources need closing when consumer stops early
- backpressure: async iterators can model it; consider batching or pausing if producer outpaces consumer
- error handling: wrap generator bodies with try/catch
- performance: tight loops with generators may be marginally slower than manual indexing