Iterators and the iterator protocol

🔄 Iterators and the Iterator Protocol in JavaScript — Explained in Detail

🔷 What is an Iterator?

An **iterator** is an object that allows you to **iterate (loop)** over a collection (like arrays, strings, maps, etc.) **one item at a time**.

An object becomes an **iterator** when it has a method named `next()` that returns an object with:

```
{ value: <data>, done: <true/false> }
```

🔶 Iterator Protocol

To conform to the **iterator protocol**, an object must have a `next()` method that returns:

Property	Meaning
`value`	The current value being iterated
`done`	`true` if the iteration is complete, `false` otherwise

✅ Example: Manual Iterator

```
function createIterator(array) {
  let index = 0;
  return {
    next: function () {
      if (index < array.length) {
        return { value: array[index++], done: false };
```

```
    } else {
      return { value: undefined, done: true };
    }
  }
 };
}

const it = createIterator(['a', 'b', 'c']);

console.log(it.next()); // { value: 'a', done: false }
console.log(it.next()); // { value: 'b', done: false }
console.log(it.next()); // { value: 'c', done: false }
console.log(it.next()); // { value: undefined, done: true }
```

🔁 Iterable Protocol

An object is **iterable** if it implements the `[Symbol.iterator]()` **method**, which returns an iterator.

Built-in Iterables:

- Arrays
- Strings
- Maps
- Sets
- TypedArrays
- NodeLists
- Arguments

🔶 Example: Array Iterator

```
const arr = ['x', 'y', 'z'];
const iterator = arr[Symbol.iterator]();
```

```
console.log(iterator.next()); // { value: 'x', done: false }
console.log(iterator.next()); // { value: 'y', done: false }
console.log(iterator.next()); // { value: 'z', done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

🔄 for...of Loop (Uses Iterator Internally)

```
const str = "Abhi";
for (let char of str) {
  console.log(char); // A b h i
}
```

The `for...of` loop automatically calls the `[Symbol.iterator]()` method and then repeatedly calls `next()`.

👨‍💻 Custom Iterable Object

You can make your object iterable by defining `[Symbol.iterator]()`:

```
const customIterable = {
  data: [10, 20, 30],
  [Symbol.iterator]() {
    let index = 0;
    return {
      next: () => {
        if (index < this.data.length) {
          return { value: this.data[index++], done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
```

```
};

for (let num of customIterable) {
  console.log(num); // 10 20 30
}
```

📌 Summary Table

Concept	Description
Iterator	Object with `next()` method
`next()` return	`{ value: ..., done: true/false }`
Iterable	Object with `[Symbol.iterator]()` method
Built-in iterables	Array, String, Set, Map, etc.
`for...of` loop	Automatically uses iterator protocol
Custom iterable	Define `[Symbol.iterator]()` manually

Would you like an example using `Set`, `Map`, or how iterators work with **Generators** next?