Session-05

Managing Objects Javascript

Thanos is on a mission to make his website standout from his rest of universe with Javascript

Web Fundamentals

Session-05

Managing Objects in Javascript

Agenda : Managing Objects

01 JSON Notation

02 Memoization, Symbols

03 Iterables , Iterators

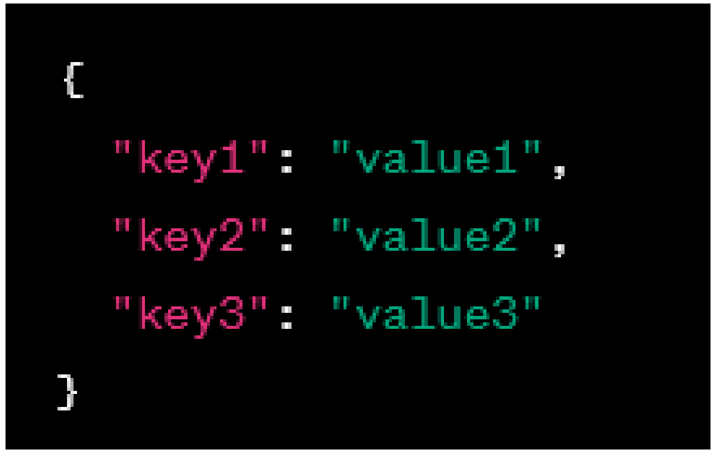
04 Generators

05 Proxies, Polyfills

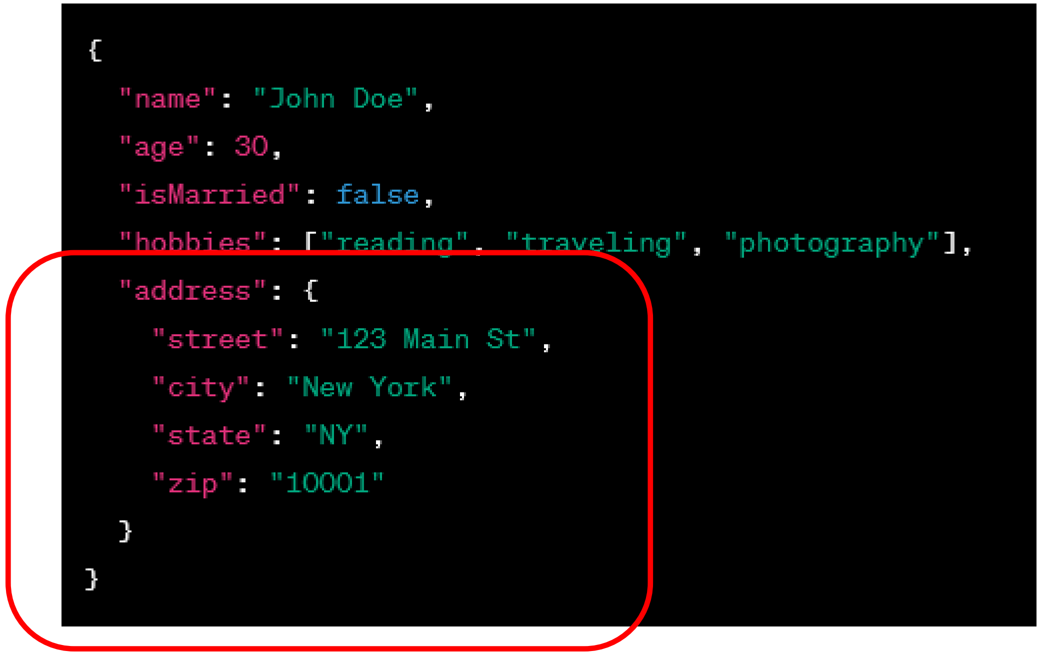
06 Hands-On

JSON

Stands for JavaScript Object Notation
Lightweight data interchange format used in many web applications for data transmission
Data is always represented as key-value pairs
The keys are strings and the values can be any valid JSON data type.
Values can include strings, numbers, booleans, arrays, and objects.



```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

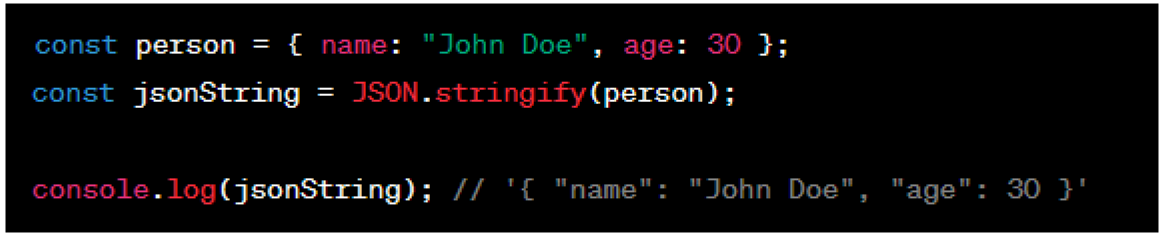


```
{
  "name": "John Doe",
  "age": 30,
  "isMarried": false,
  "hobbies": ["reading", "traveling", "photography"],
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "state": "NY",
    "zip": "10001"
  }
}
```

Value can be any JSON

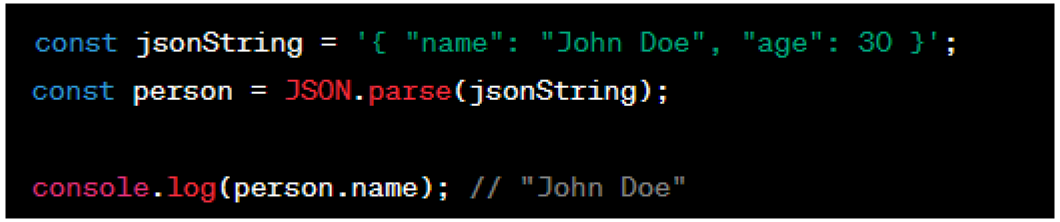
Creating & Parsing JSON

We can generate JSON data from a JavaScript object using the **JSON.stringify()**



```
const person = { name: "John Doe", age: 30 };
const jsonString = JSON.stringify(person);

console.log(jsonString); // '{ "name": "John Doe", "age": 30 }'
```

We can parse JSON data from a JavaScript object using the **JSON.parse()**



```
const jsonString = '{ "name": "John Doe", "age": 30 }';
const person = JSON.parse(jsonString);

console.log(person.name); // "John Doe"
```

Memoization

Memoization is a technique used to speed up function execution **by caching its results.**

Instead of re-executing a function with the same arguments, we can return the cached result.

```
function memoize(func) {
  const cache = {};

  return function(...args) {
    const key = JSON.stringify(args);

    if (cache.hasOwnProperty(key)) {
      return cache[key];
    }

    const result = func(...args);
    cache[key] = result;
    return result;
  };
}
```

Knowledge check

A function that makes API calls with the same parameters multiple times, How can we improve performance by reducing the number of requests to the remote server?

Memoize it in local and serve....

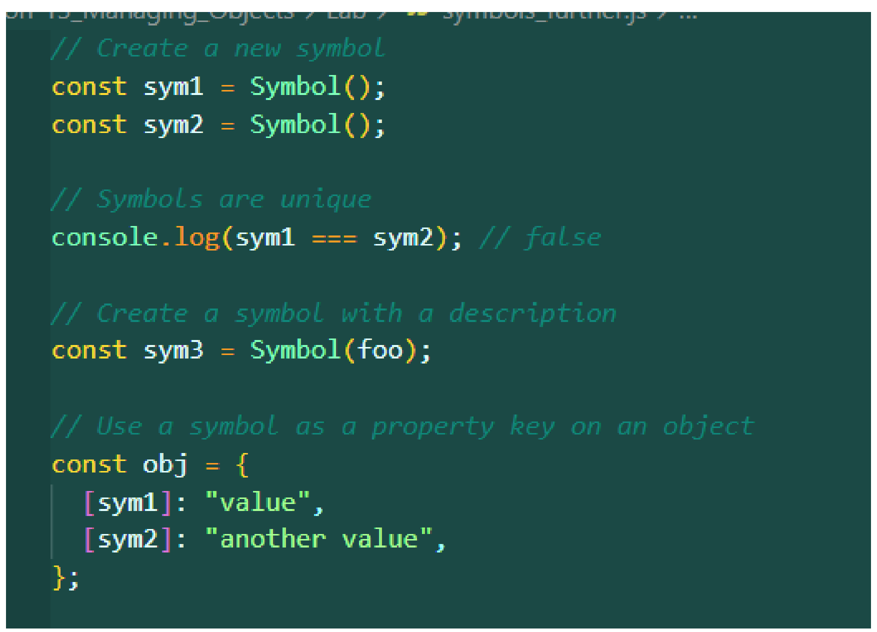
Why do we need “Symbols”?

Naming Collision

Let’s see an example

Symbols

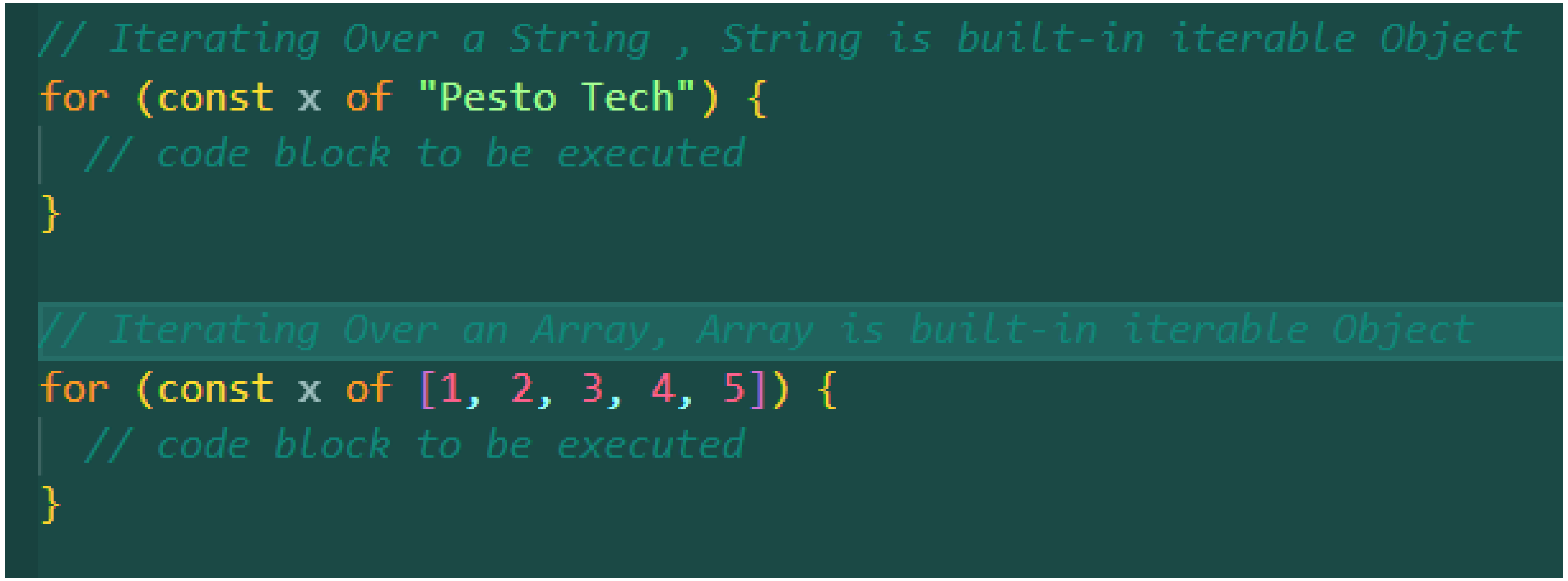
- Object symbols are a powerful feature that allow **for unique property keys** on an object
- Symbols are a **primitive data type** in JavaScript, **represented by the Symbol() function**
- Square bracket notation can be used to use symbols as property keys on an object
- **Object.defineProperty()** can be used to create symbol properties on an object
- **Object.getOwnPropertySymbols()** retrieves an array of all symbol properties on an object
- Symbols are **unique and cannot be duplicated**
- Symbols are useful for **creating private properties** on an object



```
// Create a new symbol
const sym1 = Symbol();
const sym2 = Symbol();

// Symbols are unique
console.log(sym1 === sym2); // false

// Create a symbol with a description
const sym3 = Symbol(foo);

// Use a symbol as a property key on an object
const obj = {
  [sym1]: "value",
  [sym2]: "another value",
};
```

Iterables

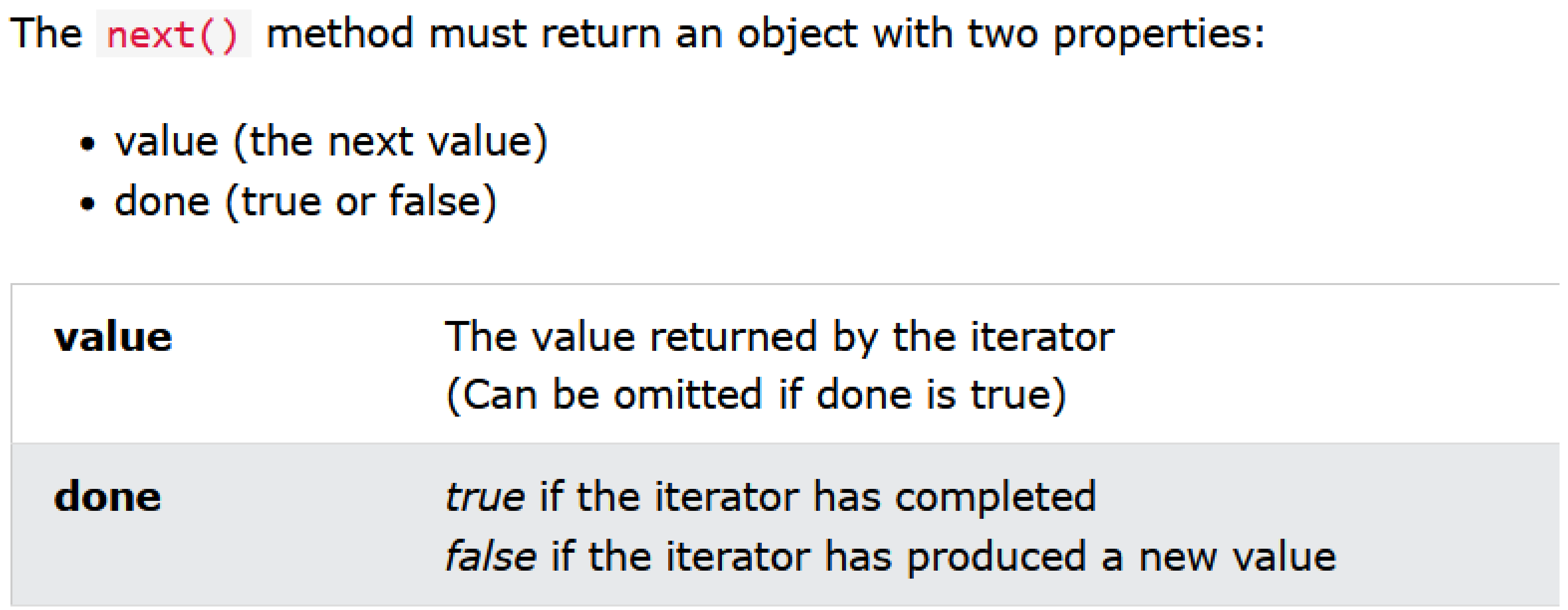
- **Iterable objects** are objects that can be iterated over with **for..of.**
- **Iterables** must implement **the Symbol.iterator method**.



```
// Iterating Over a String , String is built-in iterable Object
for (const x of "Pesto Tech") {
  // code block to be executed
}

// Iterating Over an Array, Array is built-in iterable Object
for (const x of [1, 2, 3, 4, 5]) {
  // code block to be executed
}
```

Iterators

- **The iterator protocol** defines how to **produce a sequence of values** from an object.
- An **object becomes an iterator** when it implements **a next() method**.



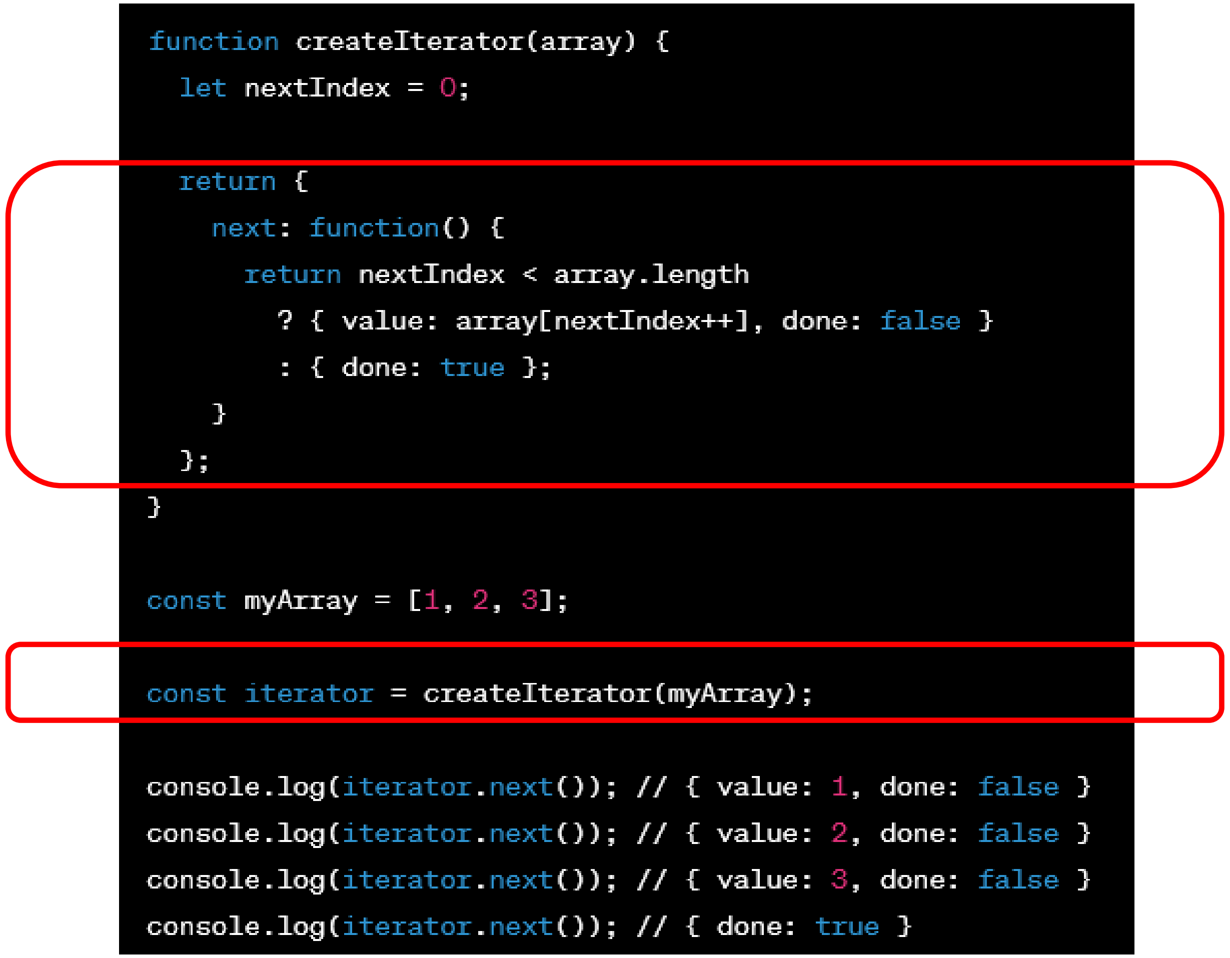
The `next()` method must return an object with two properties:

- value (the next value)
- done (true or false)

value	The value returned by the iterator (Can be omitted if done is true)
done	*true* if the iterator has completed *false* if the iterator has produced a new value

Iterators - Example

- **The iterator protocol** defines how to **produce a sequence of values** from an object.
- An **object becomes an iterator** when it implements **a next() method**.



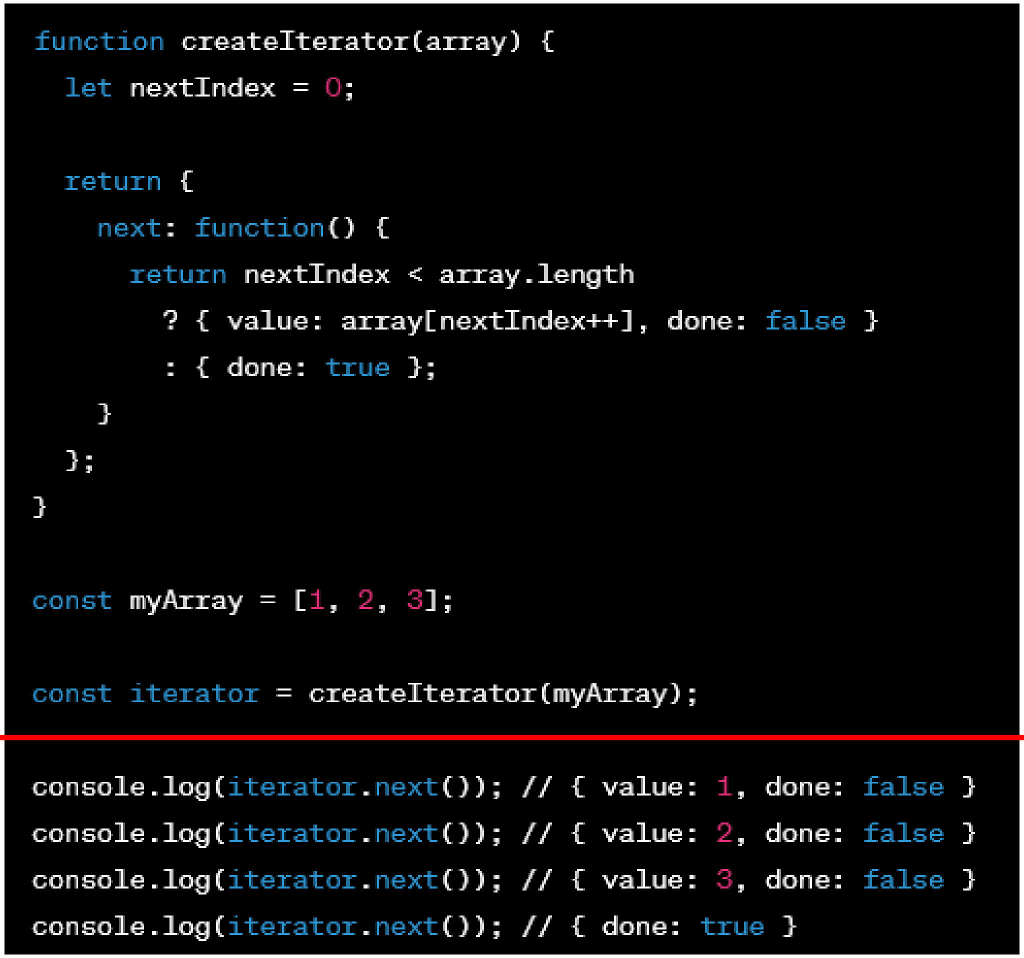
```
function createIterator(array) {
  let nextIndex = 0;

  return {
    next: function() {
      return nextIndex < array.length
        ? { value: array[nextIndex++], done: false }
        : { done: true };
    }
  };
}

const myArray = [1, 2, 3];

const iterator = createIterator(myArray);

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { done: true }
```

Returning Iterator

Assigning Iterator

Iterators

- **The iterator protocol** defines how to **produce a sequence of values** from an object.
- An **object becomes an iterator** when it implements **a next() method**.



```
function createIterator(array) {
  let nextIndex = 0;

  return {
    next: function() {
      return nextIndex < array.length
        ? { value: array[nextIndex++], done: false }
        : { done: true };
    }
  };
}

const myArray = [1, 2, 3];

const iterator = createIterator(myArray);

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { done: true }
```

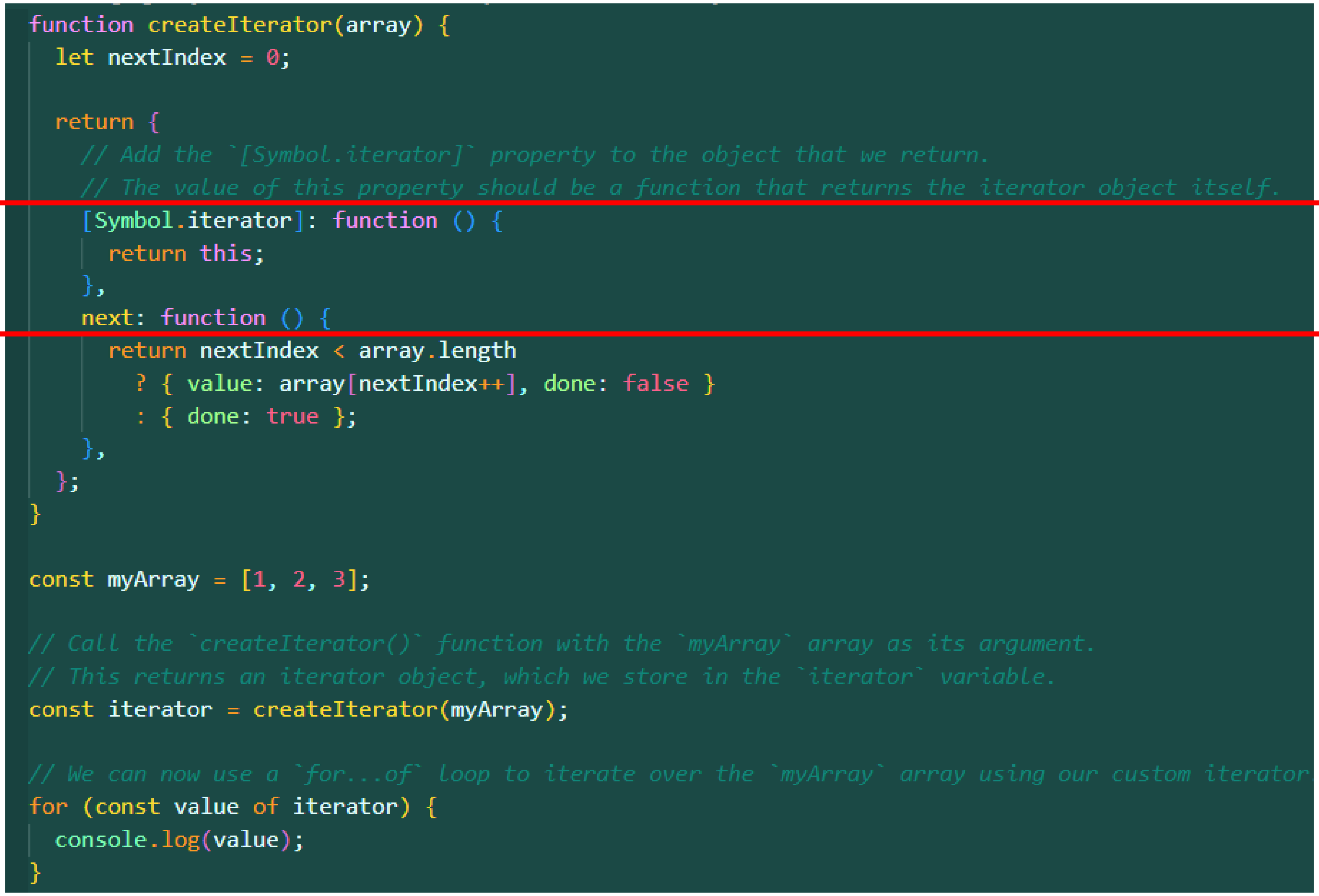
How to make the iterator as iterable object?

Is this iterable?

No, Because it doesn't support for..of

Iterators

By Adding [Symbol.Iterator] Property



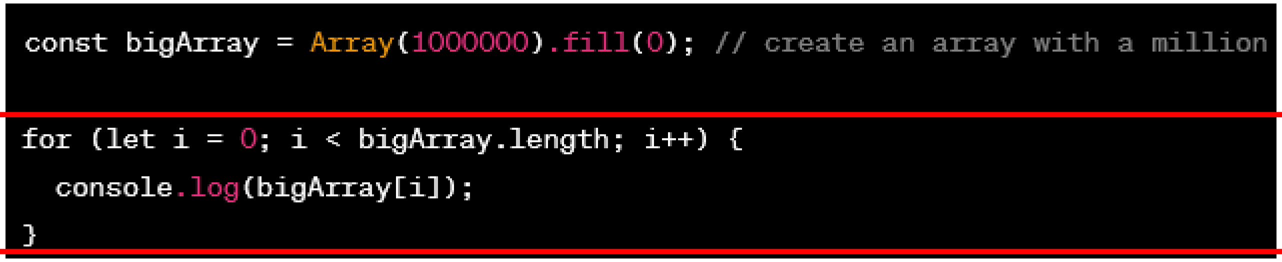
```
function createIterator(array) {
  let nextIndex = 0;

  return {
    // Add the `[Symbol.iterator]` property to the object that we return.
    // The value of this property should be a function that returns the iterator object itself.
    [Symbol.iterator]: function () {
      return this;
    },
    next: function () {
      return nextIndex < array.length
        ? { value: array[nextIndex++], done: false }
        : { done: true };
    },
  };
}

const myArray = [1, 2, 3];

// Call the `createIterator()` function with the `myArray` array as its argument.
// This returns an iterator object, which we store in the `iterator` variable.
const iterator = createIterator(myArray);

// We can now use a `for...of` loop to iterate over the `myArray` array using our custom iterator
for (const value of iterator) {
  console.log(value);
}
```

Knowledge check

What is the difference between **iterable** and **iterators**?

- An iterable is an object **that can be iterated/looped** – Supports for..of loop
- whereas an **iterator is an object that generates the next value** in the iteration sequence.

The problem with processing large arrays in JavaScript



```
const bigArray = Array(1000000).fill(0); // create an array with a million

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

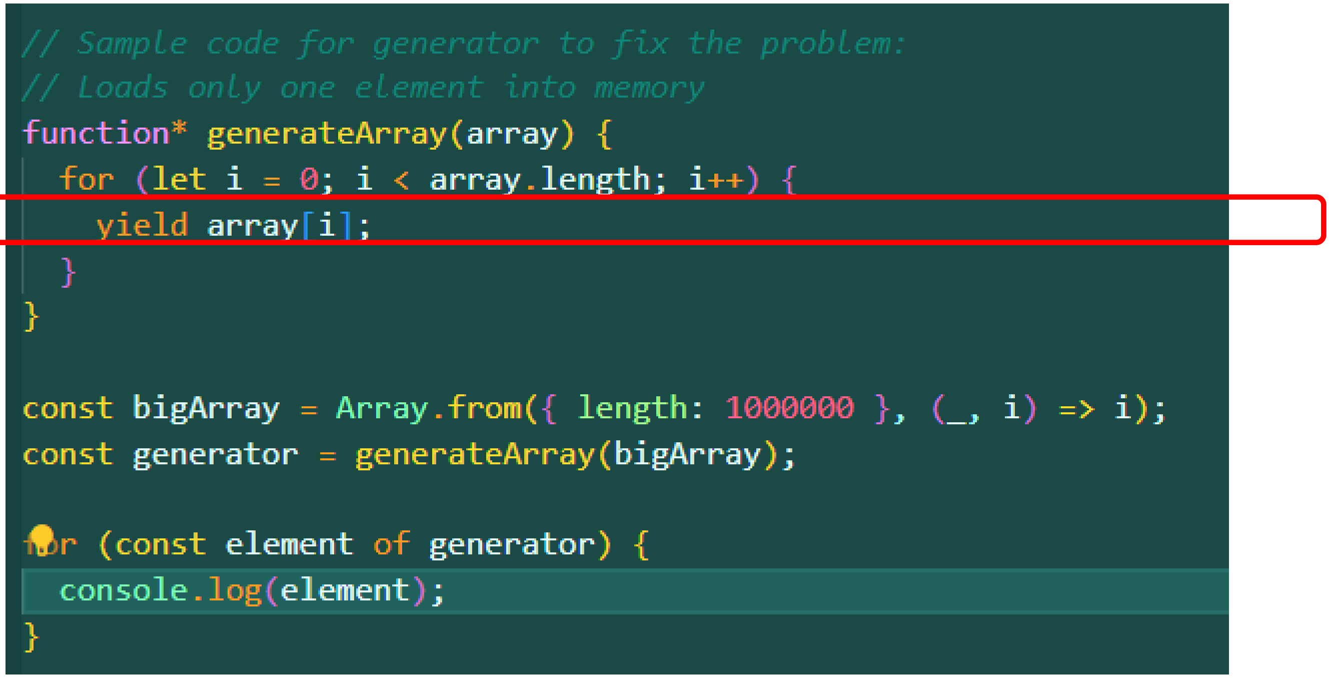
Stays in memory

Entire array is loaded into memory and start executing at once,
This will cause **performance issues** when the array is large

Hence, **Generators**..!

Generators

- A generator is a **special type of function that can be paused and resumed**
- Generator function is called, **it returns an iterator object**
- **Key difference between a generator function and a regular function** is the use of the ***yield*** *keyword*

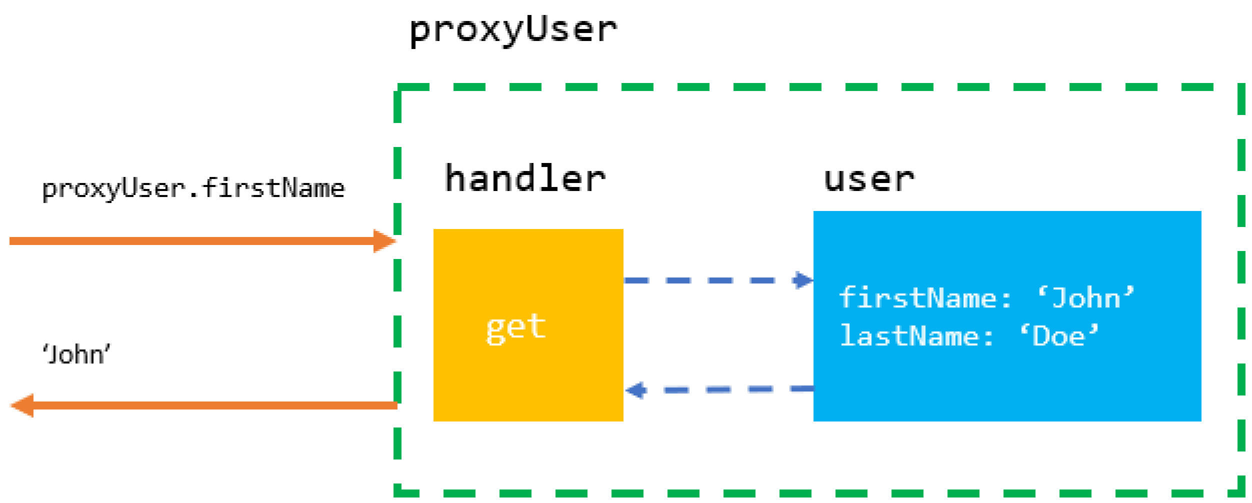


```
// Sample code for generator to fix the problem:
// Loads only one element into memory
function* generateArray(array) {
  for (let i = 0; i < array.length; i++) {
    yield array[i];
  }
}

const bigArray = Array.from({ length: 1000000 }, (_, i) => i);
const generator = generateArray(bigArray);

for (const element of generator) {
  console.log(element);
}
```

yield Keyword pauses the execution, hence one element at a time in the memory

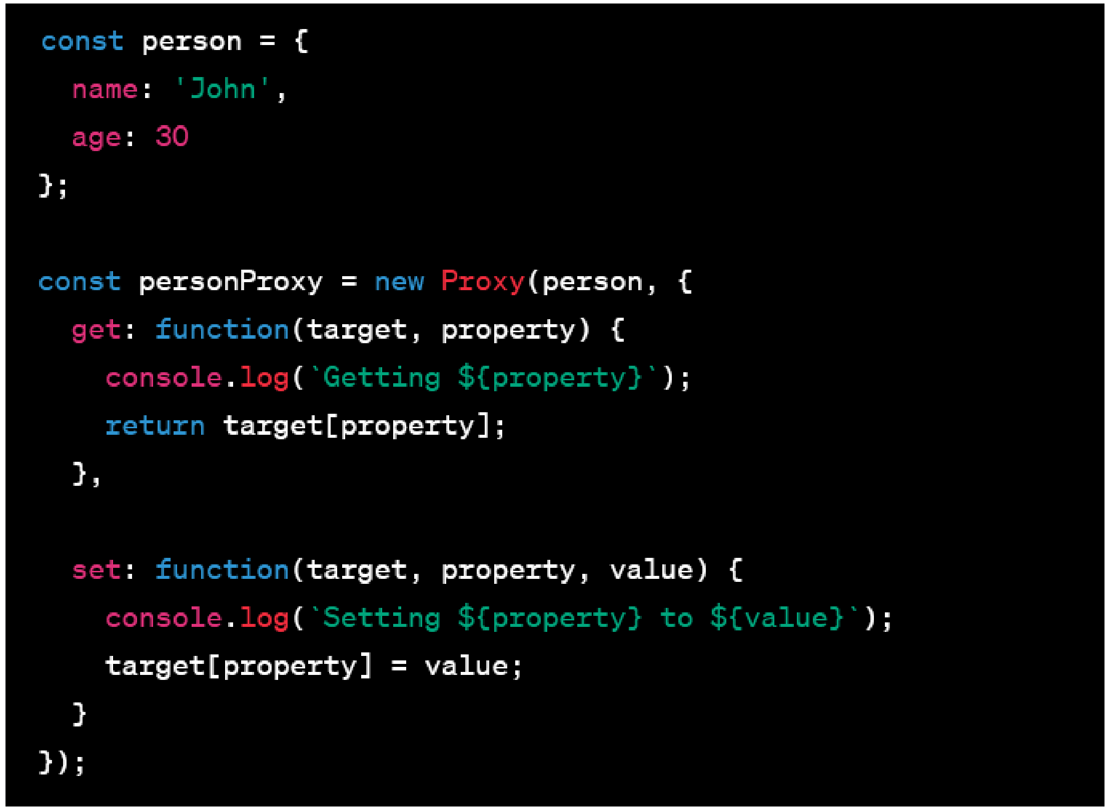
Proxies

Proxy is an object that wraps another object (target) and intercepts the fundamental operations of the target object.



Proxies - Example

Proxy is an object that wraps another object (target) and intercepts the fundamental operations of the target object.



```
const person = {
  name: 'John',
  age: 30
};

const personProxy = new Proxy(person, {
  get: function(target, property) {
    console.log(`Getting ${property}`);
    return target[property];
  },

  set: function(target, property, value) {
    console.log(`Setting ${property} to ${value}`);
    target[property] = value;
  }
});
```

Poly”fill”s

"**fill in" missing JS functionality** in **older browsers** or to add new features to **newer browsers.**

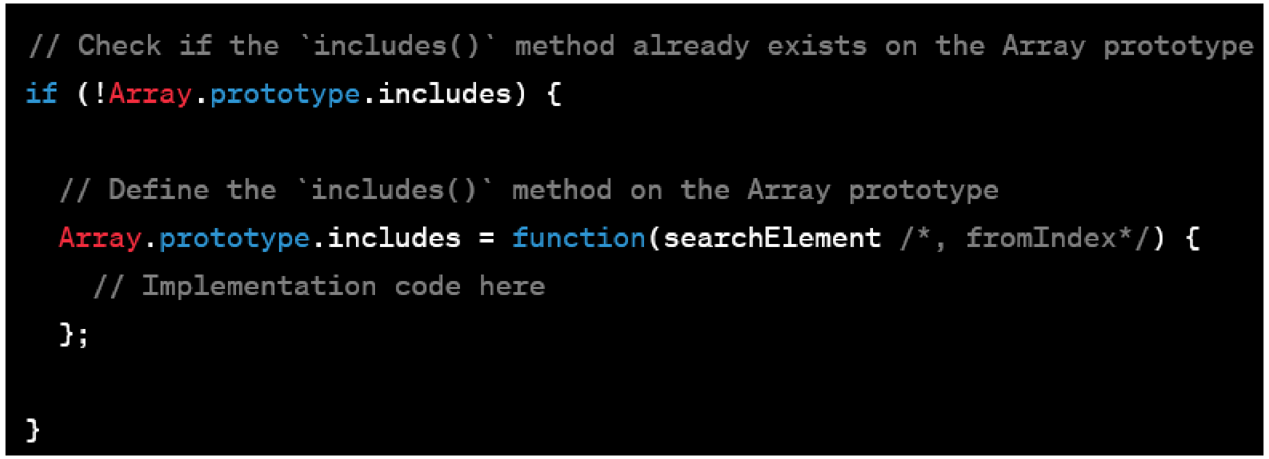
Examples:

- A Browser **doesn’t support promise**, then you **write custom polyfill to support promise**
- A Browser **may not support Array.includes,** then you **write custom JS function on top of Array** prototype

When do we use?

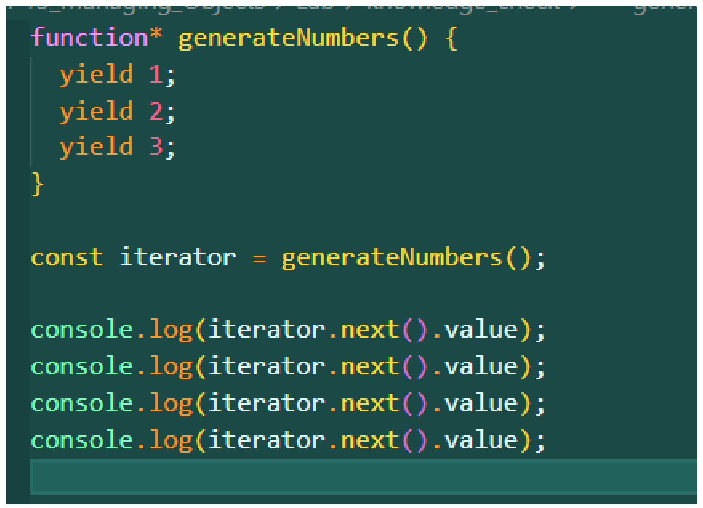
- Use it to **implement consistent behavior across browsers**

Polyfills- Example

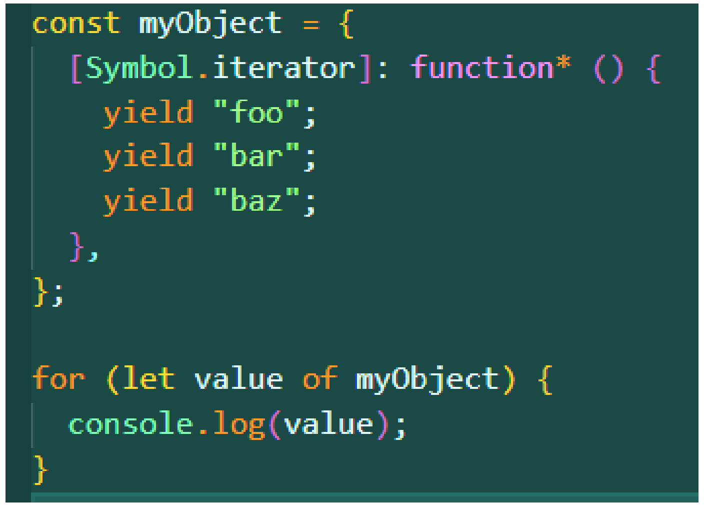


```
// Check if the `includes()` method already exists on the Array prototype
if (!Array.prototype.includes) {

  // Define the `includes()` method on the Array prototype
  Array.prototype.includes = function(searchElement /*, fromIndex*/) {
    // Implementation code here
  };

}
```

Knowledge check: What is the output?



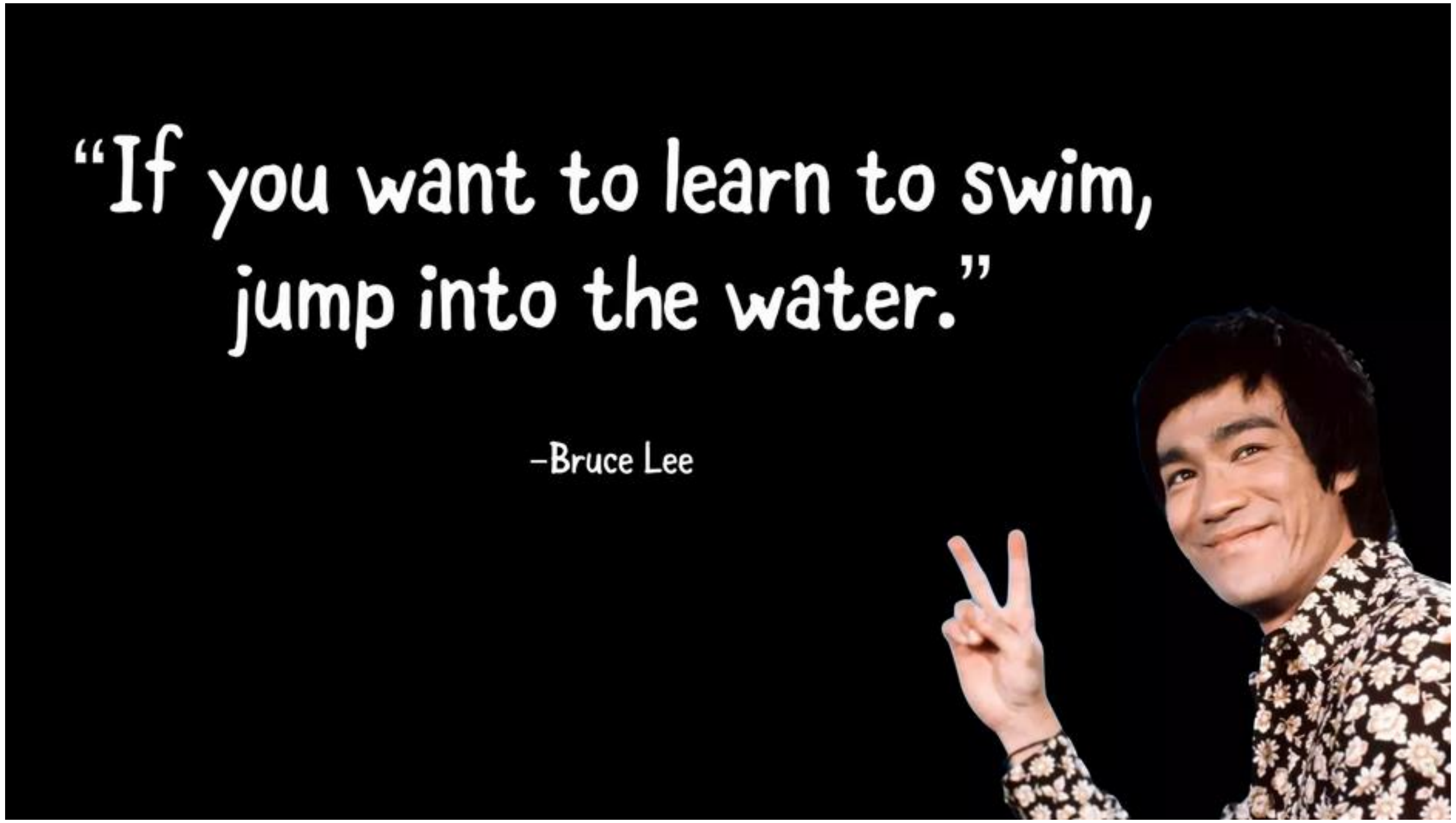
```javascript
function* generateNumbers() {
  yield 1;
  yield 2;
  yield 3;
}

const iterator = generateNumbers();

console.log(iterator.next().value);
console.log(iterator.next().value);
console.log(iterator.next().value);
console.log(iterator.next().value);
```



```javascript
const myObject = {
  [Symbol.iterator]: function* () {
    yield "foo";
    yield "bar";
    yield "baz";
  },
};

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

Output:
1
2
3
undefined

Output:
foo
bar
baz


“If you want to learn to swim,
jump into the water.”
–Bruce Lee

Q & A