The **`for...of` statement** creates a loop iterating over iterable objects, including: built-in `String`, `Array`, array-like objects (e.g., `arguments` or `NodeList`), `TypedArray`, `Map`, `Set`, and user-defined iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

Syntax

```
for (variable of iterable) {
  statement
}
```

`variable`
On each iteration a value of a different property is assigned to *`variable`*. *`variable`* may be declared with `const`, `let`, or `var`.

`iterable`
Object whose iterable properties are iterated.

Examples

Iterating over an `Array`

```
const iterable = [10, 20, 30];

for (const value of iterable) {
  console.log(value);
}
// 10
// 20
// 30
```

You can use `let` instead of `const` too, if you reassign the variable inside the block.

```
const iterable = [10, 20, 30];

for (let value of iterable) {
  value += 1;
  console.log(value);
}
// 11
// 21
// 31
```

Iterating over a `String`

```
const iterable = 'boo';

for (const value of iterable) {
  console.log(value);
}
// "b"
// "o"
// "o"
```

Iterating over a `TypedArray`

```
const iterable = new Uint8Array([0x00, 0xff]);

for (const value of iterable) {
  console.log(value);
}
// 0
// 255
```

Iterating over a `Map`

```
const iterable = new Map([['a', 1], ['b', 2], ['c', 3]]);

for (const entry of iterable) {
  console.log(entry);
}
// ['a', 1]
// ['b', 2]
// ['c', 3]

for (const [key, value] of iterable) {
  console.log(value);
}
// 1
// 2
// 3
```

Iterating over a `Set`

```
const iterable = new Set([1, 1, 2, 2, 3, 3]);

for (const value of iterable) {
  console.log(value);
}
// 1
// 2
// 3
```

Iterating over the arguments object

You can iterate over the `arguments` object to examine all of the parameters passed into a JavaScript function:

```
(function() {
  for (const argument of arguments) {
    console.log(argument);
  }
```

```
})(1, 2, 3);

// 1
// 2
// 3
```

Iterating over a DOM collection

Iterating over DOM collections like `NodeList`: the following example adds a `read` class to paragraphs that are direct descendants of an article:

```
// Note: This will only work in platforms that have
// implemented NodeList.prototype[Symbol.iterator]
const articleParagraphs = document.querySelectorAll('article > p');

for (const paragraph of articleParagraphs) {
  paragraph.classList.add('read');
}
```

Closing iterators

In `for...of` loops, abrupt iteration termination can be caused by `break`, `throw` or `return`. In these cases, the iterator is closed.

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

for (const o of foo()) {
  console.log(o);
  break; // closes iterator, execution continues outside of the loop
}
console.log('done');
```

Iterating over generators

You can also iterate over generators, i.e. functions generating an iterable object:

```
function* fibonacci() { // a generator function
  let [prev, curr] = [0, 1];
  while (true) {
    [prev, curr] = [curr, prev + curr];
    yield curr;
  }
}

for (const n of fibonacci()) {
  console.log(n);
  // truncate the sequence at 1000
  if (n >= 1000) {
    break;
  }
}
```

Do not reuse generators

Generators should not be re-used, even if the `for...of` loop is terminated early, for example via the `break` keyword. Upon exiting a loop, the generator is closed and trying to iterate over it again does not yield any further results.

```
const gen = (function *(){
  yield 1;
  yield 2;
  yield 3;
})();
for (const o of gen) {
  console.log(o);
  break;  // Closes iterator
}

// The generator should not be re-used, the following does not make sen
for (const o of gen) {
  console.log(o); // Never called.
```

```
}
```

Iterating over other iterable objects

You can also iterate over an object that explicitly implements the iterable protocol:

```
const iterable = {
  [Symbol.iterator]() {
    return {
      i: 0,
      next() {
        if (this.i < 3) {
          return { value: this.i++, done: false };
        }
        return { value: undefined, done: true };
      }
    };
  }
};

for (const value of iterable) {
  console.log(value);
}
// 0
// 1
// 2
```

Difference between `for...of` and `for...in`

Both `for...in` and `for...of` statements iterate over something. The main difference between them is in what they iterate over.

The `for...in` statement iterates over the enumerable properties of an object, in an arbitrary order.

The `for...of` statement iterates over values that the iterable object defines to be iterated over.

The following example shows the difference between a `for...of` loop and a `for...in` loop when used with an `Array`.

```
Object.prototype.objCustom = function() {};
Array.prototype.arrCustom = function() {};

const iterable = [3, 5, 7];
iterable.foo = 'hello';

for (const i in iterable) {
  console.log(i); // logs 0, 1, 2, "foo", "arrCustom", "objCustom"
}

for (const i in iterable) {
  if (iterable.hasOwnProperty(i)) {
    console.log(i); // logs 0, 1, 2, "foo"
  }
}

for (const i of iterable) {
  console.log(i); // logs 3, 5, 7
}
```

Let us look into the above code step by step.

```
Object.prototype.objCustom = function() {};
Array.prototype.arrCustom = function() {};

const iterable = [3, 5, 7];
iterable.foo = 'hello';
```

Every object will inherit the `objCustom` property and every object that is an `Array` will inherit the `arrCustom` property since these properties have been added to `Object.prototype` and `Array.prototype`, respectively. The object `iterable` inherits the properties `objCustom` and `arrCustom` because of inheritance and the prototype chain.

```
for (const i in iterable) {
  console.log(i); // logs 0, 1, 2, "foo", "arrCustom", "objCustom"
}
```

This loop logs only enumerable properties of the `iterable` object, in arbitrary order. It doesn't log array **elements** `3`, `5`, `7` or `hello` because those are **not** enumerable properties, in fact they are not properties at all, they are **values**. It logs array **indexes** as well as `arrCustom` and `objCustom`, which are. If you're not sure why these properties are iterated over, there's a more thorough explanation of how `array iteration and for...in` work.

```
for (const i in iterable) {
  if (iterable.hasOwnProperty(i)) {
    console.log(i); // logs 0, 1, 2, "foo"
  }
}
```

This loop is similar to the first one, but it uses `hasOwnProperty()` to check if the found enumerable property is the object's own, i.e. not inherited. If it is, the property is logged. Properties `0`, `1`, `2` and `foo` are logged because they are own properties (**not inherited**). Properties `arrCustom` and `objCustom` are not logged because they **are inherited**.

```
for (const i of iterable) {
  console.log(i); // logs 3, 5, 7
}
```

This loop iterates and logs **values** that `iterable`, as an iterable object, defines to be iterated over. The object's **elements** `3`, `5`, `7` are shown, but none of the object's **properties**.