Iterators and generators

Iterators and Generators bring the concept of iteration directly into the core language and provide a mechanism for customizing the behavior of `for...of` loops.

Iterators

In JavaScript an **iterator** is an object which defines a sequence and potentially a return value upon its termination.

Specifically, an iterator is any object which implements the Iterator protocol by having a `next()` method that returns an object with two properties:

`value` - The next value in the iteration sequence.

`done` - This is `true` if the last value in the sequence has already been consumed. If `value` is present alongside done, it is the iterator's return value.

Once created, an iterator object can be iterated explicitly by repeatedly calling `next()`. Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once. After a terminating value has been yielded additional calls to `next()` should continue to return `{done: true}`.

The most common iterator in JavaScript is the Array iterator, which returns each value in the associated array in sequence.

While it is easy to imagine that all iterators could be expressed as arrays, this is not true. Arrays must be allocated in their entirety, but iterators are consumed only as necessary. Because of this, iterators can express sequences of unlimited size, such as the range of integers between `0` and `Infinity`.

Here is an example which can do just that. It allows creation of a simple range iterator which defines a sequence of integers from `start` (inclusive) to `end` (exclusive) spaced `step` apart. Its final return value is the size of the sequence it created, tracked by the variable `iterationCount`.

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
  let nextIndex = start;
  let iterationCount = 0;

  const rangeIterator = {
    next() {
      let result;
      if (nextIndex < end) {
        result = { value: nextIndex, done: false };
        nextIndex += step;
        iterationCount++;
        return result;
      }
      return { value: iterationCount, done: true };
    },
  };
  return rangeIterator;
}
```

Using the iterator then looks like this:

```
const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
  console.log(result.value); // 1 3 5 7 9
  result = it.next();
}

console.log("Iterated over sequence of size:", result.value); // [5 numbers returned,
that took interval in between: 0 to 10]
```

It is not possible to know reflectively whether a particular object is an iterator. If you need to do this, use Iterables.

Generator functions

While custom iterators are a useful tool, their creation requires careful programming due to the need to explicitly maintain their internal state. **Generator functions** provide a powerful alternative: they allow you to define an iterative algorithm by writing a single function whose execution is not continuous. Generator functions are written using the `function*` syntax.

When called, generator functions do not initially execute their code. Instead, they return a special type of iterator, called a **Generator**. When a value is consumed by calling the generator's `next` method, the Generator function executes until it encounters the `yield` keyword.

The function can be called as many times as desired, and returns a new Generator each time. Each Generator may only be iterated once.

We can now adapt the example from above. The behavior of this code is identical, but the implementation is much easier to write and read.

```
function* makeRangeIterator(start = 0, end = Infinity, step = 1) {
  let iterationCount = 0;
  for (let i = start; i < end; i += step) {
    iterationCount++;
    yield i;
  }
  return iterationCount;
}
```

Iterables

An object is **iterable** if it defines its iteration behavior, such as what values are looped over in a `for...of` construct. Some built-in types, such as `Array` or `Map`, have a default iteration behavior, while other types (such as `Object`) do not.

In order to be iterable, an object must implement the **@@iterator** method. This means that the object (or one of the objects up its prototype chain) must have a property with a `Symbol.iterator` key.

It may be possible to iterate over an iterable more than once, or only once. It is up to the programmer to know which is the case. Iterables which can iterate only once (such as Generators) customarily return `this` from their @@iterator method, whereas iterables which can be iterated many times must return a new iterator on each invocation of @@iterator.

```
function* makeIterator() {
  yield 1;
  yield 2;
}
```

```
const it = makeIterator();

for (const itItem of it) {
  console.log(itItem);
}

console.log(it[Symbol.iterator]() === it); // true

// This example show us generator(iterator) is iterable object,
// which has the @@iterator method return the it (itself),
// and consequently, the it object can iterate only _once_.

// If we change it's @@iterator method to a function/generator
// which returns a new iterator/generator object, (it)
// can iterate many times

it[Symbol.iterator] = function* () {
  yield 2;
  yield 1;
};
```

User-defined iterables

You can make your own iterables like this:

```
const myIterable = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  },
};
```

User-defined iterables can be used in `for...of` loops or the spread syntax as usual.

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

[...myIterable]; // [1, 2, 3]
```

Built-in iterables

`String`, `Array`, `TypedArray`, `Map` and `Set` are all built-in iterables, because their prototype objects all have a `Symbol.iterator` method.

Syntaxes expecting iterables

Some statements and expressions expect iterables. For example: `the for-of` loops, `yield*.`

```
for (const value of ["a", "b", "c"]) {
  console.log(value);
}
// "a"
// "b"
// "c"

[..."abc"];
// ["a", "b", "c"]

function* gen() {
  yield* ["a", "b", "c"];
}

gen().next();
// { value: "a", done: false }

[a, b, c] = new Set(["a", "b", "c"]);
a;
// "a"
```

Advanced generators

Generators compute their `yield`ed values on *demand*, which allows them to efficiently represent sequences that are expensive to compute (or even infinite sequences, as demonstrated above).

The `next()` method also accepts a value, which can be used to modify the internal state of the generator. A value passed to `next()` will be received by `yield`.

A value passed to the first invocation of `next()` is always ignored.

Here is the fibonacci generator using `next(x)` to restart the sequence:

```
function* fibonacci() {
  let current = 0;
  let next = 1;
  while (true) {
    const reset = yield current;
    [current, next] = [next, next + current];
    if (reset) {
      current = 0;
      next = 1;
    }
  }
}

const sequence = fibonacci();
console.log(sequence.next().value); // 0
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
console.log(sequence.next().value); // 3
console.log(sequence.next().value); // 5
console.log(sequence.next().value); // 8
console.log(sequence.next(true).value); // 0
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
```

You can force a generator to throw an exception by calling its `throw()` method and passing the exception value it should throw. This exception will be thrown from the current suspended context of the generator, as if the yield that is currently suspended were instead a `throw value` statement.

If the exception is not caught from within the generator, it will propagate up through the call to `throw()`, and subsequent calls to `next()` will result in the done property being true.

Generators have a `return(value)` method that returns the given value and finishes the generator itself.