Exploring

ES2018 and ES2019

```
// ECMAScript 2018
for await (const x of asyncIterable) {
  console.log(x);
}
const {foo, ...rest} = obj;
const newObj = {...oldObj, qux: 4};

const RE_YM = /(?<year>[0-9]{4})-(?<month>[0-9]{2})/;
/^\p{White_Space}+$/u.test('\t \n\r'); // true
const RE_DOLLAR_PREFIX = /(?<=\$)foo/g;
const RE_NO_DOLLAR_PREFIX = /(?<!\$)foo/g;
/^.$/s.test('\n'); // true

promise.then(result => {/*···*/})
.finally(() => {/*···*/});

// ECMAScript 2019
const errors = results.flatMap(
  result => result.error ? [result.error] : []);

assert.deepEqual(
  Object.fromEntries([['foo',1], ['bar',2]]),
  { foo: 1, bar: 2 });

'  abc  '.trimStart()
```

Dr. Axel Rauschmayer

Contents

Part I

Background

Chapter 1

About this book

This book is about two versions of JavaScript:

- ECMAScript 2018 and
- ECMAScript 2019

It only covers what's new in those versions. For information on other versions, consult my other books, which are free to read online, at `ExploringJS.com`[1].

1.1 Feedback and corrections

There is a link at the end of each chapter of the online version of this book[2] that enables you to comment on that chapter.

Generated: 2019-02-01 16:16

[1] http://exploringjs.com/
[2] http://exploringjs.com/es2018-es2019/toc.html

Chapter 2

About the author

Dr. Axel Rauschmayer has been programming since 1985 and developing web applications since 1995. In 1999, he was technical manager at a German Internet startup that later expanded internationally. In 2006, he held his first talk on Ajax.

Axel specializes in JavaScript, as blogger, book author and trainer. He has done extensive research into programming language design and has followed the state of JavaScript since its creation. He started blogging about ECMAScript 6 in early 2011.

Chapter 3

The TC39 process for ECMAScript features

This chapter explains the so-called *TC39 process*, which governs how ECMAScript features are designed, starting with ECMAScript 2016 (ES7).

3.1 Who designs ECMAScript?

Answer: TC39 (Technical Committee 39).

TC39[1] is the committee that evolves JavaScript. Its members are companies (among others, all major browser vendors). TC39 meets regularly[2], its meetings are attended by delegates that members send and by invited experts. Minutes of the meetings are available online[3] and give you a good idea of how TC39 works.

Occasionally (even in this book), you'll see the term *TC39 member* referring to a human. Then it means: a delegate sent by a TC39 member company.

It is interesting to note that TC39 operates by consensus: Decisions require that a large majority agrees and nobody disagrees strongly enough to veto. For many members, agreements lead to real obligations (they'll have to implement features etc.).

3.2 How is ECMAScript designed?

3.2.1 Problem: ECMAScript 2015 (ES6) was too large a release

The most recent release of ECMAScript, ES6, is large and was standardized almost 6 years after ES5 (December 2009 vs. June 2015). There are two main problems with so much time passing between releases:

[1] http://www.ecma-international.org/memento/TC39.htm
[2] http://www.ecma-international.org/memento/TC39-M.htm
[3] https://github.com/tc39/tc39-notes

- Features that are ready sooner than the release have to wait until the release is finished.
- Features that take long are under pressure to be wrapped up, because postponing them until the next release would mean a long wait. Such features may also delay a release.

Therefore, starting with ECMAScript 2016 (ES7), releases will happen more frequently and be much smaller as a consequence. There will be one release per year and it will contain all features that are finished by a yearly deadline.

3.2.2 Solution: the TC39 process

Each proposal for an ECMAScript feature goes through the following *maturity stages*, starting with stage 0. The progression from one stage to the next one must be approved by TC39.

3.2.2.1 Stage 0: strawman

What is it? A free-form way of submitting ideas for evolving ECMAScript. Submissions must come either from a TC39 member or a non-member who has registered as a TC39 contributor[4].

What's required? The document must be reviewed at a TC39 meeting (source[5]) and is then added to the page with stage 0 proposals[6].

3.2.2.2 Stage 1: proposal

What is it? A formal proposal for the feature.

What's required? A so-called *champion* must be identified who is responsible for the proposal. Either the champion or a co-champion must be a member of TC39 (source[7]). The problem solved by the proposal must be described in prose. The solution must be described via examples, an API and a discussion of semantics and algorithms. Lastly, potential obstacles for the proposal must be identified, such as interactions with other features and implementation challenges. Implementation-wise, polyfills and demos are needed.

What's next? By accepting a proposal for stage 1, TC39 declares its willingness to examine, discuss and contribute to the proposal. Going forward, major changes to the proposal are expected.

3.2.2.3 Stage 2: draft

What is it? A first version of what will be in the specification. At this point, an eventual inclusion of the feature in the standard is likely.

What's required? The proposal must now additionally have a formal description of the syntax and semantics of the feature (using the formal language of the ECMAScript specification). The description should be as complete as possible, but can contain todos and placeholders. Two experimental implementations of the feature are needed, but one of them can be in a transpiler such as Babel.

[4] http://www.ecma-international.org/memento/contribute_TC39_Royalty_Free_Task_Group.php
[5] https://github.com/tc39/ecma262/blob/master/FAQ.md
[6] https://github.com/tc39/proposals/blob/master/stage-0-proposals.md
[7] https://github.com/tc39/ecma262/blob/master/FAQ.md

What's next? Only incremental changes are expected from now on.

3.2.2.4 Stage 3: candidate

What is it? The proposal is mostly finished and now needs feedback from implementations and users to progress further.

What's required? The spec text must be complete. Designated reviewers (appointed by TC39, not by the champion) and the ECMAScript spec editor must sign off on the spec text. There must be at least two spec-compliant implementations (which don't have to be enabled by default).

What's next? Henceforth, changes should only be made in response to critical issues raised by the implementations and their use.

3.2.2.5 Stage 4: finished

What is it? The proposal is ready to be included in the standard.

What's required? The following things are needed before a proposal can reach this stage:

- Test 262[8] acceptance tests (roughly, unit tests for the language feature, written in JavaScript).
- Two spec-compliant shipping implementations that pass the tests.
- Significant practical experience with the implementations.
- The ECMAScript spec editor must sign off on the spec text.

What's next? The proposal will be included in the ECMAScript specification as soon as possible. When the spec goes through its yearly ratification as a standard, the proposal is ratified as part of it.

3.3 A word on ECMAScript versions

Note that since the TC39 process[9] was instituted, the importance of ECMAScript versions has much decreased. What really matters now is what stage a proposed feature is in: Once it has reached stage 4, it can be used safely. But even then, you still have to check if your engines of choice support it.

3.4 FAQ: TC39 process

3.4.1 How is [my favorite proposed feature] doing?

If you are wondering what stages various proposed features are in, consult the readme of the ECMA-262 GitHub repository[10].

[8] https://github.com/tc39/test262
[9] http://exploringjs.com/es2016-es2017/ch_tc39-process.html
[10] https://github.com/tc39/proposals/blob/master/README.md

3.4.2 Is there an official list of ECMAScript features?

Yes, the TC39 repo lists finished proposals[11] and mentions in which ECMAScript versions they are introduced.

3.5 Further reading

The following were important sources of this chapter:

- The TC39 process document[12]
- Information on the ES6 design process: section "How ECMAScript 6 was designed[13]" in "Exploring ES6"

[11] https://github.com/tc39/proposals/blob/master/finished-proposals.md
[12] https://tc39.github.io/process-document/
[13] http://exploringjs.com/es6/ch_about-es6.html#_how-ecmascript-6-was-designed

Chapter 4

An overview of ES2018 and ES2019

4.1 ECMAScript 2018

Major new features:

- Asynchronous Iteration (Domenic Denicola, Kevin Smith)
- Rest/Spread Properties (Sebastian Markbåge)

New regular expression features:

- RegExp named capture groups (Gorkem Yakin, Daniel Ehrenberg)
- RegExp Unicode Property Escapes (Mathias Bynens)
- RegExp Lookbehind Assertions (Gorkem Yakin, Nozomu Katō, Daniel Ehrenberg)
- `s` (`dotAll`) flag for regular expressions (Mathias Bynens)

Other new features:

- `Promise.prototype.finally()` (Jordan Harband)
- Template Literal Revision (Tim Disney)

4.2 ECMAScript 2019

Major new features:

- `Array.prototype.{flat,flatMap}` (Michael Ficarra, Brian Terlson, Mathias Bynens)
- `Object.fromEntries` (Darien Maillet Valentine)

Minor new features:

- `String.prototype.{trimStart,trimEnd}` (Sebastian Markbåge)
- `Symbol.prototype.description` (Michael Ficarra)
- Optional catch binding (Michael Ficarra)
- `Array.prototype.sort()` is guaranteed to be stable (Mathias Bynens).

Changes that are mostly internal:

- Well-formed `JSON.stringify` (Richard Gibson)
- JSON superset (Richard Gibson)
- `Function.prototype.toString` revision (Michael Ficarra)

Part II

ECMAScript 2018

Chapter 5

Asynchronous iteration

Contents

This chapter explains the ECMAScript proposal "Asynchronous Iteration[1]" by Domenic Denicola and Kevin Smith.

5.1 Asynchronous iteration

With ECMAScript 6, JavaScript got built-in support for synchronously iterating over data. But what about data that is delivered asynchronously? For example, lines of text, read asynchronously from a file or an HTTP connection.

This proposal brings support for that kind of data. Before we go into it, let's first recap synchronous iteration.

5.1.1 Synchronous iteration

Synchronous iteration was introduced with ES6 and works as follows:

- Iterable: an object that signals that it can be iterated over, via a method whose key is `Symbol.iterator`.
- Iterator: an object returned by invoking `[Symbol.iterator]()` on an iterable. It wraps each iterated element in an object and returns it via its method `next()` – one at a time.
- IteratorResult: an object returned by `next()`. Property `value` contains an iterated element, property `done` is `true` *after* the last element (`value` can usually be ignored then; it's almost always `undefined`).

I'll demonstrate via an Array:

```
> const iterable = ['a', 'b'];
> const iterator = iterable[Symbol.iterator]();
> iterator.next()
{ value: 'a', done: false }
> iterator.next()
{ value: 'b', done: false }
> iterator.next()
{ value: undefined, done: true }
```

5.1.2 Asynchronous iteration

The problem is that the previously explained way of iterating is *synchronous*, it doesn't work for asynchronous sources of data. For example, in the following code, `readLinesFromFile()` cannot deliver its asynchronous data via synchronous iteration:

```
for (const line of readLinesFromFile(fileName)) {
    console.log(line);
}
```

The proposal specifies a new protocol for iteration that works asynchronously:

- Async iterables are marked via `Symbol.asyncIterator`.

[1] https://github.com/tc39/proposal-async-iteration

- Method `next()` of an async iterator returns Promises for IteratorResults (vs. IteratorResults directly).

You may wonder whether it would be possible to instead use a synchronous iterator that returns one Promise for each iterated element. But that is not enough – whether or not iteration is `done` is generally determined asynchronously.

Using an asynchronous iterable looks as follows. Function `createAsyncIterable()` is explained later. It converts its synchronously iterable parameter into an async iterable.

```
const asyncIterable = createAsyncIterable(['a', 'b']);
const asyncIterator = asyncIterable[Symbol.asyncIterator]();
asyncIterator.next()
.then(iterResult1 => {
    console.log(iterResult1); // { value: 'a', done: false }
    return asyncIterator.next();
})
.then(iterResult2 => {
    console.log(iterResult2); // { value: 'b', done: false }
    return asyncIterator.next();
})
.then(iterResult3 => {
    console.log(iterResult3); // { value: undefined, done: true }
});
```

Within an asynchronous function, you can process the results of the Promises via `await` and the code becomes simpler:

```
async function f() {
    const asyncIterable = createAsyncIterable(['a', 'b']);
    const asyncIterator = asyncIterable[Symbol.asyncIterator]();
    console.log(await asyncIterator.next());
        // { value: 'a', done: false }
    console.log(await asyncIterator.next());
        // { value: 'b', done: false }
    console.log(await asyncIterator.next());
        // { value: undefined, done: true }
}
```

5.1.3 The interfaces for async iteration

In TypeScript notation, the interfaces look as follows.

```
interface AsyncIterable {
    [Symbol.asyncIterator]() : AsyncIterator;
}
interface AsyncIterator {
    next() : Promise<IteratorResult>;
}
interface IteratorResult {
    value: any;
```

```
    done: boolean;
}
```

5.2 `for-await-of`

The proposal also specifies an asynchronous version of the `for-of` loop[2]: `for-await-of`:

```
async function f() {
    for await (const x of createAsyncIterable(['a', 'b'])) {
        console.log(x);
    }
}
// Output:
// a
// b
```

5.2.1 `for-await-of` and rejections

Similarly to how `await` works in async functions, the loop throws an exception if `next()` returns a rejection:

```
function createRejectingIterable() {
    return {
        [Symbol.asyncIterator]() {
            return this;
        },
        next() {
            return Promise.reject(new Error('Problem!'));
        },
    };
}
(async function () { // (A)
    try {
        for await (const x of createRejectingIterable()) {
            console.log(x);
        }
    } catch (e) {
        console.error(e);
            // Error: Problem!
    }
})(); // (B)
```

Note that we have just used an Immediately Invoked Async Function Expression (IIAFE, pronounced "yaffee"). It starts in line (A) and ends in line (B). We need to do that because `for-of-await` doesn't work at the top level of modules and scripts. It does work everywhere where `await` can be used. Namely, in async functions and async generators (which are explained later).

[2] http://exploringjs.com/es6/ch_for-of.html

5.2.2 `for-await-of` and synchronous iterables

Synchronous iterables return synchronous iterators, whose method `next()` returns `{value, done}` objects. `for-await-of` handles synchronous iterables by converting them to asynchronous iterables. Each iterated value is converted to a Promise (or left unchanged if it already is a Promise) via `Promise.resolve()`. That is, `for-await-of` works for iterables over Promises and over normal values. The conversion looks like this:

```
const nextResult = Promise.resolve(valueOrPromise)
    .then(x => ({ value: x, done: false }));
```

Two more ways of looking at the conversion are:

- `Iterable<Promise<T>>` becomes `AsyncIterable<T>`
- The following object

  ```
  { value: Promise.resolve(123), done: false }
  ```

  is converted to

  ```
  Promise.resolve({ value: 123, done: false })
  ```

Therefore, the following two statements are roughly similar.

```
for (const x of await Promise.all(syncIterableOverPromises));
for await (const x of syncIterableOverPromises);
```

The second statement is faster, because `Promise.all()` only creates the Promise for the Array after all Promises in `syncIterableOverPromises` are fulfilled. And `for-of` has to await that Promise. In contrast, `for-await-of` starts processing as soon as the first Promise is fulfilled.

5.2.3 Example: `for-await-of` with a sync iterable

Iterating over a sync iterable over Promises:

```
async function main() {
    const syncIterable = [
        Promise.resolve('a'),
        Promise.resolve('b'),
    ];
    for await (const x of syncIterable) {
        console.log(x);
    }
}
main();

// Output:
// a
// b
```

Iterating over a sync iterable over normal values:

```
async function main() {
    for await (const x of ['c', 'd']) {
```

```
        console.log(x);
    }
}
main();

// Output:
// c
// d
```

5.3 Asynchronous generators

Normal (synchronous) generators help with implementing synchronous iterables. Asynchronous generators do the same for asynchronous iterables.

For example, we have previously used the function `createAsyncIterable(syncIterable)` which converts a `syncIterable` into an asynchronous iterable. This is how you would implement this function via an async generator:

```
async function* createAsyncIterable(syncIterable) {
    for (const elem of syncIterable) {
        yield elem;
    }
}
```

Note the asterisk after `function`:

- A **normal function** is turned into a **normal generator** by putting an asterisk after `function`.
- An **async function** is turned into an **async generator** by doing the same.

How do async generators work?

- A normal generator returns a generator object `genObj`. Each invocation `genObj.next()` returns an object `{value,done}` that wraps a yielded value.
- An async generator returns a generator object `genObj`. Each invocation `genObj.next()` returns **a Promise for** an object `{value,done}` that wraps a yielded value.

5.3.1 Queuing `next()` invocations

The JavaScript engine internally queues invocations of `next()` and feeds them to an async generator once it is ready. That is, after calling `next()`, you can call again, right away; you don't have to wait for the Promise it returns to be settled. In most cases, though, you do want to wait for the settlement, because you need the value of `done` in order to decide whether to call `next()` again or not. That's how the `for-await-of` loop works.

Use cases for calling `next()` several times without waiting for settlements include:

Use case: Retrieving Promises to be processed via `Promise.all()`. If you know how many elements there are in an async iterable, you don't need to check `done`.

```
const asyncGenObj = createAsyncIterable(['a', 'b']);
const [{value:v1},{value:v2}] = await Promise.all([
```

```
    asyncGenObj.next(), asyncGenObj.next()
]);
console.log(v1, v2); // a b
```

Use case: Async generators as sinks for data, where you don't always need to know when they are done.

```
const writer = openFile('someFile.txt');
writer.next('hello'); // don't wait
writer.next('world'); // don't wait
await writer.return(); // wait for file to close
```

Acknowledgement: Thanks to [@domenic] and [@zenparsing] for these use cases.

5.3.2 `await` in async generators

You can use `await` and `for-await-of` inside async generators. For example:

```
async function* prefixLines(asyncIterable) {
    for await (const line of asyncIterable) {
        yield '> ' + line;
    }
}
```

One interesting aspect of combining `await` and `yield` is that `await` can't stop `yield` from returning a Promise, but it can stop that Promise from being settled:

```
async function* asyncGenerator() {
    console.log('Start');
    const result = await doSomethingAsync(); // (A)
    yield 'Result: '+result; // (B)
    console.log('Done');
}
```

This is the context in which this code is executed:

- Every asynchronous generator has a queue with Promises to be settled `yield` or `throw`.
- When `.next()` is called, the following steps are taken:
  - It queues a Promise.
  - Unless the async generator is already running, it resumes it and waits for it to be finished. (It may finish via `yield`, `throw`, `return`, `await`.)
  - Then it returns the Promise. Recall that the result of a settled Promise is delivered asynchronously. Therefore, the earliest delivery is during the next tick.

What does this mean for `asyncGenerator()`?

- Execution starts and progresses until line A, when the generator pauses (due to `await`) and execution reverts back to `.next()`, which returns a Promise `P`.
- When the Promise returned by `doSomethingAsync()` is fulfilled, the generator is resumed and fulfills `P` with `result` (via `yield` in line B). Then the generator is suspended.
- When it is resumed via `.next()`, it fulfills the Promise at the front of the queue via an implicit `return undefined` at the end.

That means that line A and B correspond (roughly) to this code:

```
doSomethingAsync()
.then(result => {
    const {resolve} = promiseQueue.dequeue();
    resolve({
        value: 'Result: '+result,
        done: false,
    });
});
```

If you want to dig deeper – this is a *rough approximation* of how async generators work:

```
const BUSY = Symbol('BUSY');
const COMPLETED = Symbol('COMPLETED');
function asyncGenerator() {
  const settlers = [];
  let step = 0;
  return {
    [Symbol.asyncIterator]() {
      return this;
    },
    next() {
      return new Promise((resolve, reject) => {
        settlers.push({resolve, reject});
        this._run();
      });
    }
    _run() {
      setTimeout(() => {
        if (step === BUSY || settlers.length === 0) {
          return;
        }
        const currentSettler = settlers.shift();
        try {
          switch (step) {
            case 0:
              step = BUSY;
              console.log('Start');
              doSomethingAsync()
              .then(result => {
                currentSettler.resolve({
                  value: 'Result: '+result,
                  done: false,
                });
                // We are not busy, anymore
                step = 1;
                this._run();
              })
              .catch(e => currentSettler.reject(e));
              break;
            case 1:
```

```
              console.log('Done');
              currentSettler.resolve({
                value: undefined,
                done: true,
              });
              step = COMPLETED;
              this._run();
              break;
            case COMPLETED:
              currentSettler.resolve({
                value: undefined,
                done: true,
              });
              this._run();
              break;
          }
        }
        catch (e) {
          currentSettler.reject(e);
        }
      }, 0);
    }
  }
}
```

This code assumes that `next()` is always called without arguments. A complete implementation would have to queue arguments, too.

5.3.3 `yield*` in async generators

`yield*` in async generators works analogously to how it works in normal generators – like a recursive invocation:

```
async function* gen1() {
    yield 'a';
    yield 'b';
    return 2;
}
async function* gen2() {
    const result = yield* gen1(); // (A)
        // result === 2
}
```

In line (A), `gen2()` calls `gen1()`, which means that all elements yielded by `gen1()` are yielded by `gen2()`:

```
(async function () {
    for await (const x of gen2()) {
        console.log(x);
    }
})();
```

```
// Output:
// a
// b
```

The operand of `yield*` can be any async iterable. Sync iterables are automatically converted to async iterables, just like with `for-await-of`.

5.3.4 Errors

In normal generators, `next()` can throw exceptions. In async generators, `next()` can reject the Promise it returns:

```
async function* asyncGenerator() {
    // The following exception is converted to a rejection
    throw new Error('Problem!');
}
asyncGenerator().next()
.catch(err => console.log(err)); // Error: Problem!
```

Converting exceptions to rejections is similar to how async functions work.

5.3.5 Async function vs. async generator function

Async function:

- Returns immediately with a Promise.
- That Promise is fulfilled via `return` and rejected via `throw`.

```
(async function () {
    return 'hello';
})()
.then(x => console.log(x)); // hello

(async function () {
    throw new Error('Problem!');
})()
.catch(x => console.error(x)); // Error: Problem!
```

Async generator function:

- Returns immediately with an async iterable.
- Every invocation of `next()` returns a Promise. `yield x` fulfills the "current" Promise with `{value: x, done: false}`. `throw err` rejects the "current" Promise with `err`.

```
async function* gen() {
    yield 'hello';
}
const genObj = gen();
genObj.next().then(x => console.log(x));
    // { value: 'hello', done: false }
```

5.4 Examples

The source code for the examples is available via the repository `async-iter-demo`[3] on GitHub.

5.4.1 Using asynchronous iteration via Babel

The example repo uses `babel-node` to run its code. This is how it configures Babel in its `package.json`:

```
{
  "dependencies": {
    "babel-preset-env": "···",
    "babel-plugin-transform-async-generator-functions": "···",
    ···
  },
  "babel": {
    "presets": [
      [
        "env",
        {
          "targets": {
            "node": "current"
          }
        }
      ]
    ],
    "plugins": [
      "transform-async-generator-functions"
    ]
  },
  ···
}
```

5.4.2 Example: turning an async iterable into an Array

Function `takeAsync()` collects all elements of `asyncIterable` in an Array. I don't use `for-await-of` in this case, I invoke the async iteration protocol manually. I also don't close `asyncIterable` if I'm finished before the iterable is `done`.

```
/**
 * @returns a Promise for an Array with the elements
 * in `asyncIterable`
 */
async function takeAsync(asyncIterable, count=Infinity) {
    const result = [];
    const iterator = asyncIterable[Symbol.asyncIterator]();
    while (result.length < count) {
```

[3] https://github.com/rauschma/async-iter-demo

```
        const {value,done} = await iterator.next();
        if (done) break;
        result.push(value);
    }
    return result;
}
```

This is the test for `takeAsync()`:

```
test('Collect values yielded by an async generator', async function() {
    async function* gen() {
        yield 'a';
        yield 'b';
        yield 'c';
    }

    assert.deepStrictEqual(await takeAsync(gen()), ['a', 'b', 'c']);
    assert.deepStrictEqual(await takeAsync(gen(), 3), ['a', 'b', 'c']);
    assert.deepStrictEqual(await takeAsync(gen(), 2), ['a', 'b']);
    assert.deepStrictEqual(await takeAsync(gen(), 1), ['a']);
    assert.deepStrictEqual(await takeAsync(gen(), 0), []);
});
```

Note how nicely async functions work together with the mocha test framework: for asynchronous tests, the second parameter of `test()` can return a Promise.

5.4.3 Example: a queue as an async iterable

The example repo also has an implementation for an asynchronous queue, called `AsyncQueue`. Its implementation is relatively complex, which is why I don't show it here. This is the test for `AsyncQueue`:

```
test('Enqueue before dequeue', async function() {
    const queue = new AsyncQueue();
    queue.enqueue('a');
    queue.enqueue('b');
    queue.close();
    assert.deepStrictEqual(await takeAsync(queue), ['a', 'b']);
});
test('Dequeue before enqueue', async function() {
    const queue = new AsyncQueue();
    const promise = Promise.all([queue.next(), queue.next()]);
    queue.enqueue('a');
    queue.enqueue('b');
    return promise.then(arr => {
        const values = arr.map(x => x.value);
        assert.deepStrictEqual(values, ['a', 'b']);
    });
});
```

5.4.4 Example: reading text lines asynchronously

Let's implement code that reads text lines asynchronously. We'll do it in three steps.

Step 1: read text data in chunks via the Node.js `ReadStream` API (which is based on callbacks) and push it into an `AsyncQueue` (which was introduced in the previous section).

```
/**
 * Creates an asynchronous ReadStream for the file whose name
 * is `fileName` and feeds it into an AsyncQueue that it returns.
 *
 * @returns an async iterable
 */
function readFile(fileName) {
    const queue = new AsyncQueue();
    const readStream = createReadStream(fileName,
        { encoding: 'utf8', bufferSize: 1024 });
    readStream.on('data', buffer => {
        const str = buffer.toString('utf8');
        queue.enqueue(str);
    });
    readStream.on('end', () => {
        // Signal end of output sequence
        queue.close();
    });
    return queue;
}
```

Step 2: Use `for-await-of` to iterate over the chunks of text and `yield` lines of text.

```
/**
 * Turns a sequence of text chunks into a sequence of lines
 * (where lines are separated by newlines)
 *
 * @returns an async iterable
 */
async function* splitLines(chunksAsync) {
    let previous = '';
    for await (const chunk of chunksAsync) {
        previous += chunk;
        let eolIndex;
        while ((eolIndex = previous.indexOf('\n')) >= 0) {
            const line = previous.slice(0, eolIndex);
            yield line;
            previous = previous.slice(eolIndex+1);
        }
    }
    if (previous.length > 0) {
        yield previous;
    }
}
```

Step 3: combine the two previous functions. We first feed chunks of text into a `queue` via `readFile()` and then convert that `queue` into an async iterable over lines of text via `splitLines()`.

```
/**
 * @returns an async iterable
 */
function readLines(fileName) {
    // `queue` is an async iterable
    const queue = readFile(fileName);
    return splitLines(queue);
}
```

Lastly, this is how you'd use `readLines()` from within a Node.js script:

```
(async function () {
    const fileName = process.argv[2];
    for await (const line of readLines(fileName)) {
        console.log('>', line);
    }
})();
```

5.5 WHATWG Streams are async iterables

WHATWG streams[4] are async iterables, meaning that you can use `for-await-of` to process them:

```
const rs = openReadableStream();
for await (const chunk of rs) {
    ...
}
```

5.6 The specification of asynchronous iteration

The spec introduces several new concepts and entities:

- Two new interfaces[5], `AsyncIterable` and `AsyncIterator`
- New well-known intrinsic objects[6]: `%AsyncGenerator%`, `%AsyncFromSyncIteratorPrototype%`, `%AsyncGeneratorFunction%`, `%AsyncGeneratorPrototype%`, `%AsyncIteratorPrototype%`.
- One new well-known symbol[7]: `Symbol.asyncIterator`

No new global variables are introduced by this feature.

[4] https://github.com/whatwg/streams
[5] https://tc39.github.io/proposal-async-iteration/#sec-common-iteration-interfaces
[6] https://tc39.github.io/proposal-async-iteration/#sec-well-known-intrinsic-objects-patch
[7] https://tc39.github.io/proposal-async-iteration/#sec-well-known-symbols-patch

5.6.1 Async generators

If you want to understand how async generators work, it's best to start with Sect. "AsyncGenerator Abstract Operations[8]". The key to understanding async generators is understanding how queuing works.

Two internal properties of async generator objects play important roles w.r.t. queuing:

- `[[AsyncGeneratorState]]` contains the state the generator is currently in: `"suspendedStart"`, `"suspendedYield"`, `"executing"`, `"completed"` (it is `undefined` before it is fully initialized)
- `[[AsyncGeneratorQueue]]` holds pending invocations of `next/throw/return`. Each queue entry contains two fields:
  - `[[Completion]]`: the parameter of `next()`, `throw()` or `return()` that lead to the entry being enqueued. The type of the completion (`normal`, `throw`, `return`) indicates which method call created the entry and determines what happens after dequeuing.
  - `[[Capability]]`: the `PromiseCapability` of the pending Promise.

The queue is managed mainly via two operations:

- Enqueuing happens via `AsyncGeneratorEnqueue()`. This is the operation that is called by `next()`, `return()` and `throw()`. It adds an entry to the AsyncGeneratorQueue. Then `AsyncGeneratorResumeNext()` is called, but only if the generator's state isn't `"executing"`:
  - Therefore, if a generator calls `next()`, `return()` or `throw()` from inside itself then the effects of that call will be delayed.
  - `await` leads to a suspension of the generator, but its state remains `"executing"`. Hence, it will not be resumed by `AsyncGeneratorEnqueue()`.
- Dequeuing happens via `AsyncGeneratorResumeNext()`. `AsyncGeneratorResumeNext()` is invoked after enqueuing, but also after settling a queued Promise (e.g. via `yield`), because there may now be new queued pending Promises, allowing execution to continue. If the queue is empty, return immediately. Otherwise, the current Promise is the first element of the queue:
  - If the async generator was suspended by `yield`, it is resumed and continues to run. The current Promise is later settled via `AsyncGeneratorResolve()` or `AsyncGeneratorReject()`.
  - If the generator is already completed, this operation calls `AsyncGeneratorResolve()` and `AsyncGeneratorReject()` itself, meaning that all queued pending Promises will eventually be settled.

5.6.2 Async-from-Sync Iterator Objects

To get an async iterator from an object `iterable`, you call `GetIterator(iterable, async)` (`async` is a symbol). If `iterable` doesn't have a method `[Symbol.asyncIterator]()`, `GetIterator()` retrieves a sync iterator via method `iterable[Symbol.iterator]()` and converts it to an async iterator via `CreateAsyncFromSyncIterator()`.

5.6.3 The `for-await-of` loop

`for-await-of` works almost exactly like `for-of`, but there is an `await` whenever the contents of an IteratorResult are accessed. You can see that by looking at Sect. "Runtime Semantics:

[8] https://tc39.github.io/proposal-async-iteration/#sec-asyncgenerator-abstract-operations

ForIn/OfBodyEvaluation[9]". Notably, iterators are closed similarly, via `IteratorClose()`, towards the end of this section.

5.7 Alternatives to async iteration

Let's look at two alternatives to async iteration for processing async data.

5.7.1 Alternative 1: Communicating Sequential Processes (CSP)

The following code demonstrates the CSP library `js-csp`[10]:

```
var csp = require('js-csp');

function* player(name, table) {
  while (true) {
    var ball = yield csp.take(table); // dequeue
    if (ball === csp.CLOSED) {
      console.log(name + ": table's gone");
      return;
    }
    ball.hits += 1;
    console.log(name + " " + ball.hits);
    yield csp.timeout(100); // wait
    yield csp.put(table, ball); // enqueue
  }
}

csp.go(function* () {
  var table = csp.chan(); // (A)

  csp.go(player, ["ping", table]); // (B)
  csp.go(player, ["pong", table]); // (C)

  yield csp.put(table, {hits: 0}); // enqueue
  yield csp.timeout(1000); // wait
  table.close();
});
```

`player` defines a "process" that is instantiated twice (in line (B) and in line (C), via `csp.go()`). The processes are connected via the "channel" `table`, which is created in line (A) and passed to `player` via its second parameter. A channel is basically a queue.

How does CSP compare to async iteration?

- The coding style is also synchronous.

[9]https://tc39.github.io/proposal-async-iteration/#sec-runtime-semantics-forin-div-ofbodyevaluation-lhs-stmt-iterator-lhskind-labelset
[10]https://github.com/ubolonton/js-csp

- Channels feel like a good abstraction for producing and consuming async data.
- Making the connections between processes explicit, as channels, means that you can configure how they work (how much is buffered, when to block, etc.).
- The abstraction "channel" works for many use cases: communication with and between web workers, distributed programming, etc.

5.7.2 Alternative 2: Reactive Programming

The following code demonstrates Reactive Programming via the JavaScript library RxJS[11]:

```
const button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click') // (A)
  .throttle(1000) // at most one event per second
  .scan(count => count + 1, 0)
  .subscribe(count => console.log(`Clicked ${count} times`));
```

In line (A), we create a stream of click events via `fromEvent()`. These events are then filtered so that there is at most one event per second. Every time there is an event, `scan()` counts how many events there have been, so far. In the last line, we log all counts.

How does Reactive Programming compare to async iteration?

- The coding style is not as familiar, but there are similarities to Promises.
- On the other hand, chaining operations (such as `throttle()`) works well for many push-based data sources (DOM events, server-sent events, etc.).
- Async iteration is for pull streams and single consumers. Reactive programming is for push streams and potentially multiple consumers. The former is better suited for I/O and can handle backpressure.

There is an ECMAScript proposal for Reactive Programming, called "Observable[12]" (by Jafar Husain).

5.8 Further reading

- "Streams API FAQ[13]" by Domenic Denicola (explains how streams and asynchronous iteration are related; and more)
- "Why Async Iterators Matter[14]" (slides by Benjamin Gruenbaum)

Background:

- Iterables and iterators[15] (chapter on sync iteration in "Exploring ES6")
- Generators[16] (chapter on sync generators in "Exploring ES6")
- Chapter "Async functions[17]" in "Exploring ES2016 and ES2017"

[11] https://github.com/ReactiveX/RxJS
[12] https://github.com/tc39/proposal-observable
[13] https://github.com/whatwg/streams/blob/master/FAQ.md
[14] https://docs.google.com/presentation/d/1r2V1sLG8JSSk8txiLh4wfTkom-BoOsk52FgPBy8o3RM/
[15] http://exploringjs.com/es6/ch_iteration.html
[16] http://exploringjs.com/es6/ch_generators.html
[17] http://exploringjs.com/es2016-es2017/ch_async-functions.html

Chapter 6

Rest/Spread Properties

Contents

The ECMAScript proposal "Rest/Spread Properties[1]" by Sebastian Markbåge enables:

- The rest operator (`...`) in object destructuring. At the moment, this operator only works for Array destructuring and in parameter definitions.
- The spread operator (`...`) in object literals. At the moment, this operator only works in Array literals and in function and method calls.

6.1 The rest operator (...) in object destructuring

Inside object destructuring patterns, the rest operator (`...`) copies all enumerable own properties of the destructuring source into its operand, except those that were already mentioned in the object literal.

[1] https://github.com/sebmarkbage/ecmascript-rest-spread

```
const obj = {foo: 1, bar: 2, baz: 3};
const {foo, ...rest} = obj;
    // Same as:
    // const foo = 1;
    // const rest = {bar: 2, baz: 3};
```

If you are using object destructuring to handle named parameters, the rest operator enables you to collect all remaining parameters:

```
function func({param1, param2, ...rest}) { // rest operator
    console.log('All parameters: ',
        {param1, param2, ...rest}); // spread operator
    return param1 + param2;
}
```

6.1.1 Syntactic restrictions

Per top level of each object literal, you can use the rest operator at most once and it must appear at the end:

```
const {...rest, foo} = obj; // SyntaxError
const {foo, ...rest1, ...rest2} = obj; // SyntaxError
```

You can, however, use the rest operator several times if you nest it:

```
const obj = {
    foo: {
        a: 1,
        b: 2,
        c: 3,
    },
    bar: 4,
    baz: 5,
};
const {foo: {a, ...rest1}, ...rest2} = obj;
// Same as:
// const a = 1;
// const rest1 = {b: 2, c: 3};
// const rest2 = {bar: 4, baz: 5};
```

6.2 The spread operator (...) in object literals

Inside object literals, the spread operator (...) inserts all enumerable own properties of its operand into the object created via the literal:

```
> const obj = {foo: 1, bar: 2};
> {...obj, baz: 3}
{ foo: 1, bar: 2, baz: 3 }
```

Note that order matters even if property keys don't clash, because objects record insertion order:

```
> {baz: 3, ...obj}
{ baz: 3, foo: 1, bar: 2 }
```

If keys clash, order determines which entry "wins":

```
> const obj = {foo: 1, bar: 2, baz: 3};
> {...obj, foo: true}
{ foo: true, bar: 2, baz: 3 }
> {foo: true, ...obj}
{ foo: 1, bar: 2, baz: 3 }
```

6.3 Common use cases for the object spread operator

In this section, we'll look at things that you can use the spread operator for. I'll also show how to do these things via `Object.assign()`[2], which is very similar to the spread operator (we'll compare them in more detail later).

6.3.1 Cloning objects

Cloning the enumerable own properties of an object `obj`:

```
const clone1 = {...obj};
const clone2 = Object.assign({}, obj);
```

The prototypes of the clones are always `Object.prototype`, which is the default for objects created via object literals:

```
> Object.getPrototypeOf(clone1) === Object.prototype
true
> Object.getPrototypeOf(clone2) === Object.prototype
true
> Object.getPrototypeOf({}) === Object.prototype
true
```

Cloning an object `obj`, including its prototype:

```
const clone1 = {__proto__: Object.getPrototypeOf(obj), ...obj};
const clone2 = Object.assign(
    Object.create(Object.getPrototypeOf(obj)), obj);
```

Note that `__proto__` inside object literals is only a mandatory feature in web browsers, not in JavaScript engines in general.

6.3.2 True clones of objects

Sometimes you need to faithfully copy all own properties of an object `obj` and their attributes (`writable`, `enumerable`, ...), including getters and setters. Then `Object.assign()` and the spread operator don't work. You need to use property descriptors[3]:

[2] http://exploringjs.com/es6/ch_oop-besides-classes.html#Object_assign
[3] http://speakingjs.com/es5/ch17.html#property_attributes

```
const clone1 = Object.defineProperties({},
    Object.getOwnPropertyDescriptors(obj));
```

If you additionally want to preserve the prototype of `obj`, you can use `Object.create()`[4]:

```
const clone2 = Object.create(
    Object.getPrototypeOf(obj),
    Object.getOwnPropertyDescriptors(obj));
```

`Object.getOwnPropertyDescriptors()`[5] is explained in "Exploring ES2016 and ES2017".

6.3.3 Pitfall: cloning is always shallow

Keep in mind that with all the ways of cloning that we have looked at, you only get shallow copies: If one of the original property values is an object, the clone will refer to the same object, it will not be (recursively, deeply) cloned itself:

```
const original = { prop: {} };
const clone = Object.assign({}, original);

console.log(original.prop === clone.prop); // true
original.prop.foo = 'abc';
console.log(clone.prop.foo); // abc
```

6.3.4 Various other use cases

Merging two objects `obj1` and `obj2`:

```
const merged = {...obj1, ...obj2};
const merged = Object.assign({}, obj1, obj2);
```

Filling in defaults for user data:

```
const DEFAULTS = {foo: 'a', bar: 'b'};
const userData = {foo: 1};

const data = {...DEFAULTS, ...userData};
const data = Object.assign({}, DEFAULTS, userData);
    // {foo: 1, bar: 'b'}
```

Specifying the default values for properties `foo` and `bar` inline:

```
const userData = {foo: 1};
const data = {foo: 'a', bar: 'b', ...userData};
const data = Object.assign({}, {foo:'a', bar:'b'}, userData);
    // {foo: 1, bar: 'b'}
```

Non-destructively updating property `foo`:

[4] http://speakingjs.com/es5/ch17.html#Object.create
[5] http://exploringjs.com/es2016-es2017/ch_object-getownpropertydescriptors.html

```
const obj = {foo: 'a', bar: 'b'};
const obj2 = {...obj, foo: 1};
const obj2 = Object.assign({}, obj, {foo: 1});
    // {foo: 1, bar: 'b'}
```

6.4 Spreading objects versus `Object.assign()`

The spread operator and `Object.assign()`[6] are very similar. The main difference is that spreading defines new properties, while `Object.assign()` sets them. What exactly that means is explained later.

6.4.1 The two ways of using `Object.assign()`

There are two ways of using `Object.assign()`:

First, destructively (an existing object is changed):

```
Object.assign(target, source1, source2);
```

Here, `target` is modified; `source1` and `source2` are copied into it.

Second, non-destructively (no existing object is changed):

```
const result = Object.assign({}, source1, source2);
```

Here, a new object is created via an empty object literal and `source1` and `source2` are copied into it. At the end, this new object is returned and assigned to `result`.

The spread operator is very similar to the second way of using `Object.assign()`. Next, we'll look at where the two are similar and where they differ.

6.4.2 Both spread and `Object.assign()` read values via a "get" operation

Both operations use normal "get" operations to read property values from the source, before writing them to the target. As a result, getters are turned into normal data properties during this process.

Let's look at an example:

```
const original = {
    get foo() {
        return 123;
    }
};
```

`original` has the getter `foo` (its *property descriptor*[7] has the properties `get` and `set`):

```
> Object.getOwnPropertyDescriptor(original, 'foo')
{ get: [Function: foo],
  set: undefined,
  enumerable: true,
```

[6] http://exploringjs.com/es6/ch_oop-besides-classes.html#Object_assign
[7] http://speakingjs.com/es5/ch17.html#property_attributes

```
  configurable: true }
```

But it its clones `clone1` and `clone2`, `foo` is a normal data property (its property descriptor has the properties `value` and `writable`):

```
> const clone1 = {...original};
> Object.getOwnPropertyDescriptor(clone1, 'foo')
{ value: 123,
  writable: true,
  enumerable: true,
  configurable: true }

> const clone2 = Object.assign({}, original);
> Object.getOwnPropertyDescriptor(clone2, 'foo')
{ value: 123,
  writable: true,
  enumerable: true,
  configurable: true }
```

6.4.3 Spread defines properties, `Object.assign()` sets them

The spread operator defines new properties in the target, `Object.assign()` uses a normal "set" operation to create them. That has two consequences.

6.4.3.1 Targets with setters

First, `Object.assign()` triggers setters, spread doesn't:

```
Object.defineProperty(Object.prototype, 'foo', {
    set(value) {
        console.log('SET', value);
    },
});
const obj = {foo: 123};
```

The previous piece of code installs a setter `foo` that is inherited by all normal objects.

If we clone `obj` via `Object.assign()`, the inherited setter is triggered:

```
> Object.assign({}, obj)
SET 123
{}
```

With spread, it isn't:

```
> { ...obj }
{ foo: 123 }
```

`Object.assign()` also triggers own setters during copying, it does not overwrite them.

6.4.3.2 Targets with read-only properties

Second, you can stop `Object.assign()` from creating own properties via inherited read-only properties, but not the spread operator:

```
Object.defineProperty(Object.prototype, 'bar', {
    writable: false,
    value: 'abc',
});
```

The previous piece of code installs the read-only property `bar` that is inherited by all normal objects.

As a consequence, you can't use assignment to create the own property `bar`, anymore (you only get an exception in strict mode; in sloppy mode, setting fails silently):

```
> const tmp = {};
> tmp.bar = 123;
TypeError: Cannot assign to read only property 'bar'
```

In the following code, we successfully create the property `bar` via an object literal. This works, because object literals don't set properties, they *define* them:

```
const obj = {bar: 123};
```

However, `Object.assign()` uses assignment for creating properties, which is why we can't clone `obj`:

```
> Object.assign({}, obj)
TypeError: Cannot assign to read only property 'bar'
```

Cloning via the spread operator works:

```
> { ...obj }
{ bar: 123 }
```

6.4.4 Both spread and `Object.assign()` only consider own enumerable properties

Both operations ignore all inherited properties and all non-enumerable own properties.

The following object `obj` inherits one (enumerable!) property from `proto` and has two own properties:

```
const proto = {
    inheritedEnumerable: 1,
};
const obj = Object.create(proto, {
    ownEnumerable: {
        value: 2,
        enumerable: true,
    },
    ownNonEnumerable: {
        value: 3,
        enumerable: false,
    },
});
```

If you clone `obj`, the result only has the property `ownEnumerable`. The properties `inheritedEnumerable` and `ownNonEnumerable` are not copied:

```
> {...obj}
{ ownEnumerable: 2 }
> Object.assign({}, obj)
{ ownEnumerable: 2 }
```

Chapter 7

RegExp named capture groups

Contents

This chapter explains proposal "RegExp Named Capture Groups[1]" by Gorkem Yakin, Daniel Ehrenberg.

Before we get to named capture groups, let's take a look at numbered capture groups; to introduce the idea of capture groups.

7.1 Numbered capture groups

Numbered capture groups enable you to take apart a string with a regular expression.

Successfully matching a regular expression against a string returns a match object `matchObj`. Putting a fragment of the regular expression in parentheses turns that fragment into a *capture group*: the part of the string that it matches is stored in `matchObj`.

Prior to this proposal, all capture groups were accessed by number: the capture group starting with the first parenthesis via `matchObj[1]`, the capture group starting with the second parenthesis via `matchObj[2]`, etc.

For example, the following code shows how numbered capture groups are used to extract year, month and day from a date in ISO format:

```
const RE_DATE = /([0-9]{4})-([0-9]{2})-([0-9]{2})/;
```

[1] https://github.com/tc39/proposal-regexp-named-groups

```
const matchObj = RE_DATE.exec('1999-12-31');
const year = matchObj[1]; // 1999
const month = matchObj[2]; // 12
const day = matchObj[3]; // 31
```

Referring to capture groups via numbers has several disadvantages:

1. Finding the number of a capture group is a hassle: you have to count parentheses.
2. You need to see the regular expression if you want to understand what the groups are for.
3. If you change the order of the capture groups, you also have to change the matching code.

All issues can be somewhat mitigated by defining constants for the numbers of the capture groups. However, capture groups are an all-around superior solution.

7.2 Named capture groups

The proposed feature is about identifying capture groups via names:

```
(?<year>[0-9]{4})
```

Here we have tagged the previous capture group #1 with the name `year`. The name must be a legal JavaScript identifier (think variable name or property name). After matching, you can access the captured string via `matchObj.groups.year`.

The captured strings are not properties of `matchObj`, because you don't want them to clash with current or future properties created by the regular expression API.

Let's rewrite the previous code so that it uses named capture groups:

```
const RE_DATE = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;

const matchObj = RE_DATE.exec('1999-12-31');
const year = matchObj.groups.year; // 1999
const month = matchObj.groups.month; // 12
const day = matchObj.groups.day; // 31
```

Named capture groups also create indexed entries; as if they were numbered capture groups:

```
const year2 = matchObj[1]; // 1999
const month2 = matchObj[2]; // 12
const day2 = matchObj[3]; // 31
```

Destructuring[2] can help with getting data out of the match object:

```
const {groups: {day, year}} = RE_DATE.exec('1999-12-31');
console.log(year); // 1999
console.log(day); // 31
```

Named capture groups have the following benefits:

- It's easier to find the "ID" of a capture group.
- The matching code becomes self-descriptive, as the ID of a capture group describes what is being captured.

[2] http://exploringjs.com/es6/ch_destructuring.html

- You don't have to change the matching code if you change the order of the capture groups.
- The names of the capture groups also make the regular expression easier to understand, as you can see directly what each group is for.

You can freely mix numbered and named capture groups.

7.3 Backreferences

`\k<name>` in a regular expression means: match the string that was previously matched by the named capture group `name`. For example:

```
const RE_TWICE = /^(?<word>[a-z]+)!\k<word>$/;
RE_TWICE.test('abc!abc'); // true
RE_TWICE.test('abc!ab'); // false
```

The backreference syntax for numbered capture groups works for named capture groups, too:

```
const RE_TWICE = /^(?<word>[a-z]+)!\1$/;
RE_TWICE.test('abc!abc'); // true
RE_TWICE.test('abc!ab'); // false
```

You can freely mix both syntaxes:

```
const RE_TWICE = /^(?<word>[a-z]+)!\k<word>!\1$/;
RE_TWICE.test('abc!abc!abc'); // true
RE_TWICE.test('abc!abc!ab'); // false
```

7.4 `replace()` and named capture groups

The string method `replace()` supports named capture groups in two ways.

First, you can mention their names in the replacement string:

```
const RE_DATE = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
console.log('1999-12-31'.replace(RE_DATE,
    '$<month>/$<day>/$<year>'));
    // 12/31/1999
```

Second, each replacement function receives an additional parameter that holds an object with data captured via named groups. For example (line A):

```
const RE_DATE = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
console.log('1999-12-31'.replace(
    RE_DATE,
    (g0,y,m,d,offset,input, {year, month, day}) => // (A)
        month+'/'+day+'/'+year));
    // 12/31/1999
```

These are the parameters of the callback in line A:

- `g0` contains the whole matched substring, `'1999-12-31'`

- `y`, `m`, `d` are matches for the numbered groups 1–3 (which are created via the named groups `year`, `month`, `day`).
- `offset` specifies where the match was found.
- `input` contains the complete input string.
- The last parameter is new and contains one property for each of the three named capture groups `year`, `month` and `day`. We use destructuring to access those properties.

The following code shows another way of accessing the last argument:

```
console.log('1999-12-31'.replace(RE_DATE,
    (...args) => {
        const {year, month, day} = args[args.length-1];
        return month+'/'+day+'/'+year;
    }));
    // 12/31/1999
```

We receive all arguments via the rest parameter `args`. The last element of the Array `args` is the object with the data from the named groups. We access it via the index `args.length-1`.

7.5 Named groups that don't match

If an optional named group does not match, its property is set to `undefined` (but still exists):

```
const RE_OPT_A = /^(?<as>a+)?$/;
const matchObj = RE_OPT_A.exec('');

// We have a match:
console.log(matchObj[0] === ''); // true

// Group <as> didn't match anything:
console.log(matchObj.groups.as === undefined); // true

// But property `as` exists:
console.log('as' in matchObj.groups); // true
```

7.6 Implementations

- The Babel plugin `transform-modern-regexp`[3] by Dmitry Soshnikov supports named capture groups.
- V8 6.0+ has support behind the flag `--harmony_regexp_named_captures`[4].

You can check the version of V8 in your Node.js via:

```
node -p process.versions.v8
```

[3] https://github.com/DmitrySoshnikov/babel-plugin-transform-modern-regexp#named-capturing-groups
[4] https://bugs.chromium.org/p/v8/issues/detail?id=5437

In Chrome Canary (60.0+), you can enable named capture groups as follows. First, look up the path of the Chrome Canary binary via the `about:`[5] URL. Then start Canary like this (you only need the double quotes if the path contains a space):

```
$ alias canary='"/tmp/Google Chrome Canary.app/Contents/MacOS/Google Chrome Canary"'
$ canary --js-flags='--harmony-regexp-named-captures'
```

7.7 Further reading

- Chapter "Regular Expressions[6]" in "Speaking JavaScript"
- Chapter "New regular expression features[7]" in "Exploring ES6"
- Chapter "Destructuring[8]" in "Exploring ES6"

[5] about:
[6] http://speakingjs.com/es5/ch19.html
[7] http://exploringjs.com/es6/ch_regexp.html
[8] http://exploringjs.com/es6/ch_destructuring.html

Chapter 8

RegExp Unicode property escapes

Contents

This chapter explains the proposal "RegExp Unicode Property Escapes[1]" by Mathias Bynens.

8.1 Overview

JavaScript lets you match characters by mentioning the "names" of sets of characters. For example, `\s` stands for "whitespace":

```
> /^\s+$/u.test('\t \n\r')
true
```

The proposal lets you additionally match characters by mentioning their Unicode character properties (what those are is explained next) inside the curly braces of `\p{}`. Two examples:

```
> /^\p{White_Space}+$/u.test('\t \n\r')
true
> /^\p{Script=Greek}+$/u.test('μετά')
true
```

[1] https://github.com/tc39/proposal-regexp-unicode-property-escapes

As you can see, one of the benefits of property escapes is is that they make regular expressions more self-descriptive. Additional benefits will become clear later.

Before we delve into how property escapes work, let's examine what Unicode character properties are.

8.2 Unicode character properties

In the Unicode standard, each character has *properties* – metadata describing it. Properties play an important role in defining the nature of a character. Quoting the Unicode Standard, Sect. 3.3, D3[2]:

> The semantics of a character are determined by its identity, normative properties, and behavior.

8.2.1 Examples of properties

These are a few examples of properties:

- `Name`: a unique name, composed of uppercase letters, digits, hyphens and spaces. For example:
  - A: `Name = LATIN CAPITAL LETTER A`
  - 🙂: `Name = SLIGHTLY SMILING FACE`
- `General_Category`: categorizes characters. For example:
  - x: `General_Category = Lowercase_Letter`
  - $: `General_Category = Currency_Symbol`
- `White_Space`: used for marking invisible spacing characters, such as spaces, tabs and newlines. For example:
  - \t: `White_Space = True`
  - π: `White_Space = False`
- `Age`: version of the Unicode Standard in which a character was introduced. For example: The Euro sign € was added in version 2.1 of the Unicode standard.
  - €: `Age = 2.1`
- `Block`: a contiguous range of code points. Blocks don't overlap and their names are unique. For example:
  - S: `Block = Basic_Latin` (range U+0000..U+007F)
  - 🙂: `Block = Emoticons` (range U+1F600..U+1F64F)
- `Script`: is a collection of characters used by one or more writing systems.
  - Some scripts support several writing systems. For example, the Latin script supports the writing systems English, French, German, Latin, etc.
  - Some languages can be written in multiple alternate writing systems that are supported by multiple scripts. For example, Turkish used the Arabic script before it transitioned to the Latin script in the early 20th century.
  - Examples:
    * α: `Script = Greek`
    * Д: `Script = Cyrillic`

[2]http://www.unicode.org/versions/Unicode9.0.0/ch03.pdf

8.2.2 Types of properties

The following types of properties exist:

- Enumerated property: a property whose values are few and named. `General_Category` is an enumerated property.
- Closed enumerated property: an enumerated property whose set of values is fixed and will not be changed in future versions of the Unicode Standard.
- Boolean property: a closed enumerated property whose values are `True` and `False`. Boolean properties are also called *binary*, because they are like markers that characters either have or not. `White_Space` is a binary property.
- Numeric property: has values that are integers or real numbers.
- String-valued property: a property whose values are strings.
- Catalog property: an enumerated property that may be extended as the Unicode Standard evolves. `Age` and `Script` are catalog properties.
- Miscellaneous property: a property whose values are not Boolean, enumerated, numeric, string or catalog values. `Name` is a miscellaneous property.

8.2.3 Matching properties and property values

Properties and property values are matched as follows:

- Loose matching: case, whitespace, underscores and hyphens are ignored when comparing properties and property values. For example, `"General_Category"`, `"general category"`, `"-general-category-"`, `"GeneralCategory"` are all considered to be the same property.
- Aliases: the data files `PropertyAliases.txt`[3] and `PropertyValueAliases.txt`[4] define alternative ways of referring to properties and property values.
  - Most aliases have long forms and short forms. For example:
    * Long form: `General_Category`
    * Short form: `gc`
  - Examples of property value aliases (per line, all values are considered equal):
    * `Lowercase_Letter`, `Ll`
    * `Currency_Symbol`, `Sc`
    * `True`, `T`, `Yes`, `Y`
    * `False`, `F`, `No`, `N`

8.3 Unicode property escapes for regular expressions

Unicode property escapes look like this:

1. `\p{prop=value}`: Match all characters whose property `prop` has the value `value`.
2. `\P{prop=value}`: Match all characters that do not have a property `prop` whose value is `value`.
3. `\p{bin_prop}`: Match all characters whose binary property `bin_prop` is True.
4. `\P{bin_prop}`: Match all characters whose binary property `bin_prop` is False.

Comments:

[3] http://unicode.org/Public/UNIDATA/PropertyAliases.txt
[4] http://unicode.org/Public/UNIDATA/PropertyValueAliases.txt

- You can only use Unicode property escapes if the flag `/u` is set. Without `/u`, `\p` is the same as `p`.
- Forms (3) and (4) can be used as abbreviations if the property is `General_Category`. For example, `\p{Lowercase_Letter}` is an abbreviation for `\p{General_Category=Lowercase_Letter}`

8.3.1 Details

Things to note:

- Property escapes do not support loose matching. You must use aliases exactly as they are mentioned in `PropertyAliases.txt`[5] and `PropertyValueAliases.txt`[6]
- Implementations must support at least the following Unicode properties and their aliases:
  - `General_Category`
  - `Script`
  - `Script_Extensions`
  - The binary properties listed in the specification[7] (and no others, to guarantee interoperability). These include, among others: `Alphabetic`, `Uppercase`, `Lowercase`, `White_Space`, `Noncharacter_Code_Point`, `Default_Ignorable_Code_Point`, `Any`, `ASCII`, `Assigned`, `ID_Start`, `ID_Continue`, `Join_Control`, `Emoji_Presentation`, `Emoji_Modifier`, `Emoji_Modifier_Base`.

8.4 Examples

Matching whitespace:

```
> /^\p{White_Space}+$/u.test('\t \n\r')
true
```

Matching letters:

```
> /^\p{Letter}+$/u.test('πüé')
true
```

Matching Greek letters:

```
> /^\p{Script=Greek}+$/u.test('μετά')
true
```

Matching Latin letters:

```
> /^\p{Script=Latin}+$/u.test('Grüße')
true
> /^\p{Script=Latin}+$/u.test('façon')
true
> /^\p{Script=Latin}+$/u.test('mañana')
true
```

Matching lone surrogate characters:

[5] http://unicode.org/Public/UNIDATA/PropertyAliases.txt
[6] http://unicode.org/Public/UNIDATA/PropertyValueAliases.txt
[7] https://tc39.github.io/proposal-regexp-unicode-property-escapes/#sec-static-semantics-unicodematchproperty-p

```
> /^\p{Surrogate}+$/u.test('\u{D83D}')
true
> /^\p{Surrogate}+$/u.test('\u{DE00}')
true
```

Note that Unicode code points in astral planes (such as emojis) are composed of two JavaScript characters (a leading surrogate and a trailing surrogate). Therefore, you'd expect the previous regular expression to match the emoji ☺, which is all surrogates:

```
> '☺'.length
2
> '☺'.charCodeAt(0).toString(16)
'd83d'
> '☺'.charCodeAt(1).toString(16)
'de42'
```

However, with the `/u` flag, property escapes match code points, not JavaScript characters:

```
> /^\p{Surrogate}+$/u.test('☺')
false
```

In other words, ☺ is considered to be a single character:

```
> /^.$/u.test('☺')
true
```

8.5 Trying it out

V8 5.8+ implement this proposal, it is switched on via `--harmony_regexp_property`:

- Node.js: `node --harmony_regexp_property`
  - Check Node's version of V8 via `npm version`
- Chrome:
  - Go to `chrome://version/`[8]
  - Check the version of V8.
  - Find the "Executable Path". For example: `/Applications/Google Chrome.app/Contents/MacOS/Google Chrome`
  - Start Chrome: `'/Applications/Google Chrome.app/Contents/MacOS/Google Chrome' --js-flags="--harmony_regexp_property"`

8.6 Further reading

JavaScript:

- "Unicode and JavaScript[9]" (in "Speaking JavaScript")
- Regular expressions: "New flag `/u` (unicode)[10]" (in "Exploring ES6")

[8] chrome://version/
[9] http://speakingjs.com/es5/ch24.html
[10] http://exploringjs.com/es6/ch_regexp.html#sec_regexp-flag-u

The Unicode standard:

- Unicode Technical Report #23: The Unicode Character Property Model[11] (Editors: Ken Whistler, Asmus Freytag)
- Unicode Standard Annex #44: Unicode Character Database[12] (Editors: Mark Davis, Laurențiu Iancu, Ken Whistler)
- Unicode Character Database: `PropList.txt`[13], `PropertyAliases.txt`[14], `PropertyValueAliases.txt`[15]
- "Unicode character property[16]" (Wikipedia)

[11] http://unicode.org/reports/tr23/
[12] http://www.unicode.org/reports/tr44/
[13] http://unicode.org/Public/UNIDATA/PropList.txt
[14] http://unicode.org/Public/UNIDATA/PropertyAliases.txt
[15] http://unicode.org/Public/UNIDATA/PropertyValueAliases.txt
[16] https://en.wikipedia.org/wiki/Unicode_character_property

Chapter 9

RegExp lookbehind assertions

Contents

This chapter explains the proposal "RegExp Lookbehind Assertions[1]" by Gorkem Yakin, Nozomu Katō, Daniel Ehrenberg.

A *lookaround assertion* is a construct inside a regular expression that specifies what the surroundings of the current location must look like, but has no other effect. It is also called a *zero-width assertion*.

The only lookaround assertion currently supported by JavaScript is the *lookahead assertion*, which matches what follows the current location. This chapter describes a proposal for a *lookbehind assertion*, which matches what precedes the current location.

9.1 Lookahead assertions

A lookahead assertion inside a regular expression means: whatever comes next must match the assertion, but nothing else happens. That is, nothing is captured and the assertion doesn't contribute to the overall matched string.

Take, for example, the following regular expression

```
const RE_AS_BS = /aa(?=bb)/;
```

It matches the string `'aabb'`, but the overall matched string does not include the b's:

```
const match1 = RE_AS_BS.exec('aabb');
console.log(match1[0]); // 'aa'
```

[1]https://github.com/tc39/proposal-regexp-lookbehind

Furthermore, it does not match a string that doesn't have two b's:

```
const match2 = RE_AS_BS.exec('aab');
console.log(match2); // null
```

A negative lookahead assertion means that what comes next must *not* match the assertion. For example:

```
> const RE_AS_NO_BS = /aa(?!bb)/;
> RE_AS_NO_BS.test('aabb')
false
> RE_AS_NO_BS.test('aab')
true
> RE_AS_NO_BS.test('aac')
true
```

9.2 Lookbehind assertions

Lookbehind assertions work like lookahead assertions, but in the opposite direction.

9.2.1 Positive lookbehind assertions

For a positive lookbehind assertion, the text preceding the current location must match the assertion (but nothing else happens).

```
const RE_DOLLAR_PREFIX = /(?<=\$)foo/g;
'$foo %foo foo'.replace(RE_DOLLAR_PREFIX, 'bar');
    // '$bar %foo foo'
```

As you can see, `'foo'` is only replaced if it is preceded by a dollar sign. You can also see that the dollar sign is not part of the total match, because the latter is completely replaced by `'bar'`.

Achieving the same result without a lookbehind assertion is less elegant:

```
const RE_DOLLAR_PREFIX = /(\$)foo/g;
'$foo %foo foo'.replace(RE_DOLLAR_PREFIX, '$1bar');
    // '$bar %foo foo'
```

And this approach doesn't work if the prefix should be part of the previous match:

```
> 'a1ba2ba3b'.match(/(?<=b)a.b/g)
[ 'a2b', 'a3b' ]
```

9.2.2 Negative lookbehind assertions

A negative lookbehind assertion only matches if the current location is *not* preceded by the assertion, but has no other effect. For example:

```
const RE_NO_DOLLAR_PREFIX = /(?<!\$)foo/g;
'$foo %foo foo'.replace(RE_NO_DOLLAR_PREFIX, 'bar');
    // '$foo %bar bar'
```

There is no simple (general) way to achieve the same result without a lookbehind assertion.

9.3 Conclusions

Lookahead assertions make most sense at the end of regular expressions. Lookbehind assertions make most sense at the beginning of regular expressions.

The use cases for lookaround assertions are:

- `replace()`
- `match()` (especially if the regular expression has the flag `/g`)
- `split()` (note the space at the beginning of `' b,c'`):

  ```
  > 'a, b,c'.split(/,(?= )/)
  [ 'a', ' b,c' ]
  ```

Other than those use cases, you can just as well make the assertion a real part of the regular expression.

9.4 Further reading

- V8 JavaScript Engine: RegExp lookbehind assertions[2]
- Section "Manually Implementing Lookbehind[3]" in "Speaking JavaScript"

[2] https://v8project.blogspot.de/2016/02/regexp-lookbehind-assertions.html
[3] http://speakingjs.com/es5/ch19.html#regexp-look-behind

Chapter 10

`s` (`dotAll`) flag for regular expressions

Contents

This chapter explains the proposal "`s` (`dotAll`) flag for regular expressions[1]" by Mathias Bynens.

10.1 Overview

Currently, the dot (`.`) in regular expressions doesn't match line terminator characters:

```
> /^.$/.test('\n')
false
```

The proposal specifies the regular expression flag `/s` that changes that:

```
> /^.$/s.test('\n')
true
```

10.2 Limitations of the dot (`.`) in regular expressions

The dot (`.`) in regular expressions has two limitations.

First, it doesn't match astral (non-BMP) characters such as emoji:

[1] https://github.com/tc39/proposal-regexp-dotall-flag

```
> /^.$/.test('☺')
false
```

This can be fixed via the `/u` (`unicode`) flag:

```
> /^.$/u.test('☺')
true
```

Second, the dot does not match line terminator characters:

```
> /^.$/.test('\n')
false
```

That can currently only be fixed by replacing the dot with work-arounds such as `[^]` ("all characters except no character") or `[\s\S]` ("either whitespace nor not whitespace").

```
> /^[^]$/.test('\n')
true
> /^[\s\S]$/.test('\n')
true
```

10.2.1 Line terminators recognized by ECMAScript

Line termators in ECMAScript affect:

- The dot, in all regular expressions that don't have the flag `/s`.
- The anchors `^` and `$` if the flag `/m` (`multiline`) is used.

The following for characters are considered line terminators by ECMAScript:

- U+000A LINE FEED (LF) (`\n`)
- U+000D CARRIAGE RETURN (CR) (`\r`)
- U+2028 LINE SEPARATOR
- U+2029 PARAGRAPH SEPARATOR

There are additionally some newline-ish characters that are not considered line terminators by ECMAScript:

- U+000B VERTICAL TAB (`\v`)
- U+000C FORM FEED (`\f`)
- U+0085 NEXT LINE

Those three characters *are* matched by the dot without a flag:

```
> /^...$/.test('\v\f\u{0085}')
true
```

10.3 The proposal

The proposal introduces the regular expression flag `/s` (short for "singleline"), which leads to the dot matching line terminators:

```
> /^.$/s.test('\n')
true
```

The long name of `/s` is `dotAll`:

```
> /./s.dotAll
true
> /./s.flags
's'
> new RegExp('.', 's').dotAll
true
> /./.dotAll
false
```

10.3.1 `dotAll` vs. `multiline`

- `dotAll` only affects the dot.
- `multiline` only affects `^` and `$`.

10.4 FAQ

10.4.1 Why is the flag named `/s`?

`dotAll` is a good description of what the flag does, so, arguably, `/a` or `/d` would have been better names. However, `/s` is already an established name (Perl, Python, Java, C#, …).

Chapter 11

Promise.prototype.finally()

Contents

This chapter explains the proposal "`Promise.prototype.finally`[1]" by Jordan Harband.

11.1 How does it work?

`.finally()` works as follows:

```
promise
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
```

`finally`'s callback is always executed. Compare:

- `then`'s callback is only executed if `promise` is fulfilled.
- `catch`'s callback is only executed if `promise` is rejected. Or if `then`'s callback throws an exception or returns a rejected Promise.

In other words: Take the following piece of code.

```
promise
.finally(() => {
    «statements»
});
```

This piece of code is equivalent to:

[1] https://github.com/tc39/proposal-promise-finally

```
promise
.then(
    result => {
        «statements»
        return result;
    },
    error => {
        «statements»
        throw error;
    }
);
```

11.2 Use case

The most common use case is similar to the most common use case of the synchronous `finally` clause: cleaning up after you are done with a resource. That should always happen, regardless of whether everything went smoothly or there was an error.

For example:

```
let connection;
db.open()
.then(conn => {
    connection = conn;
    return connection.select({ name: 'Jane' });
})
.then(result => {
    // Process result
    // Use `connection` to make more queries
})
···
.catch(error => {
    // handle errors
})
.finally(() => {
    connection.close();
});
```

11.3 `.finally()` is similar to `finally {}` in synchronous code

In synchronous code, the `try` statement has three parts: The `try` clause, the `catch` clause and the `finally` clause.

In Promises:

- The `try` clause very loosely corresponds to invoking a Promise-based function or calling `.then()`.
- The `catch` clause corresponds to the `.catch()` method of Promises.

- The `finally` clause corresponds to the new Promise method `.finally()` introduced by the proposal.

However, where `finally {}` can `return` and `throw`, returning has no effect inside the callback `.finally()`, only throwing. That's because the method can't distinguish between the callback returning explicitly and it finishing without doing so.

11.4 Availability

- The npm package `promise.prototype.finally`[2] is a polyfill for `.finally()`.
- V8 5.8+ (e.g. in Node.js 8.1.4+): available behind the flag `--harmony-promise-finally` (details[3]).

11.5 Further reading

- "Promises for asynchronous programming[4]" in "Exploring ES6"

[2] https://github.com/es-shims/Promise.prototype.finally
[3] https://chromium.googlesource.com/v8/v8.git/+/18ad0f13afeaabff4e035fddd9edc3d319152160
[4] http://exploringjs.com/es6/ch_promises.html

Chapter 12

Template Literal Revision

Contents

The ECMAScript proposal "Template Literal Revision[1]" by Tim Disney gives the innards of tagged template literals more syntactic freedom.

12.1 Tag functions and escape sequences

With tagged template literals, you can make a function call by mentioning a function before a template literal:

```
> String.raw`\u{4B}`
'\\u{4B}'
```

`String.raw` is a so-called *tag function*. Tag functions receive two versions of the fixed string pieces (*template strings*) in a template literal:

- Cooked: escape sequences are interpreted. `` `\u{4B}` `` becomes `'K'`.
- Raw: escape sequences are normal text. `` `\u{4B}` `` becomes `'\\u{4B}'`.

The following tag function illustrates how that works:

```
function tagFunc(tmplObj, substs) {
    return {
        Cooked: tmplObj,
        Raw: tmplObj.raw,
    };
}
```

Using the tag function:

[1] https://tc39.github.io/proposal-template-literal-revision/

```
> tagFunc`\u{4B}`;
{ Cooked: [ 'K' ], Raw: [ '\\u{4B}' ] }
```

For more information on tag functions, consult Sect. “Implementing tag functions[2]” in “Exploring ES6”.

12.2 Problem: some text is illegal after backslashes

The problem is that even with the raw version, you don’t have total freedom within template literals in ES2016. After a backslash, some sequences of characters are not legal anymore:

- `\u` starts a Unicode escape, which must look like `\u{1F4A4}` or `\u004B`.
- `\x` starts a hex escape, which must look like `\x4B`.
- `\` plus digit starts an octal escape (such as `\141`). Octal escapes are forbidden in template literals and strict mode string literals.

That prevents tagged template literals such as:

```
latex`\unicode`
windowsPath`C:\uuu\xxx\111`
```

12.3 Solution

The solution is drop all syntactic restrictions related to escape sequences. Then illegal escape sequences simply show up verbatim in the raw representation. But what about the cooked representation? Every template string with an illegal escape sequence is an undefined element in the cooked Array:

```
> tagFunc`\uu ${1} \xx`
{ Cooked: [ undefined, undefined ], Raw: [ '\\uu ', ' \\xx' ] }
```

[2] http://exploringjs.com/es6/ch_template-literals.html#_implementing-tag-functions

Part III

ECMAScript 2019

Chapter 13

`Array.prototype.{flat,flatMap}`

Contents

13.1 Overview

The ES2019 feature `Array.prototype.{flat,flatMap}`[1] (by Michael Ficarra, Brian Terlson, Mathias Bynens) adds two new methods to Arrays (to `Array<T>.prototype`): `.flat()` and `.flatMap()`.

13.1.1 `.flat()`

The type signature of `Array<T>.prototype.flat()` is:

```
.flat(depth = 1): any[]
```

[1] https://github.com/tc39/proposal-flatMap

`.flat()` "flattens" an Array: It creates a copy of the Array where values in nested Arrays all appear at the top level. The parameter `depth` controls how deeply `.flat()` looks for non-Array values. For example:

```
> [ 1,2, [3,4], [[5,6]] ].flat(0) // no change
[ 1, 2, [ 3, 4 ], [ [ 5, 6 ] ] ]

> [ 1,2, [3,4], [[5,6]] ].flat(1)
[ 1, 2, 3, 4, [ 5, 6 ] ]

> [ 1,2, [3,4], [[5,6]] ].flat(2)
[ 1, 2, 3, 4, 5, 6 ]
```

13.1.2 `.flatMap()`

The type signature of `Array<T>.prototype.flatMap()` is:

```
.flatMap<U>(
  callback: (value: T, index: number, array: T[]) => U|Array<U>,
  thisValue?: any
): U[]
```

`.flatMap()` is the same as first calling `.map()` and then flattening the result. That is, the following two expressions are equivalent:

```
arr.flatMap(func)
arr.map(func).flat(1)
```

For example:

```
> ['a', 'b', 'c'].flatMap(x => x)
[ 'a', 'b', 'c' ]
> ['a', 'b', 'c'].flatMap(x => [x])
[ 'a', 'b', 'c' ]
> ['a', 'b', 'c'].flatMap(x => [[x]])
[ [ 'a' ], [ 'b' ], [ 'c' ] ]

> ['a', 'b', 'c'].flatMap((x, i) => new Array(i+1).fill(x))
[ 'a', 'b', 'b', 'c', 'c', 'c' ]
```

13.2 More information on `.flatMap()`

Both `.map()` and `.flatMap()` take a function `f` as a parameter that controls how an input Array is translated to an output Array:

- With `.map()`, each input Array element is translated to exactly one output element. That is, `f` returns a single value.
- With `.flatMap()`, each input Array element is translated to zero or more output elements. That is, `f` returns an Array of values (it can also return non-Array values, but that is less common).

This is an implementation of `.flatMap()` (a simplified version of JavaScript's implementation that does not conform to the specification):

```
function flatMap(arr, mapFunc) {
  const result = [];
  for (const [index, elem] of arr.entries()) {
    const x = mapFunc(elem, index, arr);
    // We allow mapFunc() to return non-Arrays
    if (Array.isArray(x)) {
      result.push(...x);
    } else {
      result.push(x);
    }
  }
  return result;
}
```

`.flatMap()` is simpler if `mapFunc()` is only allowed to return Arrays, but JavaScript doesn't impose this restriction, because non-Array values are occasionally useful (see the section on `.flat()` for an example).

What is `.flatMap()` good for? Let's look at use cases!

13.3 Use case: filtering and mapping at the same time

The result of the Array method `.map()` always has the same length as the Array it is invoked on. That is, its callback can't skip Array elements it isn't interested in.

The ability of `.flatMap()` to do so is useful in the next example: `processArray()` returns an Array where each element is either a wrapped value or a wrapped error.

```
function processArray(arr, process) {
  return arr.map(x => {
    try {
      return { value: process(x) };
    } catch (e) {
      return { error: e };
    }
  });
}
```

The following code shows `processArray()` in action:

```
let err;
function myFunc(value) {
  if (value < 0) {
    throw (err = new Error('Illegal value: '+value));
  }
  return value;
}
const results = processArray([1, -5, 6], myFunc);
assert.deepEqual(results, [
  { value: 1 },
  { error: err },
```

```
  { value: 6 },
]);
```

`.flatMap()` enables us to extract just the values or just the errors from `results`:

```
const values = results.flatMap(
  result => result.value ? [result.value] : []);
assert.deepEqual(values, [1, 6]);

const errors = results.flatMap(
  result => result.error ? [result.error] : []);
assert.deepEqual(errors, [err]);
```

13.4 Use case: mapping to multiple values

The Array method `.map()` maps each input Array element to one output element. But what if we want to map it to multiple output elements?

That becomes necessary in the following example: The React component `TagList` is invoked with two attributes.

```
<TagList tags={['foo', 'bar', 'baz']}
        handleClick={x => console.log(x)} />
```

The attributes are:

- An Array of tags, each tag being a string.
- A callback for handling clicks on tags.

`TagList` is rendered as a series of links separated by commas:

```
class TagList extends React.Component {
  render() {
    const {tags, handleClick} = this.props;
    return tags.flatMap(
      (tag, index) => {
        const link = <a key={index} href=""
           onClick={e => handleClick(tag, e)}>
           {tag}
        </a>;
        if (index === 0) {
          return [link];
        } else {
          return [', ', link];
        }
      }
    );
  }
}
```

Due to `.flatMap()`, `TagList` is rendered as a single flat Array. The first tag contributes one element to this Array (a link); each of the remaining tags contributes two elements (comma and link).

13.5 Other versions of `.flatMap()`

13.5.1 Arbitrary iterables

`.flatMap()` can be generalized to work with arbitrary iterables:

```
function* flatMapIter(iterable, mapFunc) {
  let index = 0;
  for (const x of iterable) {
    yield* mapFunc(x, index);
    index++;
  }
}
```

Due to Arrays being iterables, you can process them via `flatMapIter()`:

```
function fillArray(x) {
  return new Array(x).fill(x);
}
const iterable = flatMapIter([1,2,3], fillArray);
assert.deepEqual(
  [...iterable], // convert to Array, to check contents
  [1, 2, 2, 3, 3, 3]);
```

One benefit of `flatMapIter()` is that it works incrementally: as soon as the first input value is available, output is produced. In contrast, the Array-based `.flatMap()` needs all of its input to produce its output.

That can be demonstrated via the infinite iterable created by the generator function `naturalNumbers()`:

```
function* naturalNumbers() {
  for (let n=0;; n++) {
    yield n;
  }
}
const infiniteInput = naturalNumbers();
const infiniteOutput = flatMapIter(infiniteInput, fillArray);
const [a,b,c,d,e] = infiniteOutput; // (A)
assert.deepEqual([a,b,c,d,e], [1, 2, 2, 3, 3]);
```

In line A, we extract the first 5 values of `infiniteOutput` via destructuring.

13.5.2 Implementing `.flatMap()` via `.reduce()`

We can use the Array method `.reduce()` to implement a simple version of `.flatMap()`:

```
function flatMap(arr, mapFunc) {
  return arr.reduce(
    (prev, x) => prev.concat(mapFunc(x)),
    []
  );
}
```

It depends on your taste, if you prefer the original, more efficient imperative version or this more concise functional version.

13.6 More information on `.flat()`

This is an implementation of `.flat()` (a simplified version of JavaScript's implementation, that does not conform to the ECMAScript specification):

```
function flat(arr, depth) {
  return flatInto(arr, depth, []);
}
function flatInto(value, depth, target) {
  if (!Array.isArray(value)) {
    target.push(value);
  } else {
    for (const x of value) {
      if (depth >= 1) {
        flatInto(x, depth-1, target);
      } else {
        target.push(x);
      }
    }
  }
  return target;
}
```

`.flat()` with a `depth` of 1 can also be implemented as follows:

```
const flat = (arr) => [].concat(...arr)
```

`.flat(1)` is the same as using `.flatMap()` with the *identity function* (`x => x`). That is, the following two expressions are equivalent:

```
arr.flatMap(x => x)
arr.flat(1)
```

The next subsections cover use cases for `.flat()`.

13.6.1 Use case: conditionally inserting values into an Array

The following code only inserts `'a'` if `cond` is `true`:

```
const cond = false;
const arr = [
  (cond ? 'a' : []),
  'b',
].flat();
assert.deepEqual(arr, ['b']);
```

Caveat: If you replace either `'a'` or `'b'` with an Array, then you have to wrap it in another Array.

13.6.2 Use case: filtering out failures

In the following example, `downloadFiles()` only returns the texts that could be downloaded.

```
async function downloadFiles(urls) {
  const downloadAttempts = await Promises.all( // (A)
    urls.map(url => downloadFile(url)));
  return downloadAttempts.flat(); // (B)
}
async function downloadFile(url) {
  try {
    const response = await fetch(url);
    const text = await response.text();
    return [text]; // (C)
  } catch (err) {
    return []; // (D)
  }
}
```

`downloadFiles()` first maps each URL to a Promise resolving to either:

- An Array with the successfully downloaded text (line C)
- An empty Array (line D)

`Promises.all()` (line A) converts the Array of Promises into a Promise that resolves to a nested Array. `await` (line A) unwraps that Promise and `.flat()` un-nests the Array (line B).

Note that we couldn't have used `.flatMap()` here, because of the barrier imposed by the Promises returned by `downloadFile()`: when it returns a value, it doesn't know yet if it will be a text or an empty Array.

13.7 FAQ

13.7.1 Do `.flat()` and `.flatMap()` also flatten iterable elements?

No, only Arrays are flattened:

```
const set = new Set([3,4]);
assert.deepEqual(
  [[1,2], set].flat(),
  [1, 2, set]);
```

13.8 Further reading

- "A collection of Scala 'flatMap' examples[2]" by Alvin Alexander
- Sect. "Transformation Methods[3]" (which include `.map()`) in "Speaking JavaScript"

[2] http://alvinalexander.com/scala/collection-scala-flatmap-examples-map-flatten
[3] http://speakingjs.com/es5/ch18.html#Array.prototype.map

- Conditionally adding entries inside Array and object literals[4] [loosely related to what is covered in this blog post]

[4]http://2ality.com/2017/04/conditional-literal-entries.html

Chapter 14

`Object.fromEntries()`

Contents

This chapter explains the ES2019 feature "`Object.fromEntries()`[1]" (by Darien Maillet Valentine).

14.1 `Object.fromEntries()` vs. `Object.entries()`

Given an iterable over [key,value] pairs, `Object.fromEntries()` creates an object:

```
assert.deepEqual(
  Object.fromEntries([['foo',1], ['bar',2]]),
  {
    foo: 1,
    bar: 2,
  }
);
```

It does the opposite of `Object.entries()`[2]:

```
const obj = {
  foo: 1,
  bar: 2,
```

[1] https://github.com/tc39/proposal-object-from-entries
[2] http://exploringjs.com/es2016-es2017/ch_object-entries-object-values.html

```
};
assert.deepEqual(
  Object.entries(obj),
  [['foo',1], ['bar',2]]
);
```

Combining `Object.entries()` with `Object.fromEntries()` helps with implementing a variety of operations related to objects. Read on for examples.

14.2 Examples

In this section, we'll use `Object.entries()` and `Object.fromEntries()` to implement several tool functions from the library Underscore[3].

14.2.1 `_.pick(object, ...keys)`

`pick()`[4] removes all properties from `object` whose keys are not among `keys`. The removal is *non-destructive*: `pick()` creates a modified copy and does not change the original. For example:

```
const address = {
  street: 'Evergreen Terrace',
  number: '742',
  city: 'Springfield',
  state: 'NT',
  zip: '49007',
};
assert.deepEqual(
  pick(address, 'street', 'number'),
  {
    street: 'Evergreen Terrace',
    number: '742',
  }
);
```

We can implement `pick()` as follows:

```
function pick(object, ...keys) {
  const filteredEntries = Object.entries(object)
    .filter(([key, _value]) => keys.includes(key));
  return Object.fromEntries(filteredEntries);
}
```

14.2.2 `_.invert(object)`

`invert()`[5] non-destructively swaps the keys and the values of an object:

[3]https://underscorejs.org
[4]https://underscorejs.org/#pick
[5]https://underscorejs.org/#invert

```
assert.deepEqual(
  invert({a: 1, b: 2, c: 3}),
  {1: 'a', 2: 'b', 3: 'c'}
);
```

We can implement it like this:

```
function invert(object) {
  const mappedEntries = Object.entries(object)
    .map(([key, value]) => [value, key]);
  return Object.fromEntries(mappedEntries);
}
```

14.2.3 `_.mapObject(object, iteratee, context?)`

`mapObject()`[6] is like the Array method `.map()`, but for objects:

```
assert.deepEqual(
  mapObject({x: 7, y: 4}, value => value * 2),
  {x: 14, y: 8}
);
```

This is an implementation:

```
function mapObject(object, callback, thisValue) {
  const mappedEntries = Object.entries(object)
    .map(([key, value]) => {
      const mappedValue = callback.call(thisValue, value, key, object);
      return [key, mappedValue];
    });
  return Object.fromEntries(mappedEntries);
}
```

14.2.4 `_.findKey(object, predicate, context?)`

`findKey()`[7] returns the key of the first property for which `predicate` returns `true`:

```
const address = {
  street: 'Evergreen Terrace',
  number: '742',
  city: 'Springfield',
  state: 'NT',
  zip: '49007',
};
assert.equal(
  findKey(address, (value, _key) => value === 'NT'),
  'state'
);
```

[6] https://underscorejs.org/#mapObject
[7] https://underscorejs.org/#findKey

We can implement it as follows:

```
function findKey(object, callback, thisValue) {
  for (const [key, value] of Object.entries(object)) {
    if (callback.call(thisValue, value, key, object)) {
      return key;
    }
  }
  return undefined;
}
```

14.3 An implementation

`Object.fromEntries()` could be implemented as follows (I've omitted a few checks):

```
function fromEntries(iterable) {
  const result = {};
  for (const [key, value] of iterable) {
    let coercedKey;
    if (typeof key === 'string' || typeof key === 'symbol') {
      coercedKey = key;
    } else {
      coercedKey = String(key);
    }
    Object.defineProperty(result, coercedKey, {
      value,
      writable: true,
      enumerable: true,
      configurable: true,
    });
  }
  return result;
}
```

The official polyfill is available via the npm package `object.fromentries`[8].

14.4 A few more details about `Object.fromEntries()`

- Duplicate keys: If you mention the same key multiple times, the last mention "wins".

  ```
  > Object.fromEntries([['a', 1], ['a', 2]])
  { a: 2 }
  ```

- Symbols as keys: Even though `Object.entries()` ignores properties whose keys are symbols, `Object.fromEntries()` accepts symbols as keys.
- Coercion of keys: The keys of the [key,value] pairs are coerced to property keys: Values other than strings and symbols are coerced to strings.

[8] https://github.com/es-shims/Object.fromEntries

- Iterables vs. Arrays:
  - `Object.entries()` returns an Array (which is consistent with `Object.keys()` etc.). Its [key,value] pairs are 2-element Arrays.
  - `Object.fromEntries()` is flexible: It accepts iterables (which includes Arrays and is consistent with `new Map()` etc.). Its [key,value] pairs are only required to be objects that have properties with the keys `'0'` and `'1'` (which includes 2-element Arrays).
- Only enumerable data properties are supported: If you want to create non-enumerable properties and/or non-data properties, you need to use `Object.defineProperty()` or `Object.defineProperties()`[9].

[9] http://speakingjs.com/es5/ch17.html#functions_for_property_descriptors

Chapter 15

`String.prototype.{trimStart,trimEnd}`

Contents

This chapter describes the ES2019 feature "`String.prototype.{trimStart,trimEnd}`[1]" (by Sebastian Markbåge).

15.1 The string methods `.trimStart()` and `.trimEnd()`

JavaScript already supports removing all whitespace from both ends of a string:

```
> '  abc  '.trim()
'abc'
```

The feature additionally introduces methods for only trimming the start of a string and for only trimming the end of a string:

```
> '  abc  '.trimStart()
'abc  '
> '  abc  '.trimEnd()
'  abc'
```

15.2 Legacy string methods: `.trimLeft()` and `.trimRight()`

Many web browsers have the string methods `.trimLeft()` and `.trimRight()`. Those were added to Annex B of the ECMAScript specification (as aliases for `.trimStart()` and `.trimEnd()`): features that are required for web browsers and optional elsewhere.

[1]https://github.com/tc39/proposal-string-left-right-trim

For the core standard, this feature chose different names, because "start" and "end" make more sense than "left" and "right" for human languages whose scripts aren't left-to-right. In that regard, they are consistent with the string methods `.padStart()` and `.padEnd()`.

15.3 What characters count as whitespace?

For trimming, whitespace means:

- WhiteSpace code points (spec[2]):
  - `<TAB>` (CHARACTER TABULATION, U+0009)
  - `<VT>` (LINE TABULATION, U+000B)
  - `<FF>` (FORM FEED, U+000C)
  - `<SP>` (SPACE, U+0020)
  - `<NBSP>` (NO-BREAK SPACE, U+00A0)
  - `<ZWNBSP>` (ZERO WIDTH NO-BREAK SPACE, U+FEFF)
  - Any other Unicode character with the property `White_Space` in category `Space_Separator` (`Zs`).
- LineTerminator code points (spec[3]):
  - `<LF>` (LINE FEED, U+000A)
  - `<CR>` (CARRIAGE RETURN, U+000D)
  - `<LS>` (LINE SEPARATOR, U+2028)
  - `<PS>` (PARAGRAPH SEPARATOR, U+2029)

[2] https://tc39.github.io/ecma262/#sec-white-space
[3] https://tc39.github.io/ecma262/#sec-line-terminators

Chapter 16

`Symbol.prototype.description`

This chapter describes the ES2019 feature "`Symbol.prototype.description`[1]" (by Michael Ficarra).

When creating a symbol via the factory function `Symbol()`, you can optionally provide a string as a description, via a parameter:

```
const sym = Symbol('The description');
```

Until recently, the only way to access the description was by converting the symbol to a string:

```
assert.equal(String(sym), 'Symbol(The description)');
```

The feature introduces the getter `Symbol.prototype.description` to access the description directly:

```
assert.equal(sym.description, 'The description');
```

[1] https://github.com/tc39/proposal-Symbol-description

Chapter 17

Optional catch binding

Contents

This chapter explains the ES2019 feature "Optional catch binding[1]" by Michael Ficarra.

17.1 Overview

The proposal allows you to do the following:

```
try {
  // ···
} catch {
  // ···
}
```

That is useful whenever you don't need the binding ("parameter") of the `catch` clause:

```
try {
  // ···
} catch (error) {
  // ···
}
```

[1]https://github.com/tc39/proposal-optional-catch-binding

If you never use the variable `error`, you may as well omit it, but JavaScript doesn't let you do `catch ()`. Furthermore, linters that check for unused variables complain in such cases.

17.2 Use cases

There are two general reasons for omitting the `catch` binding:

- If you want to completely ignore the error.
- You don't care about the error or you already know what it will be, but you do want to react to it.

My recommendation is to avoid doing that:

- Instead of completely ignoring an error, at least log it to the console.
- Instead of assuming you know what the error will be, check for unexpected types of exceptions and re-throw them.

If you can't and don't want to avoid it, I suggest encapsulating your code, e.g. inside a function, and to document it well.

Next, we'll take a look at use cases for omitting `catch` bindings and at risks and alternatives.

17.2.1 Use case: `JSON.parse()`

With `JSON.parse()`, there is one predictable kind of exception – if the input is not legal JSON:

```
> JSON.parse('abc')
SyntaxError: Unexpected token a in JSON at position 0
```

That's why it can make sense to use it like this:

```
let jsonData;
try {
  jsonData = JSON.parse(str); // (A)
} catch {
  jsonData = DEFAULT_DATA;
}
```

There is one problem with this approach: errors in line A that are not related to parsing will be silently ignored. For example, you may make a typo such as `JSON.prase(str)`. Cases like this have bitten me a few times in the past. Therefore, I now prefer to conditionally re-throw the errors I catch:

```
let jsonData;
try {
  jsonData = JSON.parse(str);
} catch (err) {
  if (err instanceof SyntaxError) {
    jsonData = DEFAULT_DATA;
  } else {
    throw err;
  }
}
```

17.2.2 Use case: property chains

When accessing nested properties that may or may not exist, you can avoid checking for their existence if you simply access them and use a default if there is an exception:

```
function logId(person) {
  let id = 'No ID';
  try {
    id = person.data.id;
  } catch {}
  console.log(id);
}
```

I prefer explicit checks. For example:

```
function logId(person) {
  let id = 'No ID';
  if (person && person.data && person.data.id) {
    id = person.data.id;
  }
  console.log(id);
}
```

This code can be shortened if you consider that the && operator returns the first falsy operand or the last operand (if there is no falsy operand):

```
function logId(person) {
  const id = (person && person.data && person.data.id) || 'No ID';
  console.log(id);
}
```

However, this shorter version is also more obscure.

17.2.3 Use case: `assert.throws()`

Node.js has the API function `assert.throws(func)`[2] that checks whether an error is thrown inside `func`. It could be implemented as follows.

```
function throws(func) {
  try {
    func();
  } catch {
    return; // everything OK
  }
  throw new Error('Function didn’t throw an exception!');
}
```

This function is an example of wrapping an documenting code that ignores caught exceptions.

[2] https://nodejs.org/api/assert.html#assert_assert_throws_block_error_message

17.2.4 Use case: feature detection

The following code snippet demonstrates how to detect whether a given feature exists:

```
let supported;
try {
  useTheFeature();
  supported = true;
} catch {
  supported = false;
}
```

17.2.5 Use case: even logging fails

If even logging doesn't work then, as a last resort, you have no choice but to ignore exceptions (because further logging could make things worse).

```
function logError(err) {
  try {
    // Log or otherwise report the error
    console.error(err);
  } catch {} // there is nothing we can do
}
```

Again, we encapsulate and document the slightly unorthodox code.

17.3 Further reading

- Stack Exchange: "Is it ever ok to have an empty catch statement?[3]"
- GitHub issue (repo of proposal): "Why?[4]"

[3] https://softwareengineering.stackexchange.com/questions/16807/is-it-ever-ok-to-have-an-empty-catch-statement/16822#16822

[4] https://github.com/tc39/proposal-optional-catch-binding/issues/2

Chapter 18

Stable `Array.prototype.sort()`

Starting with ECMAScript 2019, the Array method `.sort()` is guaranteed to be stable. What does that mean (as proposed by Mathias Bynens)?

It means that if elements that are considered equal by sorting (not necessarily in any other way!) then sorting does not change the order of those elements. For example:

```
const arr = [
  { key: 'b', value: 1 },
  { key: 'a', value: 2 },
  { key: 'b', value: 3 },
];
arr.sort((x, y) => x.key.localeCompare(y.key, 'en-US'));
assert.deepEqual(arr, [
  { key: 'a', value: 2 },
  { key: 'b', value: 1 },
  { key: 'b', value: 3 },
]);
```

Two objects have the same `.key`, `'b'`. Their order will always be preserved by `.sort()`. One benefit is that a unit test where stability matters, now works the same across engines.

Chapter 19

Well-formed `JSON.stringify`

This chapter covers the ES2019 feature "Well-formed `JSON.stringify`[1]" by Richard Gibson.

According to the RFC for JSON[2], if you exchange JSON "in public", you must encode it as UTF-8. That can be a problem if you use `JSON.stringify()`, because it may return sequences of UTF-16 code units that can't be encoded as UTF-8.

How can that happen? If a JavaScript string contains a *lone surrogate* (a JavaScript character in the range 0xD800–0xDFFF) then `JSON.stringify()` produces a string with a lone surrogate:

```
assert.equal(JSON.stringify('\u{D800}'), '"\u{D800}"');
```

Lone UTF-16 surrogates cannot be encoded as UTF-8, which is why this proposal changes `JSON.stringify()` so that it represents them via code unit escape sequences:

```
assert.equal(JSON.stringify('\u{D800}'), '"\\ud800"');
```

Note: JSON supports code unit escape sequences (e.g. `\uD800`), but not code point escape sequences (e.g. `\u{D800}`).

[1] https://github.com/tc39/proposal-well-formed-stringify
[2] https://tools.ietf.org/html/rfc8259#section-8.1

Chapter 20

JSON superset

This chapter covers the ES2019 feature “JSON superset[1]” by Richard Gibson) is at [stage 4](http://exploringjs.com/es2016-es2017/ch_tc39-process.html.

At the moment, JSON (as standardized via ECMA-404[2]) is not a subset of ECMAScript:

- Until recently, ECMAScript string literals couldn’t contain the characters U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR (you had to use an escape sequence to put them into a string). That is, the following source code produced a syntax error:

```
const sourceCode = '"\u2028"';
eval(sourceCode); // SyntaxError
```

- JSON string literals can contain these two characters:

```
const json = '"\u2028"';
JSON.parse(json); // OK
```

Given that the syntax of JSON is fixed, a decision was made to remove the restriction for ECMAScript string literals. That simplifies the grammar of the specification, because you don’t need separate rules for ECMAScript string literals and JSON string literals.

[1] https://github.com/tc39/proposal-json-superset
[2] http://www.ecma-international.org/publications/standards/Ecma-404.htm

Chapter 21

`Function.prototype.toString` revision

This chapter covers the ES2019 feature "`Function.prototype.toString` revision[1]" by Michael Ficarra. It brings two major improvements compared to ES2018[2]:

- Whenever possible – source code: If a function was created via ECMAScript source code, `toString()` must return that source code. In ES2016, whether to do so is left up to engines.
- Otherwise – standardized placeholder: In ES2016, if `toString()` could not (or would not) create syntactically valid ECMAScript code, it had to return a string for which `eval()` throws a `SyntaxError`. In other words, `eval()` must not be able to parse the string. This requirement was forward-incompatible – whatever string you come up with, you can never be completely sure that a future version of ECMAScript doesn't make it syntactically valid. In contrast, the proposal standardizes a placeholder: a function whose body is `{ [native code] }`. Details are explained in the next section.

21.1 The algorithm

The proposal distinguishes:

- Functions defined via ECMAScript code: `toString()` must return their original source code.

  `toString()` may return code that is only syntactically valid within its syntactic context:

  ```
  > class C { foo() { /*hello*/ } }
  > C.prototype.foo.toString()
  'foo() { /*hello*/ }'
  ```

  The following two kinds of line breaks are converted to Unix-style `'\n'`:

  - Windows: `'\r\n'`
  - Classic macOS: `'\r'`

[1] https://tc39.github.io/Function-prototype-toString-revision/
[2] https://tc39.github.io/ecma262/2018/#sec-function.prototype.tostring

- Built-in function objects, bound function exotic objects and callable objects which were not defined via ECMAScript code: `toString()` must return a so-called *NativeFunction* string, which looks as follows.

  ```
  "function" BindingIdentifier? "(" FormalParameters ")"
  "{ [native code] }"
  ```

  The parameters can be omitted. If the function is a "well-known intrinsic object[3]" (such as `Array`, `Error`, `isNaN`, etc.) then the initial value of its `name` property must appear in the result. Examples:

  ```
  > isNaN.toString()
  'function isNaN() { [native code] }'
  > Math.pow.toString()
  'function pow() { [native code] }'
  > (function foo() {}).bind(null).toString()
  'function () { [native code] }'
  ```

- Functions created dynamically via the constructors `Function` and `GeneratorFunction`: engines must create the appropriate source code and attach it to the functions. This source code is then returned by `toString()`.

- In all other cases (the receiver `this` is not callable): throw a `TypeError`.

[3] https://tc39.github.io/ecma262/#sec-well-known-intrinsic-objects