Exploring ES2016 and ES2017

```
// ES2016

['a', 'b', 'c'].includes('a'); // true
console.log(6 ** 2); // 36

// ES2017

async function asyncFunc() {
    const result = await otherAsyncFunc();
    console.log(result);
}

const sharedBuffer = new SharedArrayBuffer(
    1 * Int32Array.BYTES_PER_ELEMENT);
const sharedArray = new Int32Array(sharedBuffer);

for (let [k,v] of Object.entries(obj)) {
    console.log(k, ':', v);
}
Object.values({ one: 1, two: 2 }); // [ 1, 2 ]

'x'.padStart(5, 'ab'); // 'ababx'
'x'.padEnd(5, 'ab'); // 'xabab'

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

func(
    'abc',
    123, // tailing comma
);
```

Dr. Axel Rauschmayer

Exploring ES2016 and ES2017

Axel Rauschmayer

This book is for sale at http://leanpub.com/exploring-es2016-es2017

This version was published on 2018-02-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 - 2018 Axel Rauschmayer

Contents

What you need to know about this book

This book is about two versions of JavaScript:

- ECMAScript 2016 and
- ECMAScript 2017

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].

Support

- **Forum**: The "Exploring ES2016 and ES2017" homepage links to a forum[2] where you can discuss questions and ideas related to this book.
- **Errata (typos, errors, etc.)**: On the "Exploring ES6" homepage[3], there are links to a form for submitting errata and to a list with submitted errata.

[1]http://exploringjs.com/
[2]http://exploringjs.com/es2016-es2017.html#forum
[3]http://exploringjs.com/es2016-es2017.html#errata

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.

I Background

1. 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).

1.1 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.).

1.2 How is ECMAScript designed?

1.2.1 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:

- 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.

[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

1.2.2 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.

1.2.2.1 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].

1.2.2.2 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.

1.2.2.3 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.

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

1.2.2.4 Stage 3: candidate

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

[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/ecma262/blob/master/stage0.md
[7]https://github.com/tc39/ecma262/blob/master/FAQ.md

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.

1.2.2.5 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.

1.3 Don't call them ECMAScript 20xx features

As you can see, you can only be sure that a feature will be included in the standard once its proposal has reached stage 4. Then its inclusion in the next ECMAScript release is probable, but not 100% sure, either (it may take longer). Therefore, you can't call proposals (e.g.) "ES7 features" or "ES2016 features", anymore. My two favorite ways of writing headings for articles and blog posts are therefore:

- "ECMAScript proposal: the foo feature". The stage of the proposal is mentioned at the beginning of the article.
- "ES.stage2: the foo feature"

If a proposal is at stage 4, I'd be OK with calling it an ES20xx feature, but it's safest to wait until the spec editor confirms what release it will be included in. `Object.observe` is an example of an ECMAScript proposal that had progressed until stage 2, but was ultimately withdrawn.

[8]https://github.com/tc39/test262

1.4 Further reading

The following were important sources of this chapter:

- The ecma262 (ECMA-262 is the ID of the ECMAScript standard) GitHub repository[9], which contains:
  - A readme file with all proposals at stage 1 or higher[10]
  - A list of stage 0 proposals[11]
  - ECMA-262 frequently asked questions[12]
- The TC39 process document[13]

Other things to read:

- Kangax' ES7 compatibility table[14] shows what proposals are supported where and groups proposals by stage.
- More information on the ES6 design process: section "How ECMAScript 6 was designed[15]" in "Exploring ES6"

[9]https://github.com/tc39/ecma262
[10]https://github.com/tc39/ecma262/blob/master/README.md
[11]https://github.com/tc39/ecma262/blob/master/stage0.md
[12]https://github.com/tc39/ecma262/blob/master/FAQ.md
[13]https://tc39.github.io/process-document/
[14]https://kangax.github.io/compat-table/es7/
[15]http://exploringjs.com/es6/ch_about-es6.html#_how-ecmascript-6-was-designed

2. FAQ: ES2016 and ES2017

2.1 Isn't ECMAScript 2016 too small?

ES2016 being so small demonstrates that the new release process (as described in the previous chapter) works:

- New features are only included after they are completely ready and after there are at least two implementations that were sufficiently field-tested.
- Releases happen much more frequently (once a year) and can be more incremental.

ES2016 will give everyone (TC39, engine implementors, JS developers) time to catch their breath and is a welcome break after the enormous ES6 release.

II ECMAScript 2016

ECMAScript 2016 has just two new features:

- `Array.prototype.includes`
- Exponentiation operator (`**`)

3. `Array.prototype.includes`

This chapter describes the ECMAScript 2016 feature "`Array.prototype.includes`" by Domenic Denicola and Rick Waldron.

3.1 Overview

```
> ['a', 'b', 'c'].includes('a')
true
> ['a', 'b', 'c'].includes('d')
false
```

3.2 The Array method `includes`

The Array method `includes` has the following signature:

```
Array.prototype.includes(value : any) : boolean
```

It returns `true` if `value` is an element of its receiver (`this`) and `false`, otherwise:

```
> ['a', 'b', 'c'].includes('a')
true
> ['a', 'b', 'c'].includes('d')
false
```

`includes` is similar to `indexOf` – the following two expressions are mostly equivalent:

```
arr.includes(x)
arr.indexOf(x) >= 0
```

The main difference is that `includes()` finds `NaN`, whereas `indexOf()` doesn't:

```
> [NaN].includes(NaN)
true
> [NaN].indexOf(NaN)
-1
```

`includes` does not distinguish between `+0` and `-0` (which is how almost all of JavaScript works[1]):

[1] http://speakingjs.com/es5/ch11.html#two_zeros

```
> [-0].includes(+0)
true
```

Typed Arrays will also have a method `includes()`:

```
let tarr = Uint8Array.of(12, 5, 3);
console.log(tarr.includes(5)); // true
```

3.3 Frequently asked questions

- **Why is the method called `includes` and not `contains`?**
  The latter was the initial choice, but that broke code on the web (MooTools adds this method to `Array.prototype`[2]).
- **Why is the method called `includes` and not `has`?**
  `has` is used for keys (`Map.prototype.has`), `includes` is used for elements (`String.prototype.includes`). The elements of a Set can be viewed as being both keys and values, which is why there is a `Set.prototype.has` (and no `includes`).
- **The ES6 method `String.prototype.includes`[3] works with strings, not characters. Isn't that inconsistent w.r.t. `Array.prototype.includes`?**
  If Array `includes` worked exactly like string `includes`, it would accept arrays, not single elements. But the two `includes` follow the example of `indexOf`; characters are seen as a special case and strings with arbitrary lengths as the general case.

3.4 Further reading

- `Array.prototype.includes`[4] (Domenic Denicola, Rick Waldron)

[2]https://esdiscuss.org/topic/having-a-non-enumerable-array-prototype-contains-may-not-be-web-compatible
[3]http://exploringjs.com/es6/ch_strings.html#_checking-for-containment-and-repeating-strings
[4]https://github.com/tc39/Array.prototype.includes/

4. Exponentiation operator (**)

The exponentiation operator (**) is an ECMAScript 2016 feature by Rick Waldron.

4.1 Overview

```
> 6 ** 2
36
```

4.2 An infix operator for exponentiation

** is an infix operator for exponentiation:

```
x ** y
```

produces the same result as

```
Math.pow(x, y)
```

4.3 Examples

Normal use:

```
const squared = 3 ** 2; // 9
```

Exponentiation assignment operator:

```
let num = 3;
num **= 2;
console.log(num); // 9
```

Using exponentiation in a function (Pythagorean theorem):

```
function dist(x, y) {
  return Math.sqrt(x**2 + y**2);
}
```

4.4 Precedence

The exponentiation operator binds very strongly, more strongly than * (which, in turn, binds more strongly than +):

```
> 2**2 * 2
8
> 2 ** (2*2)
16
```

4.5 Further reading

- Exponentiation Operator[1] (Rick Waldron)

[1]https://github.com/rwaldron/exponentiation-operator

III ECMAScript 2017

Major new features:

- Async functions
- Shared memory and atomics

Minor new features:

- `Object.entries()` and `Object.values()`
- New string methods: `padStart` and `padEnd`
- `Object.getOwnPropertyDescriptors()`
- Trailing commas in function parameter lists and calls

5. Async functions

The ECMAScript 2017 feature "Async Functions[1]" was proposed by Brian Terlson.

5.1 Overview

5.1.1 Variants

The following variants of async functions exist. Note the keyword `async` everywhere.

- Async function declarations: `async function foo() {}`
- Async function expressions: `const foo = async function () {};`
- Async method definitions: `let obj = { async foo() {} }`
- Async arrow functions: `const foo = async () => {};`

5.1.2 Async functions always return Promises

Fulfilling the Promise of an async function:

```
async function asyncFunc() {
    return 123;
}

asyncFunc()
.then(x => console.log(x));
    // 123
```

Rejecting the Promise of an async function:

```
async function asyncFunc() {
    throw new Error('Problem!');
}

asyncFunc()
.catch(err => console.log(err));
    // Error: Problem!
```

[1] https://github.com/tc39/ecmascript-asyncawait

5.1.3 Handling results and errors of asynchronous computations via `await`

The operator `await` (which is only allowed inside async functions) waits for its operand, a Promise, to be settled:

- If the Promise is fulfilled, the result of `await` is the fulfillment value.
- If the Promise is rejected, `await` throws the rejection value.

Handling a single asynchronous result:

```
async function asyncFunc() {
    const result = await otherAsyncFunc();
    console.log(result);
}

// Equivalent to:
function asyncFunc() {
    return otherAsyncFunc()
    .then(result => {
        console.log(result);
    });
}
```

Handling multiple asynchronous results sequentially:

```
async function asyncFunc() {
    const result1 = await otherAsyncFunc1();
    console.log(result1);
    const result2 = await otherAsyncFunc2();
    console.log(result2);
}

// Equivalent to:
function asyncFunc() {
    return otherAsyncFunc1()
    .then(result1 => {
        console.log(result1);
        return otherAsyncFunc2();
    })
    .then(result2 => {
        console.log(result2);
    });
}
```

Handling multiple asynchronous results in parallel:

```
async function asyncFunc() {
    const [result1, result2] = await Promise.all([
        otherAsyncFunc1(),
        otherAsyncFunc2(),
    ]);
    console.log(result1, result2);
}

// Equivalent to:
function asyncFunc() {
    return Promise.all([
        otherAsyncFunc1(),
        otherAsyncFunc2(),
    ])
    .then([result1, result2] => {
        console.log(result1, result2);
    });
}
```

Handling errors:

```
async function asyncFunc() {
    try {
        await otherAsyncFunc();
    } catch (err) {
        console.error(err);
    }
}

// Equivalent to:
function asyncFunc() {
    return otherAsyncFunc()
    .catch(err => {
        console.error(err);
    });
}
```

5.2 Understanding async functions

Before I can explain async functions, I need to explain how Promises and generators can be combined to perform asynchronous operations via synchronous-looking code.

For functions that compute their one-off results asynchronously, Promises, which are part of ES6, have become popular. One example is the client-side `fetch` API[2], which is an alternative to XMLHttpRequest for retrieving files. Using it looks as follows:

[2] https://fetch.spec.whatwg.org/#concept-request

```
function fetchJson(url) {
    return fetch(url)
    .then(request => request.text())
    .then(text => {
        return JSON.parse(text);
    })
    .catch(error => {
        console.log(`ERROR: ${error.stack}`);
    });
}
fetchJson('http://example.com/some_file.json')
.then(obj => console.log(obj));
```

5.2.1 Writing asynchronous code via generators

co is a library that uses Promises and generators to enable a coding style that looks more synchronous, but works the same as the style used in the previous example:

```
const fetchJson = co.wrap(function* (url) {
    try {
        let request = yield fetch(url);
        let text = yield request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
});
```

Every time the callback (a generator function!) yields a Promise to co, the callback gets suspended. Once the Promise is settled, co resumes the callback: if the Promise was fulfilled, `yield` returns the fulfillment value, if it was rejected, `yield` throws the rejection error. Additionally, co promisifies the result returned by the callback (similarly to how `then()` does it).

5.2.2 Writing asynchronous code via async functions

Async functions are basically dedicated syntax for what co does:

```
async function fetchJson(url) {
    try {
        let request = await fetch(url);
        let text = await request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
}
```

Internally, async functions work much like generators.

5.2.3 Async functions are started synchronously, settled asynchronously

This is how async functions are executed:

1. The result of an async function is always a Promise `p`. That Promise is created when starting the execution of the async function.
2. The body is executed. Execution may finish permanently via `return` or `throw`. Or it may finish temporarily via `await`; in which case execution will usually continue later on.
3. The Promise `p` is returned.

While executing the body of the async function, `return x` resolves the Promise `p` with `x`, while `throw err` rejects `p` with `err`. The notification of a settlement happens asynchronously. In other words: the callbacks of `then()` and `catch()` are always executed after the current code is finished.

The following code demonstrates how that works:

```
async function asyncFunc() {
    console.log('asyncFunc()'); // (A)
    return 'abc';
}
asyncFunc().
then(x => console.log(`Resolved: ${x}`)); // (B)
console.log('main'); // (C)

// Output:
// asyncFunc()
// main
// Resolved: abc
```

You can rely on the following order:

1. Line (A): the async function is started synchronously. The async function's Promise is resolved via `return`.
2. Line (C): execution continues.
3. Line (B): Notification of Promise resolution happens asynchronously.

5.2.4 Returned Promises are not wrapped

Resolving a Promise is a standard operation. `return` uses it to resolve the Promise `p` of an async function. That means:

1. Returning a non-Promise value fulfills `p` with that value.
2. Returning a Promise means that `p` now mirrors the state of that Promise.

Therefore, you can return a Promise and that Promise won't be wrapped in a Promise:

```
async function asyncFunc() {
    return Promise.resolve(123);
}
asyncFunc()
.then(x => console.log(x)) // 123
```

Intriguingly, returning a rejected Promise leads to the result of the async function being rejected (normally, you'd use `throw` for that):

```
async function asyncFunc() {
    return Promise.reject(new Error('Problem!'));
}
asyncFunc()
.catch(err => console.error(err)); // Error: Problem!
```

That is in line with how Promise resolution works. It enables you to forward both fulfillments and rejections of another asynchronous computation, without an `await`:

```
async function asyncFunc() {
    return anotherAsyncFunc();
}
```

The previous code is roughly similar to – but more efficient than – the following code (which unwraps the Promise of `anotherAsyncFunc()` only to wrap it again):

```
async function asyncFunc() {
    return await anotherAsyncFunc();
}
```

5.3 Tips for using `await`

5.3.1 Don't forget `await`

One easy mistake to make in async functions is to forget `await` when making an asynchronous function call:

```
async function asyncFunc() {
    const value = otherAsyncFunc(); // missing `await`!
    ...
}
```

In this example, `value` is set to a Promise, which is usually not what you want in async functions.

`await` can even make sense if an async function doesn't return anything. Then its Promise is simply used as a signal for telling the caller that it is finished. For example:

```
async function foo() {
    await step1(); // (A)
    ...
}
```

The `await` in line (A) guarantees that `step1()` is completely finished before the remainder of `foo()` is executed.

5.3.2 You don't need `await` if you "fire and forget"

Sometimes, you only want to trigger an asynchronous computation and are not interested in when it is finished. The following code is an example:

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello'); // don't wait
    writer.write('world'); // don't wait
    await writer.close(); // wait for file to close
}
```

Here, we don't care when individual writes are finished, only that they are executed in the right order (which the API would have to guarantee, but that is encouraged by the execution model of async functions – as we have seen).

The `await` in the last line of `asyncFunc()` ensures that the function is only fulfilled after the file was successfully closed.

Given that returned Promises are not wrapped, you can also `return` instead of `await writer.close()`:

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello');
    writer.write('world');
    return writer.close();
}
```

Both versions have pros and cons, the `await` version is probably slightly easier to understand.

5.3.3 `await` is sequential, `Promise.all()` is parallel

The following code make two asynchronous function calls, `asyncFunc1()` and `asyncFunc2()`.

```
async function foo() {
    const result1 = await asyncFunc1();
    const result2 = await asyncFunc2();
}
```

However, these two function calls are executed sequentially. Executing them in parallel tends to speed things up. You can use `Promise.all()` to do so:

```
async function foo() {
    const [result1, result2] = await Promise.all([
        asyncFunc1(),
        asyncFunc2(),
    ]);
}
```

Instead of awaiting two Promises, we are now awaiting a Promise for an Array with two elements.

5.4 Async functions and callbacks

One limitation of async functions is that `await` only affects the directly surrounding async function. Therefore, an async function can't `await` in a callback (however, callbacks can be async functions themselves, as we'll see later on). That makes callback-based utility functions and methods tricky to use. Examples include the Array methods `map()` and `forEach()`.

5.4.1 `Array.prototype.map()`

Let's start with the Array method `map()`. In the following code, we want to download the files pointed to by an Array of URLs and return them in an Array.

```
async function downloadContent(urls) {
    return urls.map(url => {
        // Wrong syntax!
        const content = await httpGet(url);
        return content;
    });
}
```

This does not work, because `await` is syntactically illegal inside normal arrow functions. How about using an async arrow function, then?

```
async function downloadContent(urls) {
    return urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
    });
}
```

There are two issues with this code:

- The result is now an Array of Promises, not an Array of strings.
- The work performed by the callbacks isn't finished once `map()` is finished, because `await` only pauses the surrounding arrow function and `httpGet()` is resolved asynchronously. That means you can't use `await` to wait until `downloadContent()` is finished.

We can fix both issues via `Promise.all()`, which converts an Array of Promises to a Promise for an Array (with the values fulfilled by the Promises):

```
async function downloadContent(urls) {
    const promiseArray = urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
    });
    return await Promise.all(promiseArray);
}
```

The callback for `map()` doesn't do much with the result of `httpGet()`, it only forwards it. Therefore, we don't need an async arrow function here, a normal arrow function will do:

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return await Promise.all(promiseArray);
}
```

There is one small improvement that we still can make: This async function is slightly inefficient – it first unwraps the result of `Promise.all()` via `await`, before wrapping it again via `return`. Given that `return` doesn't wrap Promises, we can return the result of `Promise.all()` directly:

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return Promise.all(promiseArray);
}
```

5.4.2 `Array.prototype.forEach()`

Let's use the Array method `forEach()` to log the contents of several files pointed to via URLs:

```
async function logContent(urls) {
    urls.forEach(url => {
        // Wrong syntax
        const content = await httpGet(url);
        console.log(content);
    });
}
```

Again, this code will produce a syntax error, because you can't use `await` inside normal arrow functions.

Let's use an async arrow function:

```
async function logContent(urls) {
    urls.forEach(async url => {
        const content = await httpGet(url);
        console.log(content);
    });
    // Not finished here
}
```

This does work, but there is one caveat: the Promise returned by `httpGet()` is resolved asynchronously, which means that the callbacks are not finished when `forEach()` returns. As a consequence, you can't await the end of `logContent()`.

If that's not what you want, you can convert `forEach()` into a `for-of` loop:

```
async function logContent(urls) {
    for (const url of urls) {
        const content = await httpGet(url);
        console.log(content);
    }
}
```

Now everything is finished after the `for-of` loop. However, the processing steps happen sequentially: `httpGet()` is only called a second time *after* the first call is finished. If you want the processing steps to happen in parallel, you must use `Promise.all()`:

```
async function logContent(urls) {
    await Promise.all(urls.map(
        async url => {
            const content = await httpGet(url);
            console.log(content);
        }));
}
```

`map()` is used to create an Array of Promises. We are not interested in the results they fulfill, we only `await` until all of them are fulfilled. That means that we are completely done at the end of this async function. We could just as well return `Promise.all()`, but then the result of the function would be an Array whose elements are all `undefined`.

5.5 Tips for using async functions

5.5.1 Know your Promises

The foundation of async functions is Promises[3]. That's why understanding the latter is crucial for understanding the former. Especially when connecting old code that isn't based on Promises with async functions, you often have no choice but to use Promises directly.

For example, this is a "promisified" version of `XMLHttpRequest`:

```
function httpGet(url, responseType="") {
    return new Promise(
        function (resolve, reject) {
            const request = new XMLHttpRequest();
            request.onload = function () {
                if (this.status === 200) {
                    // Success
                    resolve(this.response);
                } else {
```

[3]http://exploringjs.com/es6/ch_promises.html

```
                // Something went wrong (404 etc.)
                reject(new Error(this.statusText));
            }
        };
        request.onerror = function () {
            reject(new Error(
                'XMLHttpRequest Error: '+this.statusText));
        };
        request.open('GET', url);
        xhr.responseType = responseType;
        request.send();
    });
}
```

The API of `XMLHttpRequest` is based on callbacks. Promisifying it via an async function would mean that you'd have to fulfill or reject the Promise returned by the function from within callbacks. That's impossible, because you can only do so via `return` and `throw`. And you can't `return` the result of a function from within a callback. `throw` has similar constraints.

Therefore, the common coding style for async functions will be:

- Use Promises directly to build asynchronous primitives.
- Use those primitives via async functions.

Further reading: chapter "Promises for asynchronous programming[4]" in "Exploring ES6".

5.5.2 Immediately Invoked Async Function Expressions

Sometimes, it'd be nice if you could use `await` at the top level of a module or script. Alas, it's only available inside async functions. You therefore have several options. You can either create an async function `main()` and call it immediately afterwards:

```
async function main() {
    console.log(await asyncFunction());
}
main();
```

Or you can use an Immediately Invoked Async Function Expression:

```
(async function () {
    console.log(await asyncFunction());
})();
```

Another option is an Immediately Invoked Async Arrow Function:

[4]http://exploringjs.com/es6/ch_promises.html

```
(async () => {
    console.log(await asyncFunction());
})();
```

5.5.3 Unit testing with async functions

The following code uses the test-framework mocha[5] to unit-test the asynchronous functions `asyncFunc1()` and `asyncFunc2()`:

```
import assert from 'assert';

// Bug: the following test always succeeds
test('Testing async code', function () {
    asyncFunc1() // (A)
    .then(result1 => {
        assert.strictEqual(result1, 'a'); // (B)
        return asyncFunc2();
    })
    .then(result2 => {
        assert.strictEqual(result2, 'b'); // (C)
    });
});
```

However, this test always succeeds, because mocha doesn't wait until the assertions in line (B) and line (C) are executed.

You can fix this by returning the result of the Promise chain, because mocha recognizes if a test returns a Promise and then waits until that Promise is settled (unless there is a timeout).

```
return asyncFunc1() // (A)
```

Conveniently, async functions always return Promises, which makes them perfect for this kind of unit test:

```
import assert from 'assert';
test('Testing async code', async function () {
    const result1 = await asyncFunc1();
    assert.strictEqual(result1, 'a');
    const result2 = await asyncFunc2();
    assert.strictEqual(result2, 'b');
});
```

There are thus two advantages to using async functions for asynchronous unit tests in mocha: the code is more concise and returning Promises is taken care of, too.

[5] https://mochajs.org/

5.5.4 Don’t worry about unhandled rejections

JavaScript engines are becoming increasingly good at warning about rejections that are not handled. For example, the following code would often fail silently in the past, but most modern JavaScript engines now report an unhandled rejection:

```
async function foo() {
    throw new Error('Problem!');
}
foo();
```

5.6 Further reading

- Async Functions[6] (proposal by Brian Terlson)
- Simplifying asynchronous computations via generators[7] (section in “Exploring ES6”)

[6] https://github.com/tc39/ecmascript-asyncawait
[7] http://exploringjs.com/es6/ch_generators.html#sec_co-library

6. Shared memory and atomics

The ECMAScript 2017 feature "Shared memory and atomics[1]" was designed by Lars T. Hansen. It introduces a new constructor `SharedArrayBuffer` and a namespace object `Atomics` with helper functions. This chapter explains the details.

6.1 Parallelism vs. concurrency

Before we begin, let's clarify two terms that are similar, yet distinct: "parallelism" and "concurrency". Many definitions for them exist; I'm using them as follows:

- Parallelism (parallel vs. serial): execute multiple tasks simultaneously
- Concurrency (concurrent vs. sequential): execute several tasks during overlapping periods of time (and not one after another).

Both are closely related, but not the same:

- Parallelism without concurrency: single instruction, multiple data (SIMD). Multiple computations happen in parallel, but only a single task (instruction) is executed at any given moment.
- Concurrency without parallelism: multitasking via time-sharing on a single-core CPU.

However, it is difficult to use these terms precisely, which is why interchanging them is usually not a problem.

6.1.1 Models of parallelism

Two models of parallelism are:

- Data parallelism: The same piece of code is executed several times in parallel. The instances operate on different elements of the same dataset. For example: MapReduce is a data-parallel programming model.
- Task parallelism: Different pieces of code are executed in parallel. Examples: web workers and the Unix model of spawning processes.

[1] https://github.com/tc39/ecmascript_sharedmem

6.2 A history of JS parallelism

- JavaScript started as being executed in a single thread. Some tasks could be performed asynchronously: browsers usually ran those tasks in separate threads and later fed their results back into the single thread, via callbacks.
- Web workers brought task parallelism to JavaScript: They are relatively heavweight processes. Each worker has its own global environment. By default, nothing is shared. Communication between workers (or between workers and the main thread) evolved:
  - At first, you could only send and receive strings.
  - Then, structured cloning was introduced: copies of data could be sent and received. Structured cloning works for most data[2] (JSON data, Typed Arrays, regular expressions, `Blob` objects, `ImageData` objects, etc.). It can even handle cyclic references between objects correctly. However, error objects, function objects and DOM nodes cannot be cloned.
  - Transferables move data between workers: the sending party loses access as the receiving party gains access to data.
- Computing on GPUs (which tend to do data parallelism well) via WebGL[3]: It's a bit of a hack and works as follows.[4]
  - Input: your data, converted into an image (pixel by pixel).
  - Processing: OpenGL pixel shaders can perform arbitrary computations on GPUs. Your pixel shader transforms the input image.
  - Output: again an image that you can convert back to your kind of data.
- SIMD (low-level data parallelism): is supported via the ECMAScript proposal SIMD.js[5]. It allows you to perform operations (such as addition and square root) on several integers or floats at the same time.
- PJS (codenamed River Trail): the plan of this ultimately abandoned project was to bring high-level data parallelism (think map-reduce via pure functions) to JavaScript. However, there was not enough interest from developers and engine implementers. Without implementations, one could not experiment with this API, because it can't be polyfilled. On 2015-01-05, Lars T. Hansen announced[6] that an experimental implementation was going to be removed from Firefox.

6.2.1 The next step: `SharedArrayBuffer`

What's next? For low-level parallelism, the direction is quite clear: support SIMD and GPUs as well as possible. However, for high-level parallelism, things are much less clear, especially after the failure of PJS.

What is needed is a way to try out many approaches, to find out how to best bring high-level parallelism to JavaScript. Following the principles of the extensible web manifesto, the proposal "shared memory and atomics" (a.k.a. "Shared Array Buffers") does so by providing low-level primitives that can be used to implement higher-level constructs.

[2]https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm
[3]https://www.khronos.org/registry/webgl/specs/latest/2.0/
[4]http://blog.stevensanderson.com/2014/06/11/write-massively-parallel-gpu-code-for-the-browser-with-webgl-ndc-2014/
[5]http://2ality.com/2013/12/simd-js.html
[6]https://bugzilla.mozilla.org/show_bug.cgi?id=1117724

6.3 Shared Array Buffers

Shared Array Buffers are a primitive building block for higher-level concurrency abstractions. They allow you to share the bytes of a `SharedArrayBuffer` object between multiple workers and the main thread (the buffer is shared, to access the bytes, wrap it in a Typed Array). This kind of sharing has two benefits:

- You can share data between workers more quickly.
- Coordination between workers becomes simpler and faster (compared to `postMessage()`).

6.3.1 Creating and sending a Shared Array Buffer

```
// main.js

const worker = new Worker('worker.js');

// To be shared
const sharedBuffer = new SharedArrayBuffer( // (A)
    10 * Int32Array.BYTES_PER_ELEMENT); // 10 elements

// Share sharedBuffer with the worker
worker.postMessage({sharedBuffer}); // clone

// Local only
const sharedArray = new Int32Array(sharedBuffer); // (B)
```

You create a Shared Array Buffer the same way you create a normal Array Buffer: by invoking the constructor and specifying the size of the buffer in bytes (line A). What you share with workers is the buffer. For your own, local, use, you normally wrap Shared Array Buffers in Typed Arrays (line B).

Warning: Cloning a Shared Array Buffer is the correct way of sharing it, but some engines still implement an older version of the API and require you to transfer it:

```
worker.postMessage({sharedBuffer}, [sharedBuffer]); // transfer (deprecated)
```

In the final version of the API, transferring a Shared Array Buffer means that you lose access to it.

6.3.2 Receiving a Shared Array Buffer

The implementation of the worker looks as follows.

```
// worker.js

self.addEventListener('message', function (event) {
    const {sharedBuffer} = event.data;
    const sharedArray = new Int32Array(sharedBuffer); // (A)

    // ···
});
```

We first extract the Shared Array Buffer that was sent to us and then wrap it in a Typed Array (line A), so that we can use it locally.

6.4 Atomics: safely accessing shared data

6.4.1 Problem: Optimizations make code unpredictable across workers

In single threads, compilers can make optimizations that break multi-threaded code.

Take, for example the following code:

```
while (sharedArray[0] === 123) ;
```

In a single thread, the value of `sharedArray[0]` never changes while the loop runs (if `sharedArray` is an Array or Typed Array that wasn't patched in some manner). Therefore, the code can be optimized as follows:

```
const tmp = sharedArray[0];
while (tmp === 123) ;
```

However, in a multi-threaded setting, this optimization prevents us from using this pattern to wait for changes made in another thread.

Another example is the following code:

```
// main.js
sharedArray[1] = 11;
sharedArray[2] = 22;
```

In a single thread, you can rearrange these write operations, because nothing is read in-between. For multiple threads, you get into trouble whenever you expect the writes to be done in a specific order:

```
// worker.js
while (sharedArray[2] !== 22) ;
console.log(sharedArray[1]); // 0 or 11
```

These kinds of optimizations make it virtually impossible to synchronize the activity of multiple workers operating on the same Shared Array Buffer.

6.4.2 Solution: atomics

The proposal provides the global variable `Atomics` whose methods have three main use cases.

6.4.2.1 Use case: synchronization

`Atomics` methods can be used to synchronize with other workers. For example, the following two operations let you read and write data and are never rearranged by compilers:

- `Atomics.load(ta : TypedArray<T>, index) : T`
- `Atomics.store(ta : TypedArray<T>, index, value : T) : T`

The idea is to use normal operations to read and write most data, while `Atomics` operations (`load`, `store` and others) ensure that the reading and writing is done safely. Often, you'll use custom synchronization mechanisms, such as locks, whose implementations are based on `Atomics`.

This is a very simple example that always works, thanks to `Atomics` (I've omitted setting up `sharedArray`):

```
// main.js
console.log('notifying...');
Atomics.store(sharedArray, 0, 123);

// worker.js
while (Atomics.load(sharedArray, 0) !== 123) ;
console.log('notified');
```

6.4.2.2 Use case: waiting to be notified

Using a `while` loop to wait for a notification is not very efficient, which is why `Atomics` has operations that help:

- `Atomics.wait(ta: Int32Array, index, value, timeout)`
  waits for a notification at `ta[index]`, but only if `ta[index]` is `value`.
- `Atomics.wake(ta : Int32Array, index, count)`
  wakes up `count` workers that are waiting at `ta[index]`.

6.4.2.3 Use case: atomic operations

Several `Atomics` operations perform arithmetic and can’t be interrupted while doing so, which helps with synchronization. For example:

- `Atomics.add(ta : TypedArray<T>, index, value) : T`

Roughly, this operation performs:

```
ta[index] += value;
```

6.4.3 Problem: torn values

Another problematic effect with shared memory is *torn values* (garbage): when reading, you may see an intermediate value – neither the value before a new value was written to memory nor the new value.

Sect “Tear-Free Reads” in the spec states that there is no tear if and only if:

- Both reading and writing happens via Typed Arrays (not DataViews).
- Both Typed Arrays are *aligned* with their Shared Array Buffers:

  ```
  sharedArray.byteOffset % sharedArray.BYTES_PER_ELEMENT === 0
  ```

- Both Typed Arrays have the same number of bytes per element.

In other words, torn values are an issue whenever the same Shared Array Buffer is accessed via:

- One or more DateViews
- One or more unaligned Typed Arrays
- Typed Arrays with different element sizes

To avoid torn values in these cases, use `Atomics` or synchronize.

6.5 Shared Array Buffers in use

6.5.1 Shared Array Buffers and the run-to-completion semantics of JavaScript

JavaScript has so-called *run-to-completion semantics*: every function can rely on not being interrupted by another thread until it is finished. Functions become transactions and can perform complete algorithms without anyone seeing the data they operate on in an intermediate state.

Shared Array Buffers break run to completion (RTC): data a function is working on can be changed by another thread during the runtime of the function. However, code has complete control over whether or not this violation of RTC happens: if it doesn’t use Shared Array Buffers, it is safe.

This is loosely similar to how async functions violate RTC. There, you opt into a blocking operation via the keyword `await`.

6.5.2 Shared Array Buffers and asm.js and WebAssembly

Shared Array Buffers enable emscripten to compile pthreads to asm.js. Quoting an emscripten documentation page[7]:

> [Shared Array Buffers allow] Emscripten applications to share the main memory heap between web workers. This along with primitives for low level atomics and futex support enables Emscripten to implement support for the Pthreads (POSIX threads) API.

That is, you can compile multithreaded C and C++ code to asm.js.

Discussion on how to best bring multi-threading to WebAssembly is ongoing[8]. Given that web workers are relatively heavyweight, it is possible that WebAssembly will introduce lightweight threads. You can also see that threads are on the roadmap for WebAssembly's future[9].

6.5.3 Sharing data other than integers

At the moment, only Arrays of integers (up to 32 bits long) can be shared. That means that the only way of sharing other kinds of data is by encoding them as integers. Tools that may help include:

- `TextEncoder` and `TextDecoder`[10]: The former converts strings to instances of `Uint8Array`. The latter does the opposite.
- stringview.js[11]: a library that handles strings as arrays of characters. Uses Array Buffers.
- FlatJS[12]: enhances JavaScript with ways of storing complex data structures (structs, classes and arrays) in flat memory (`ArrayBuffer` and `SharedArrayBuffer`). JavaScript+FlatJS is compiled to plain JavaScript. JavaScript dialects (TypeScript etc.) are supported.
- TurboScript[13]: is a JavaScript dialect for fast parallel programming. It compiles to asm.js and WebAssembly.

Eventually, there will probably be additional – higher-level – mechanisms for sharing data. And experiments will continue to figure out what these mechanisms should look like.

[7] https://kripken.github.io/emscripten-site/docs/porting/pthreads.html
[8] https://github.com/WebAssembly/design/issues/104
[9] http://webassembly.org/docs/future-features/
[10] https://encoding.spec.whatwg.org/#api
[11] https://github.com/madmurphy/stringview.js
[12] https://github.com/lars-t-hansen/flatjs
[13] https://dump.01alchemist.com/2016/12/31/future-webhpc-parallel-programming-with-javascript-the-new-era-about-to-begin/

6.5.4 How much faster is code that uses Shared Array Buffers?

Lars T. Hansen has written two implementations of the Mandelbrot algorithm (as documented in his article "A Taste of JavaScript's New Parallel Primitives[14]" where you can try them out online): A serial version and a parallel version that uses multiple web workers. For up to 4 web workers (and therefore processor cores), speed-up improves almost linearly, from 6.9 frames per seconds (1 web worker) to 25.4 frames per seconds (4 web workers). More web workers bring additional performance improvements, but more modest ones.

Hansen notes that the speed-ups are impressive, but going parallel comes at the cost of the code being more complex.

6.6 Example

Let's look at a more comprehensive example. Its code is available on GitHub, in the repository `shared-array-buffer-demo`[15]. And you can run it online.[16]

6.6.1 Using a shared lock

In the main thread, we set up shared memory so that it encodes a closed lock and send it to a worker (line A). Once the user clicks, we open the lock (line B).

```
// main.js

// Set up the shared memory
const sharedBuffer = new SharedArrayBuffer(
    1 * Int32Array.BYTES_PER_ELEMENT);
const sharedArray = new Int32Array(sharedBuffer);

// Set up the lock
Lock.initialize(sharedArray, 0);
const lock = new Lock(sharedArray, 0);
lock.lock(); // writes to sharedBuffer

worker.postMessage({sharedBuffer}); // (A)

document.getElementById('unlock').addEventListener(
    'click', event => {
        event.preventDefault();
        lock.unlock(); // (B)
    });
```

[14] https://hacks.mozilla.org/2016/05/a-taste-of-javascripts-new-parallel-primitives/
[15] https://github.com/rauschma/shared-array-buffer-demo
[16] https://rauschma.github.io/shared-array-buffer-demo/

In the worker, we set up a local version of the lock (whose state is shared with the main thread via a Shared Array Buffer). In line B, we wait until the lock is unlocked. In lines A and C, we send text to the main thread, which displays it on the page for us (how it does that is not shown in the previous code fragment). That is, we are using `self.postMessage()` much like `console.log()` in these two lines.

```
// worker.js

self.addEventListener('message', function (event) {
    const {sharedBuffer} = event.data;
    const lock = new Lock(new Int32Array(sharedBuffer), 0);

    self.postMessage('Waiting for lock...'); // (A)
    lock.lock(); // (B) blocks!
    self.postMessage('Unlocked'); // (C)
});
```

It is noteworthy that waiting for the lock in line B stops the complete worker. That is real blocking, which hasn't existed in JavaScript until now (`await` in async functions is an approximation).

6.6.2 Implementing a shared lock

Next, we'll look at an ES6-ified version of a `Lock` implementation by Lars T. Hansen[17] that is based on `SharedArrayBuffer`.

In this section, we'll need (among others) the following `Atomics` function:

- `Atomics.compareExchange(ta : TypedArray<T>, index, expectedValue, replacementValue) : T`
  If the current element of `ta` at `index` is `expectedValue`, replace it with `replacementValue`. Return the previous (or unchanged) element at `index`.

The implementation starts with a few constants and the constructor:

```
const UNLOCKED = 0;
const LOCKED_NO_WAITERS = 1;
const LOCKED_POSSIBLE_WAITERS = 2;

// Number of shared Int32 locations needed by the lock.
const NUMINTS = 1;

class Lock {
```

[17] https://github.com/lars-t-hansen/parlib-simple/blob/master/src/lock.js

```
    /**
     * @param iab an Int32Array wrapping a SharedArrayBuffer
     * @param ibase an index inside iab, leaving enough room for NUMINTS
     */
    constructor(iab, ibase) {
        // OMITTED: check parameters
        this.iab = iab;
        this.ibase = ibase;
    }
```

The constructor mainly stores its parameters in instance properties.

The method for locking looks as follows.

```
/**
 * Acquire the lock, or block until we can. Locking is not recursive:
 * you must not hold the lock when calling this.
 */
lock() {
    const iab = this.iab;
    const stateIdx = this.ibase;
    var c;
    if ((c = Atomics.compareExchange(iab, stateIdx, // (A)
    UNLOCKED, LOCKED_NO_WAITERS)) !== UNLOCKED) {
        do {
            if (c === LOCKED_POSSIBLE_WAITERS // (B)
            || Atomics.compareExchange(iab, stateIdx,
            LOCKED_NO_WAITERS, LOCKED_POSSIBLE_WAITERS) !== UNLOCKED) {
                Atomics.wait(iab, stateIdx, // (C)
                    LOCKED_POSSIBLE_WAITERS, Number.POSITIVE_INFINITY);
            }
        } while ((c = Atomics.compareExchange(iab, stateIdx,
        UNLOCKED, LOCKED_POSSIBLE_WAITERS)) !== UNLOCKED);
    }
}
```

In line A, we change the lock to `LOCKED_NO_WAITERS` if its current value is `UNLOCKED`. We only enter the then-block if the lock is already locked (in which case `compareExchange()` did not change anything).

In line B (inside a `do-while` loop), we check if the lock is locked with waiters or not unlocked. Given that we are about to wait, the `compareExchange()` also switches to `LOCKED_POSSIBLE_-WAITERS` if the current value is `LOCKED_NO_WAITERS`.

In line C, we wait if the lock value is `LOCKED_POSSIBLE_WAITERS`. The last parameter, `Num-ber.POSITIVE_INFINITY`, means that waiting never times out.

After waking up, we continue the loop if we are not unlocked. `compareExchange()` also switches to `LOCKED_POSSIBLE_WAITERS` if the lock is `UNLOCKED`. We use `LOCKED_POSSIBLE_WAITERS` and not `LOCKED_NO_WAITERS`, because we need to restore this value after `unlock()` temporarily set it to `UNLOCKED` and woke us up.

The method for unlocking looks as follows.

```
    /**
     * Unlock a lock that is held.  Anyone can unlock a lock that
     * is held; nobody can unlock a lock that is not held.
     */
    unlock() {
        const iab = this.iab;
        const stateIdx = this.ibase;
        var v0 = Atomics.sub(iab, stateIdx, 1); // A

        // Wake up a waiter if there are any
        if (v0 !== LOCKED_NO_WAITERS) {
            Atomics.store(iab, stateIdx, UNLOCKED);
            Atomics.wake(iab, stateIdx, 1);
        }
    }

    // ···
}
```

In line A, `v0` gets the value that `iab[stateIdx]` had *before* 1 was subtracted from it. The subtraction means that we go (e.g.) from `LOCKED_NO_WAITERS` to `UNLOCKED` and from `LOCKED_-POSSIBLE_WAITERS` to `LOCKED`.

If the value was previously `LOCKED_NO_WAITERS` then it is now `UNLOCKED` and everything is fine (there is no one to wake up).

Otherwise, the value was either `LOCKED_POSSIBLE_WAITERS` or `UNLOCKED`. In the former case, we are now unlocked and must wake up someone (who will usually lock again). In the latter case, we must fix the illegal value created by subtraction and the `wake()` simply does nothing.

6.6.3 Conclusion for the example

This gives you a rough idea how locks based on `SharedArrayBuffer` work. Keep in mind that multithreaded code is notoriously difficult to write, because things can change at any time. Case in point: `lock.js` is based on a paper documenting a futex implementation for the Linux kernel. And the title of that paper is "Futexes are tricky[18]" (PDF).

If you want to go deeper into parallel programming with Shared Array Buffers, take a look at `synchronic.js`[19] and the document it is based on (PDF)[20].

[18]http://www.akkadia.org/drepper/futex.pdf
[19]https://github.com/lars-t-hansen/parlib-simple/blob/master/src/synchronic.js
[20]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4195.pdf

6.7 The API for shared memory and atomics

6.7.1 `SharedArrayBuffer`

Constructor:

- `new SharedArrayBuffer(length)`
  Create a buffer for `length` bytes.

Static property:

- `get SharedArrayBuffer[Symbol.species]`
  Returns `this` by default. Override to control what `slice()` returns.

Instance properties:

- `get SharedArrayBuffer.prototype.byteLength()`
  Returns the length of the buffer in bytes.
- `SharedArrayBuffer.prototype.slice(start, end)`
  Create a new instance of `this.constructor[Symbol.species]` and fill it with the bytes at the indices from (including) `start` to (excluding) `end`.

6.7.2 `Atomics`

The main operand of `Atomics` functions must be an instance of `Int8Array`, `Uint8Array`, `Int16Array`, `Uint16Array`, `Int32Array` or `Uint32Array`. It must wrap a `SharedArrayBuffer`.

All functions perform their operations atomically. The ordering of store operations is fixed and can't be reordered by compilers or CPUs.

6.7.2.1 Loading and storing

- `Atomics.load(ta : TypedArray<T>, index) : T`
  Read and return the element of `ta` at `index`.
- `Atomics.store(ta : TypedArray<T>, index, value : T) : T`
  Write `value` to `ta` at `index` and return `value`.
- `Atomics.exchange(ta : TypedArray<T>, index, value : T) : T`
  Set the element of `ta` at `index` to `value` and return the previous value at that index.

- `Atomics.compareExchange(ta : TypedArray<T>, index, expectedValue, replacement-Value) : T`
  If the current element of `ta` at `index` is `expectedValue`, replace it with `replacementValue`. Return the previous (or unchanged) element at `index`.

6.7.2.2 Simple modification of Typed Array elements

Each of the following functions changes a Typed Array element at a given index: It applies an operator to the element and a parameter and writes the result back to the element. It returns *the original value* of the element.

- `Atomics.add(ta : TypedArray<T>, index, value) : T`
  Perform `ta[index] += value` and return the original value of `ta[index]`.
- `Atomics.sub(ta : TypedArray<T>, index, value) : T`
  Perform `ta[index] -= value` and return the original value of `ta[index]`.
- `Atomics.and(ta : TypedArray<T>, index, value) : T`
  Perform `ta[index] &= value` and return the original value of `ta[index]`.
- `Atomics.or(ta : TypedArray<T>, index, value) : T`
  Perform `ta[index] |= value` and return the original value of `ta[index]`.
- `Atomics.xor(ta : TypedArray<T>, index, value) : T`
  Perform `ta[index] ^= value` and return the original value of `ta[index]`.

6.7.2.3 Waiting and waking

Waiting and waking requires the parameter `ta` to be an instance of `Int32Array`.

- `Atomics.wait(ta: Int32Array, index, value, timeout=Number.POSITIVE_INFINITY) : ('not-equal' | 'ok' | 'timed-out')`
  If the current value at `ta[index]` is not `value`, return `'not-equal'`. Otherwise go to sleep until we are woken up via `Atomics.wake()` or until sleeping times out. In the former case, return `'ok'`. In the latter case, return `'timed-out'`. `timeout` is specified in milliseconds. Mnemonic for what this function does: "wait if `ta[index]` is `value`".
- `Atomics.wake(ta : Int32Array, index, count)`
  Wake up `count` workers that are waiting at `ta[index]`.

6.7.2.4 Miscellaneous

- `Atomics.isLockFree(size)`
  This function lets you ask the JavaScript engine if operands with the given `size` (in bytes) can be manipulated without locking. That can inform algorithms whether they want to rely on built-in primitives (`compareExchange()` etc.) or use their own locking. `Atomics.isLockFree(4)` always returns `true`, because that's what all currently relevant supports.

6.8 FAQ

6.8.1 What browsers support Shared Array Buffers?

At the moment, I'm aware of:

- Firefox (50.1.0+): go to `about:config` and set `javascript.options.shared_memory` to `true`
- Safari Technology Preview (Release 21+): enabled by default.
- Chrome Canary (58.0+): There are two ways to switch it on.
  - Via `chrome://flags/` ("Experimental enabled SharedArrayBuffer support in JavaScript")
  - `--js-flags=--harmony-sharedarraybuffer --enable-blink-feature=SharedArrayBuffer`

6.9 Further reading

More information on Shared Array Buffers and supporting technologies:

- "Shared memory – a brief tutorial[21]" by Lars T. Hansen
- "A Taste of JavaScript's New Parallel Primitives[22]" by Lars T. Hansen [a good intro to Shared Array Buffers]
- "SharedArrayBuffer and Atomics Stage 2.95 to Stage 3[23]" (PDF), slides by Shu-yu Guo and Lars T. Hansen (2016-11-30) [slides accompanying the ES proposal]
- "The Basics of Web Workers[24]" by Eric Bidelman [an introduction to web workers]

Other JavaScript technologies related to parallelism:

- "The Path to Parallel JavaScript[25]" by Dave Herman [a general overview of where JavaScript is heading after the abandonment of PJS]
- "Write massively-parallel GPU code for the browser with WebGL[26]" by Steve Sanderson [fascinating talk that explains how to get WebGL to do computations for you on the GPU]

Background on parallelism:

- "Concurrency is not parallelism[27]" by Rob Pike [Pike uses the terms "concurrency" and "parallelism" slightly differently than I do in this chapter, providing an interesting complementary view]

Acknowledgement: I'm very grateful to Lars T. Hansen for reviewing this chapter and for answering my `SharedArrayBuffer`-related questions.

[21] https://github.com/tc39/ecmascript_sharedmem/blob/master/TUTORIAL.md
[22] https://hacks.mozilla.org/2016/05/a-taste-of-javascripts-new-parallel-primitives/
[23] http://tc39.github.io/ecmascript_sharedmem/presentation-nov-2016.pdf
[24] https://www.html5rocks.com/en/tutorials/workers/basics/
[25] https://blog.mozilla.org/javascript/2015/02/26/the-path-to-parallel-javascript/
[26] http://blog.stevensanderson.com/2014/06/11/write-massively-parallel-gpu-code-for-the-browser-with-webgl-ndc-2014/
[27] https://blog.golang.org/concurrency-is-not-parallelism

7. `Object.entries()` and `Object.values()`

This chapter describes the ECMAScript 2017 feature "Object.values/Object.entries[1]" by Jordan Harband.

7.1 Overview

7.1.1 `Object.entries()`

```
let obj = { one: 1, two: 2 };
for (let [k,v] of Object.entries(obj)) {
    console.log(`${JSON.stringify(k)}: ${JSON.stringify(v)}`);
}
// Output:
// "one": 1
// "two": 2
```

7.1.2 `Object.values()`

```
> Object.values({ one: 1, two: 2 })
[ 1, 2 ]
```

7.2 `Object.entries()`

This method has the following signature:

```
Object.entries(value : any) : Array<[string,any]>
```

If a JavaScript data structure has keys and values then an *entry* is a key-value pair, encoded as a 2-element Array. `Object.entries(x)` coerces `x` to an Object and returns the entries of its enumerable own string-keyed properties, in an Array:

```
> Object.entries({ one: 1, two: 2 })
[ [ 'one', 1 ], [ 'two', 2 ] ]
```

Properties, whose keys are symbols, are ignored:

[1] https://github.com/tc39/proposal-object-values-entries

```
> Object.entries({ [Symbol()]: 123, foo: 'abc' });
[ [ 'foo', 'abc' ] ]
```

`Object.entries()` finally gives us a way to iterate over the properties of an object (read here why objects aren't iterable by default[2]):

```
let obj = { one: 1, two: 2 };
for (let [k,v] of Object.entries(obj)) {
    console.log(`${JSON.stringify(k)}: ${JSON.stringify(v)}`);
}
// Output:
// "one": 1
// "two": 2
```

7.2.1 Setting up Maps via `Object.entries()`

`Object.entries()` also lets you set up a Map via an object. This is more concise than using an Array of 2-element Arrays, but keys can only be strings.

```
let map = new Map(Object.entries({
    one: 1,
    two: 2,
}));
console.log(JSON.stringify([...map]));
    // [["one",1],["two",2]]
```

7.2.2 FAQ: `Object.entries()`

- **Why is the return value of `Object.entries()` an Array and not an iterator?**
  The relevant precedent in this case is `Object.keys()`, not, e.g., `Map.prototype.entries()`.
- **Why does `Object.entries()` only return the enumerable own string-keyed properties?**
  Again, this is done to be consistent with `Object.keys()`. That method also ignores properties whose keys are symbols. Eventually, there may be a method `Reflect.ownEntries()` that returns all own properties.

7.3 `Object.values()`

`Object.values()` has the following signature:

[2] http://exploringjs.com/es6/ch_iteration.html#sec_plain-objects-not-iterable

```
Object.values(value : any) : Array<any>
```

It works much like `Object.entries()`, but, as its name suggests, it only returns the values of the own enumerable string-keyed properties:

```
> Object.values({ one: 1, two: 2 })
[ 1, 2 ]
```

8. New string methods: `padStart` and `padEnd`

This chapter explains the ECMAScript 2017 feature “String padding[1]” by Jordan Harband & Rick Waldron.

8.1 Overview

ECMAScript 2017 has two new string methods:

```
> 'x'.padStart(5, 'ab')
'ababx'
> 'x'.padEnd(5, 'ab')
'xabab'
```

8.2 Why pad strings?

Use cases for padding strings include:

- Displaying tabular data in a monospaced font.
- Adding a count or an ID to a file name or a URL: `'file 001.txt'`
- Aligning console output: `'Test 001: ✓'`
- Printing hexadecimal or binary numbers that have a fixed number of digits: `'0x00FF'`

8.3 `String.prototype.padStart(maxLength, fillString=' ')`

This method (possibly repeatedly) prefixes the receiver with `fillString`, until its length is `maxLength`:

```
> 'x'.padStart(5, 'ab')
'ababx'
```

If necessary, a fragment of `fillString` is used so that the result’s length is exactly `maxLength`:

[1]https://github.com/tc39/proposal-string-pad-start-end

```
> 'x'.padStart(4, 'ab')
'abax'
```

If the receiver is as long as, or longer than, `maxLength`, it is returned unchanged:

```
> 'abcd'.padStart(2, '#')
'abcd'
```

If `maxLength` and `fillString.length` are the same, `fillString` becomes a mask into which the receiver is inserted, at the end:

```
> 'abc'.padStart(10, '0123456789')
'0123456abc'
```

If you omit `fillString`, a string with a single space in it is used (`' '`):

```
> 'x'.padStart(3)
'  x'
```

8.3.1 A simple implementation of `padStart()`

The following implementation gives you a rough idea of how `padStart()` works, but isn't completely spec-compliant (for a few edge cases).

```
String.prototype.padStart =
function (maxLength, fillString=' ') {
    let str = String(this);
    if (str.length >= maxLength) {
        return str;
    }

    fillString = String(fillString);
    if (fillString.length === 0) {
        fillString = ' ';
    }

    let fillLen = maxLength - str.length;
    let timesToRepeat = Math.ceil(fillLen / fillString.length);
    let truncatedStringFiller = fillString
        .repeat(timesToRepeat)
        .slice(0, fillLen);
    return truncatedStringFiller + str;
};
```

8.4 `String.prototype.padEnd(maxLength, fillString=' ')`

`padEnd()` works similarly to `padStart()`, but instead of inserting the repeated `fillString` at the start, it inserts it at the end:

```
> 'x'.padEnd(5, 'ab')
'xabab'
> 'x'.padEnd(4, 'ab')
'xaba'
> 'abcd'.padEnd(2, '#')
'abcd'
> 'abc'.padEnd(10, '0123456789')
'abc0123456'
> 'x'.padEnd(3)
'x  '
```

Only the last line of an implementation of `padEnd()` is different, compared to the implementation of `padStart()`:

```
return str + truncatedStringFiller;
```

8.5 FAQ: `padStart` and `padEnd`

8.5.1 Why aren't the padding methods called `padLeft` and `padRight`?

For bidirectional or right-to-left languages, the terms `left` and `right` don't work well. Therefore, the naming of `padStart` and `padEnd` follows the existing names `startsWith` and `endsWith`.

9. `Object.getOwnPropertyDescriptors()`

This chapter explains the ECMAScript 2017 feature "`Object.getOwnPropertyDescriptors()`[1]" by Jordan Harband and Andrea Giammarchi.

9.1 Overview

`Object.getOwnPropertyDescriptors(obj)` returns the property descriptors of all own properties of `obj`, in an Array:

```
const obj = {
    [Symbol('foo')]: 123,
    get bar() { return 'abc' },
};
console.log(Object.getOwnPropertyDescriptors(obj));

// Output:
// { [Symbol('foo')]:
//    { value: 123,
//      writable: true,
//      enumerable: true,
//      configurable: true },
//   bar:
//    { get: [Function: bar],
//      set: undefined,
//      enumerable: true,
//      configurable: true } }
```

9.2 `Object.getOwnPropertyDescriptors()`

`Object.getOwnPropertyDescriptors(obj)` accepts an object `obj` and returns an object `result`:

- For each own (non-inherited) property of `obj`, it adds a property to `result` whose key is the same and whose value is the the former property's *descriptor*.

Property descriptors describe the *attributes* of a property (its value, whether it is writable, etc.). For more information, consult Sect. "Property Attributes and Property Descriptors[2]" in "Speaking JavaScript".

This is an example of using `Object.getOwnPropertyDescriptors()`:

[1] https://tc39.github.io/proposal-object-getownpropertydescriptors/
[2] http://speakingjs.com/es5/ch17.html#property_attributes

```
const obj = {
    [Symbol('foo')]: 123,
    get bar() { return 'abc' },
};
console.log(Object.getOwnPropertyDescriptors(obj));

// Output:
// { [Symbol('foo')]:
//    { value: 123,
//      writable: true,
//      enumerable: true,
//      configurable: true },
//   bar:
//    { get: [Function: bar],
//      set: undefined,
//      enumerable: true,
//      configurable: true } }
```

This is how you would implement `Object.getOwnPropertyDescriptors()`:

```
function getOwnPropertyDescriptors(obj) {
    const result = {};
    for (let key of Reflect.ownKeys(obj)) {
        result[key] = Object.getOwnPropertyDescriptor(obj, key);
    }
    return result;
}
```

9.3 Use cases for `Object.getOwnPropertyDescriptors()`

9.3.1 Use case: copying properties into an object

Since ES6, JavaScript already has a tool method for copying properties: `Object.assign()`. However, this method uses simple get and set operations to copy a property whose key is `key`:

```
const value = source[key]; // get
target[key] = value; // set
```

That means that it doesn't properly copy properties with non-default attributes (getters, setters, non-writable properties, etc.). The following example illustrates this limitation. The object `source` has a setter whose key is `foo`:

```
const source = {
    set foo(value) {
        console.log(value);
    }
};
console.log(Object.getOwnPropertyDescriptor(source, 'foo'));
// { get: undefined,
//   set: [Function: foo],
//   enumerable: true,
//   configurable: true }
```

Using `Object.assign()` to copy property `foo` to object `target` fails:

```
const target1 = {};
Object.assign(target1, source);
console.log(Object.getOwnPropertyDescriptor(target1, 'foo'));
// { value: undefined,
//   writable: true,
//   enumerable: true,
//   configurable: true }
```

Fortunately, using `Object.getOwnPropertyDescriptors()` together with `Object.defineProperties()`[3] works:

```
const target2 = {};
Object.defineProperties(target2, Object.getOwnPropertyDescriptors(source));
console.log(Object.getOwnPropertyDescriptor(target2, 'foo'));
// { get: undefined,
//   set: [Function: foo],
//   enumerable: true,
//   configurable: true }
```

9.3.2 Use case: cloning objects

Shallow cloning is similar to copying properties, which is why `Object.getOwnPropertyDescriptors()` is a good choice here, too.

This time, we use `Object.create()`[4] that has two parameters:

- The first parameter specifies the prototype of the object it returns.
- The optional second parameter is a property descriptor collection like the ones returned by `Object.getOwnPropertyDescriptors()`.

[3]http://speakingjs.com/es5/ch17.html#Object.defineProperties
[4]http://speakingjs.com/es5/ch17.html#Object.create

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

9.3.3 Use case: cross-platform object literals with arbitrary prototypes

The syntactically nicest way of using an object literal to create an object with an arbitrary prototype `prot` is to use the special property `__proto__`:

```
const obj = {
    __proto__: prot,
    foo: 123,
};
```

Alas, that feature is only guaranteed to be there in browsers. The common work-around is `Object.create()` and assignment:

```
const obj = Object.create(prot);
obj.foo = 123;
```

But you can also use `Object.getOwnPropertyDescriptors()`:

```
const obj = Object.create(
    prot,
    Object.getOwnPropertyDescriptors({
        foo: 123,
    })
);
```

Another alternative is `Object.assign()`:

```
const obj = Object.assign(
    Object.create(prot),
    {
        foo: 123,
    }
);
```

9.4 Pitfall: copying methods that use `super`

A method that uses `super` is firmly connected with its *home object* (the object it is stored in). There is currently no way to copy or move such a method to a different object.

10. Trailing commas in function parameter lists and calls

The ECMAScript 2017 feature "Trailing commas in function parameter lists and calls[1]" was proposed by Jeff Morrison.

10.1 Overview

Trailing commas in parameter definitions are now legal:

```
function foo(
    param1,
    param2,
) {}
```

Similarly, trailing commas in function calls are now also legal:

```
foo(
    'abc',
    'def',
);
```

10.2 Trailing commas in object literals and Array literals

Trailing commas are ignored in object literals:

```
let obj = {
    first: 'Jane',
    last: 'Doe',
};
```

And they are also ignored in Array literals:

[1] https://github.com/tc39/proposal-trailing-function-commas

```
let arr = [
    'red',
    'green',
    'blue',
];
console.log(arr.length); // 3
```

Why is that useful? There are two benefits.

First, rearranging items is simpler, because you don't have to add and remove commas if the last item changes its position.

Second, it helps version control systems with tracking what actually changed. For example, going from:

```
[
    'foo'
]
```

to:

```
[
    'foo',
    'bar'
]
```

leads to both the line with `'foo'` and the line with `'bar'` being marked as changed, even though the only real change is the latter line being added.

10.3 Feature: allow trailing commas in parameter definitions and function calls

Given the benefits of optional and ignored trailing commas, the feature brings them to parameter definitions and function calls.

For example, the following function declaration causes a SyntaxError in ECMAScript 6, but is now legal:

```
function foo(
    param1,
    param2,
) {}
```

Similarly, this invocation of `foo()` is now syntactically legal:

```
foo(
    'abc',
    'def',
);
```