ES2017

async / await

by Denis Vlassenko aka vladen

ECMAScript (ES)

- Is a trademark
- Scripting-language specification
- Standardized by ECMA International
  in ECMA-262 and ISO/IEC 16262

Well-known implementations:

- JavaScript
- JScript
- ActionScript

Brendan Eich
JavaScript developer @ Netscape
Mozilla co-founder, former CTO, CEO

"ECMAScript was always an unwanted trade name that sounds like a skin disease."

ECMA

European Computer Manufacturers Association - international private (membership-based) non-profit standards organization for information and communication systems, founded in 1961, located in Geneva.

Some relevant standards:

- **ECMA-262 – ECMAScript Language Specification**
- ECMA-334 – C# Language Specification
- ECMA-335 – Common Language Infrastructure
- ECMA-402 – ECMAScript Internationalization API Specification
- **ECMA-404 – JSON**

ecma INTERNATIONAL

TC39

The standards committee of ECMAScript

TC39 - ECMAScript® (formerly TC39-TG1)

Scope - Programme of work - Activities - TC39-Royalty Free Task Group

Scope:

Standardization of the general purpose, cross platform, vendor-neutral programming language ECMAScript. This includes the language syntax, semantics, and libraries and complementary technologies that support the language.

Programme of work:

1. To maintain and update the standard for the ECMAScript programming language.
2. To identify, develop and maintain standards for libraries that extend the capabilities of ECMAScript.
3. To develop test suites that may be used to verify correct implementation of these standards.
4. To contribute selected standards to ISO/IEC JTC 1.
5. To evaluate and consider proposals for complementary or additional technologies.


Erik Arvidsson — Google


Gavin Barraclough — Apple


Nebojša Ćirić — Google


Doug Crockford — eBay


Brendan Eich — Mozilla


Stephan Herhut — Intel


Dave Herman — Mozilla


Luke Hoban — Microsoft

ecma INTERNATIONAL

ECMAScript Language Specification

- **Standard ECMA-262, ECMAScript® 2015 Language Specification**
  http://www.ecma-international.org/ecma-262/6.0/

- **Draft ECMA-262, ECMAScript® 2017 Language Specification**
  https://tc39.github.io/ecma262/

Bits of ECMAScript History

[1997] First edition

…

[2009] Fifth edition, JavaScript: strict mode, getters and setters, JSON

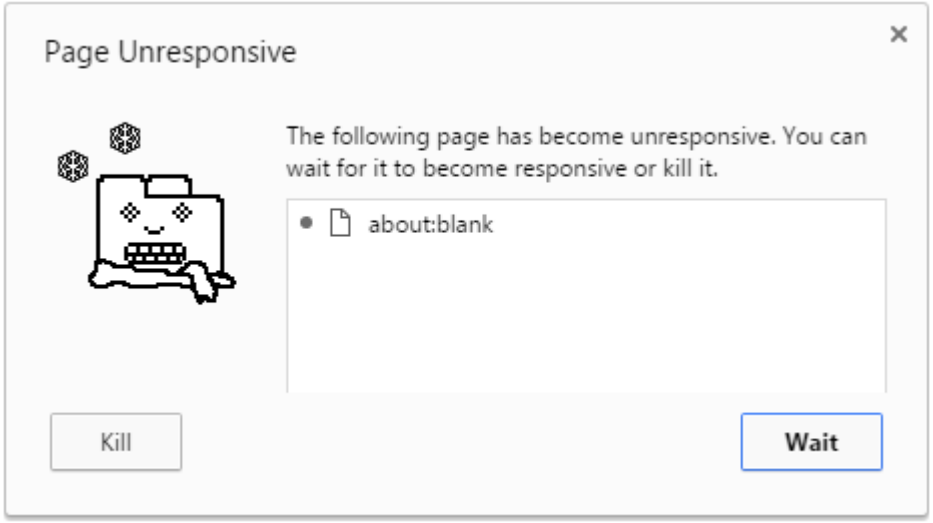
[2015] Sixth Edition, ES2015: **iterators (for/of), generators**, modules, classes, arrow functions, promises, reflection, proxies, collections, typed arrays, binary data, …

> *The introduction of promises and generators in ECMAScript presents an opportunity to dramatically improve the language-level model for writing asynchronous code.*

[next] Seventh Edition: **syntactic integration with promises** (**async functions**), aspects, concurrency, observable streams, value types, …

Why do we need asynchrony?

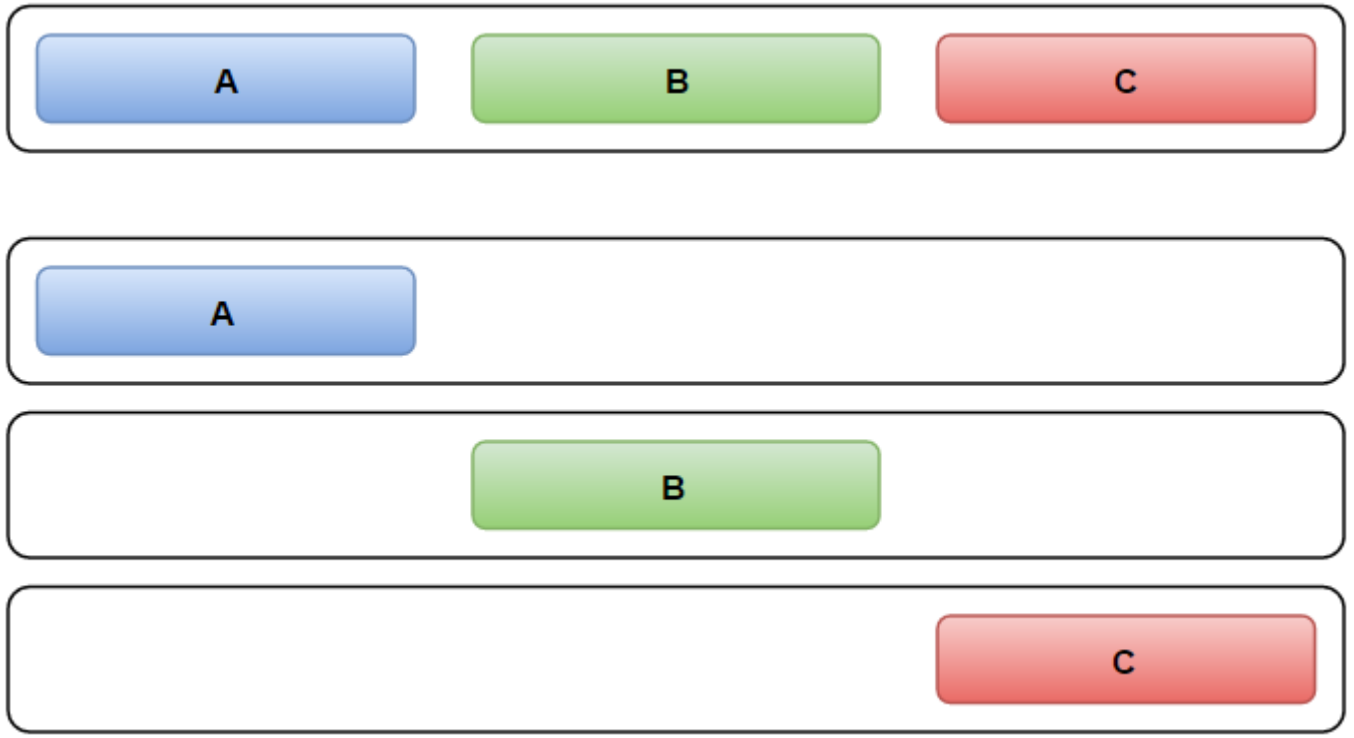
Because we prefer responsive applications, not blocking on long-running and background operations.



And we already have many operating systems supporting multitasking and multiprocessing.

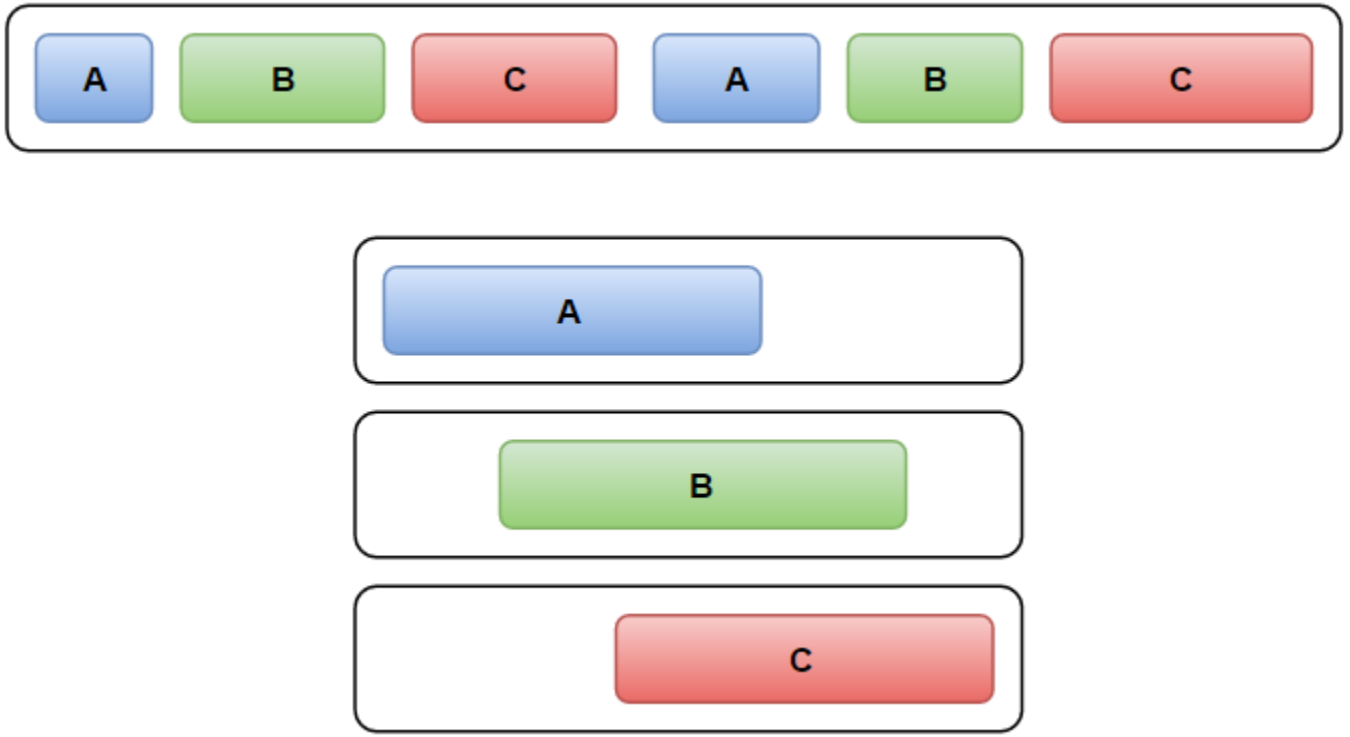
Synchronous/serial execution

When you execute something synchronously, you **wait for it to finish** before moving on to another task.



Asynchronous/parallel execution

When you execute something asynchronously, you can move on to another task **before it finishes**.



Some asynchronous APIs in JavaScript

Client

- Timers
- XHR/fetch
- Animations
- UX/Platform Events
- IndexedDB
- Notification API
- WebSockets
- WebRTC
- etc.

Server

- Timers
- FS
- Stream
- Net
- DNS
- HTTP/HTTPS
- UDP
- Child process
- etc.

Asynchronous Strategies

Simple: Fire and forget (no explicit result is required)

- Nothing special, the platform hides all the magic

Complex: Fire and await (explicit result is required)

- Need special mechanism able to eventually notify us with the result

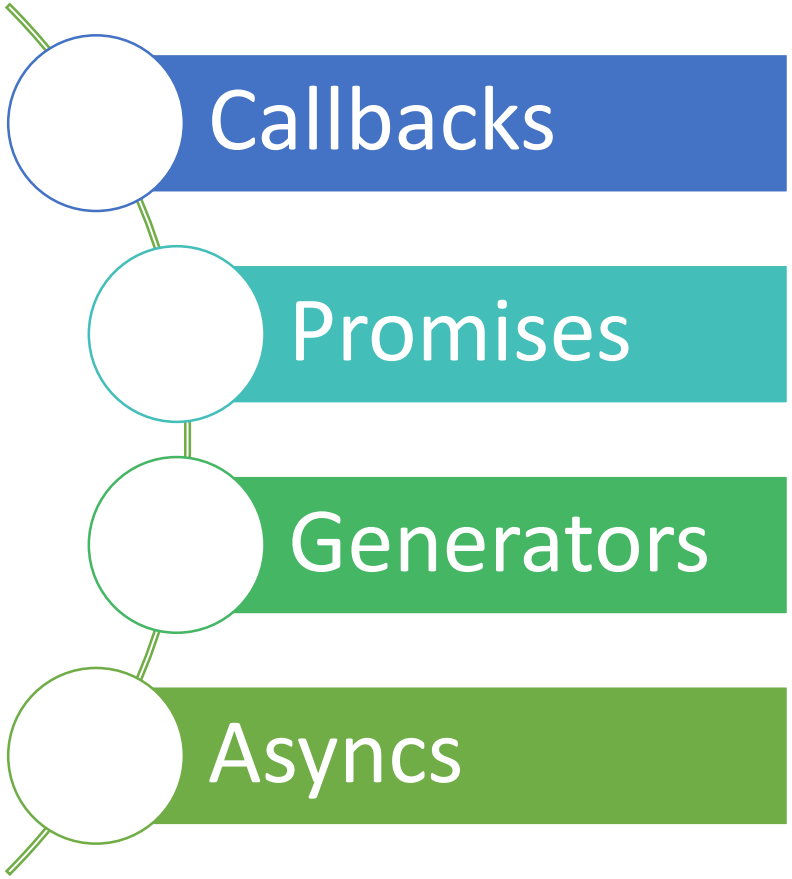
Continuation-passing style

A CPS function takes an extra argument: an explicit "continuation". When this function has computed its result value, it "returns" it by calling the continuation function with this value as the argument.

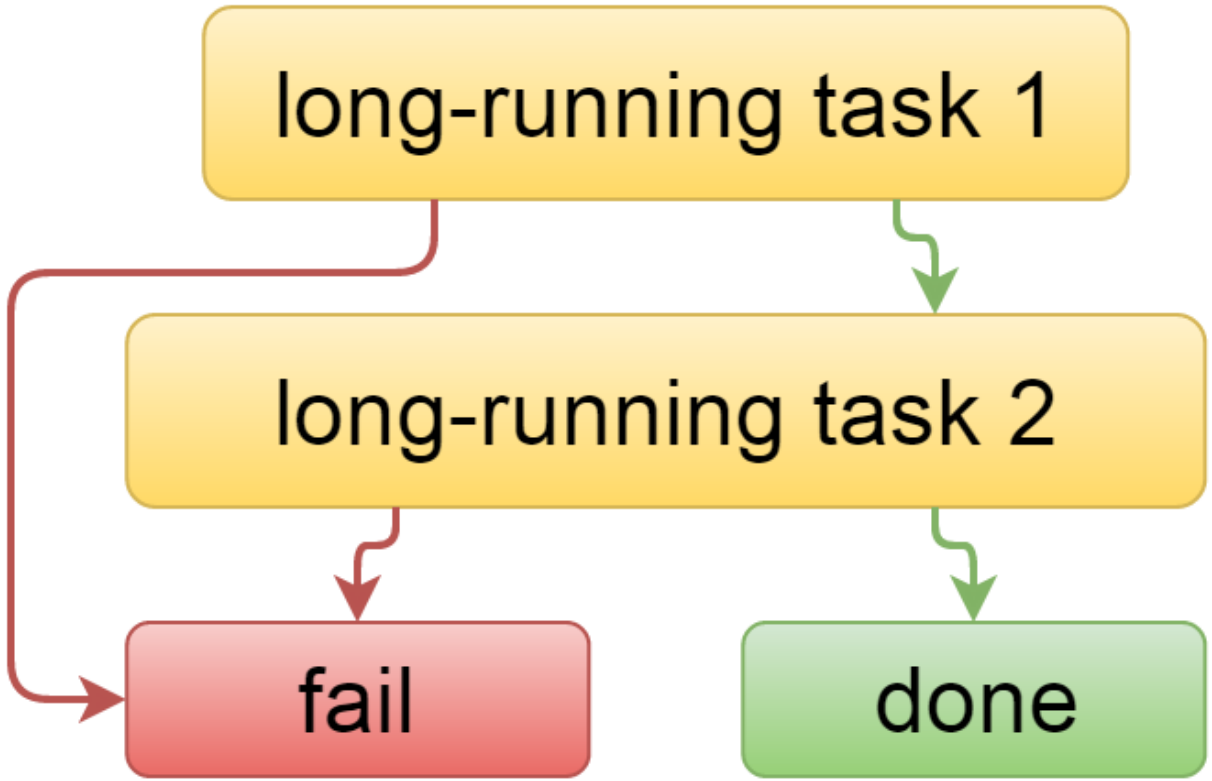
```
> var add = (a, c) => (b) => c(a + b)
    , mul = (a, c) => (b) => c(a * b)
    , ret = (v) => v;
  var fun = add(2, mul(2, ret));
  fun(2);
< 8
```

*In JavaScript we know such "continuation" functions as **callbacks**.*

Journey to asynchronous JavaScript



Imitation of asynchronous process



Node.js callbacks

Node.js uses an event-driven, **non-blocking I/O model**...

Non-blocking means asynchronous.

```
function callback(error, result) {
  if (error) {
    // handle error
  }
  else {
    // handle result
  }
}
```

Callback-based approach

```
function done(timestamp) {
  console.info('done', Date.now() - timestamp, 'ms');
}
function fail(error) {
  console.error('fail', error);
}
function task(timestamp, callback) {
  console.log('task', Date.now() - timestamp, 'ms');
  setTimeout(callback.bind(this, null, timestamp), 1000);
}
```

Callback-based approach

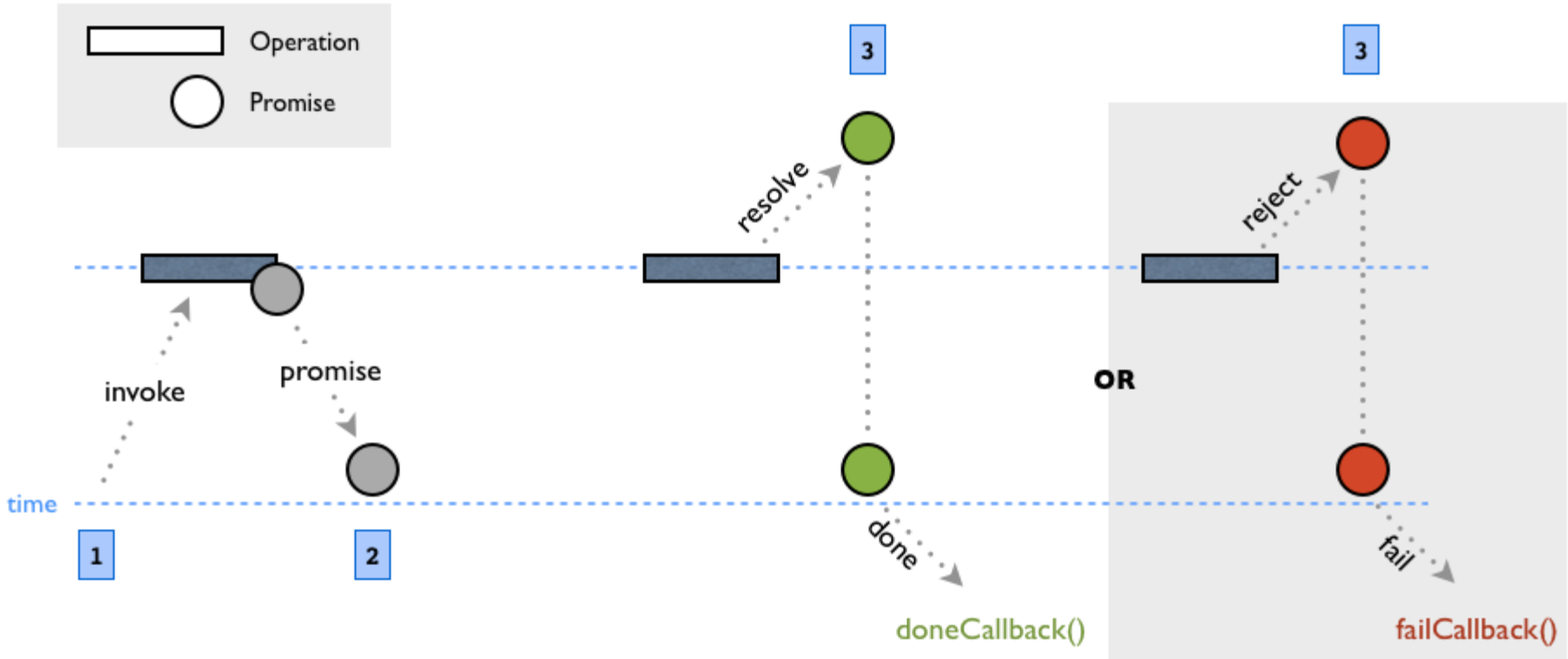
Callback Hell

```
> task(Date.now(), function (error, timestamp) {
    if (error) fail(error);
    else task(timestamp, function(error, timestamp) {
      if (error) fail(error);
      else done(timestamp);
    });
  });
  task 0 ms
< undefined
  task 1007 ms
ℹ done 2010 ms
```

This approach has been perfectly improved in the "async" open source library:
https://github.com/caolan/async

Promises

A Promise is an object that is used as a placeholder for the **eventual** result of a deferred/**asynchronous** computation.



Promise-based approach

```
function task(timestamp) {
  console.log('task', Date.now() - timestamp, 'ms');
  return new Promise(resolve =>
    setTimeout(() => resolve(timestamp), 1000));
}
```

Promise-based approach

```
> task(Date.now()).then(task).then(done).catch(fail);
  task 0 ms
<· Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]:
  task 1013 ms
ⓘ done 2020 ms
```

Current support for promises

Feature name	Current browser 82%	XS6 98%	CH 49, OP 36[0] 91%	FF 45 86%	Edge 13[3] 80%	Babel + core-js[1] 71%	FF 38 ESR 64%	Edge 12[3] 61%	Traceur 58%
Promise	7/8	8/8	7/8	8/8	8/8	8/8	7/8	7/8	4/8
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
constructor requires new	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Promise.prototype isn't an instance	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Promise.all	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Promise.all, generic iterables	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Promise.race	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Promise.race, generic iterables	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Promise[Symbol.species]	No	Yes	No	Yes	Yes	Yes	No	No	No

http://kangax.github.io/compat-table/es6/

To support legacy platforms It's normal to use polyfill:
https://github.com/stefanpenner/es6-promise

COMPAT ES

Iterators

Iterator objects enable custom **iteration** of arbitrary data containers. In ES2015 iteration is based on these **duck-typed** interfaces:

```
interface IteratorResult {
  done: boolean;
  value: any;
}
interface Iterator {
  next(): IteratorResult;
}
interface Iterable {
  [Symbol.iterator](): Iterator
}
```

Iterators

Without syntactic sugaring an example iterable object might look like:

```
var iterable = new class { // instance of Iterable
  [Symbol.iterator]() {
    let value = -1;
    return { // instance of Iterator
      next() {
        return ++value < 3
          ? { value } // instance of IteratorResult
          : { done: true };
      }
    };
  }
}
```

Iterators

And to use it properly we would type something similar to:

```
> let iterator = iterable[Symbol.iterator]()
    , iteration;
  for(;!(iteration = iterator.next()).done;)
    console.log(iteration.value);
  0
  1
  2
< undefined
```

Generators and for…of statement

Generators simplify iterator-authoring using **function*** statement. It returns an object of the following interface:

```
interface Generator extends Iterator {
    next(value?: any): IteratorResult;
    throw(exception: any);
}
```

Internally generator function allowed to use **yield** statement to produce iterable values.

The **for…of** statement creates a loop Iterating over iterable objects, including results of generator functions.

Generators and for…of statement

With syntactic sugaring the example iterable object looks cleaner:

```
var iterable = new class {
  get [Symbol.iterator]() {
    return function*() {
      let value = -1;
      while (++value < 3) yield value;
    }
  }
}
```

Generators and for…of statement

…and much more easier to use:

```
> for (let value of iterable)
    console.log(value);
  0
  1
  2
```

But, both generators and iterators are intended to be used for **synchronous** computation. To make them asynchronous we still need **promises**.

Generator-based approach

```
// verbose
function* work(timestamp) {
  timestamp = yield task(timestamp);
  timestamp = yield task(timestamp);
  done(timestamp);
}
// concise
function* work(timestamp) {
  done(yield task(yield task(timestamp)));
}
```

Generator-based approach

```
> spawn(work, Date.now()).catch(fail);
  task 0 ms
< Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undef
  task 1013 ms
i done 2023 ms
```

Generator-based approach

```
const spawn = (generator, ...args) =>
  new Promise((resolve, reject) => {
    const iterator = generator(...args);
    !function proceed(result) {
      const { done, value } = iterator.next(result);
      done
        ? resolve(result)
        : Promise.resolve(value).then(proceed, reject);
    }();
  });
```

This approach has been implemented in many open-source libraries:

- https://github.com/tj/co
- https://github.com/kriskowal/q
- https://github.com/mozilla/task.js
- https://github.com/jmar777/suspend

Current support for generators

Feature name	Current browser 82%	XS6 98%	CH 49, OP 36[0] 91%	FF 45 86%	Edge 13[3] 80%	Babel + core-js[1] 71%	FF 38 ESR 64%	Edge 12[3] 61%	Traceur 58%
generators	23/27	27/27	23/27	25/27	27/27	24/27	20/27	0/27	24/27
basic functionality	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Yes
generator function expressions	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Yes
correct "this" binding	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Yes
can't use "this" with new	No	Yes	No	Yes	Yes	No	No	No	No
sending	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Yes
%GeneratorPrototype%	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Flag	Yes
%GeneratorPrototype% prototype chain	Yes	Yes	Yes	Yes	Yes	Yes	No	Flag	Yes
%GeneratorPrototype%.constructor	Yes	Yes	Yes	Yes	Yes	No	Yes	Flag	No

http://kangax.github.io/compat-table/es6/

To support any legacy platform you need syntax transforming software:
https://babeljs.io/

COMPAT ES

Async-based approach

```
const sleep = delay =>
  new Promise(resolve =>
    setTimeout(resolve, delay));

async function task(timestamp) {
  console.log('next', Date.now() - timestamp, 'ms');
  await sleep(1000);
  return timestamp;
}

const work = async timestamp => {
  done(await task(await task(timestamp)));
}
```

Async-based approach

```
try {
  work(Date.now());
}
catch(error) {
  fail(error);
}
```

Current support for async functions

Feature name	Current browser (12%)	Traceur (9%)	Babel + core-js[1] (48%)	Type-Script + core-js (42%)	es7-shim (15%)	IE 11 (0%)	Edge 12[2] (5%)	Edge 13[2] (9%)
async functions	0/3	3/3	3/3	2/3	0/3	0/3	0/3	0/3
basic support	No	Yes	Yes	Yes[5]	No	No	No	Flag
await support	No	Yes	Yes	Yes[5]	No	No	No	Flag
arrow async functions	No	Yes	Yes	No	No	No	No	Flag

http://kangax.github.io/compat-table/es6/

To support any platform you need syntax transforming software:
https://babeljs.io/

COMPAT ES

BabelJs.io

Babel is a **compiler** for writing next generation JavaScript.

Under the hood it uses the Babylon JavaScript parser and myriad of plugins transforming new language syntax into old one, understandable by legacy platforms.

Pure magic of BabelJs

Without es2015 preset it transforms my code into **generator** function and wraps it with utility function logically similar to **spawn**.

```
let task = function () {
  var ref = _asyncToGenerator(function* (timestamp) {
    console.log('next', Date.now() - timestamp, 'ms');
    yield sleep(1000);
    return timestamp;
  });

  return function task(_x) {
    return ref.apply(this, arguments);
  };
}();

function _asyncToGenerator(fn) { return function () {
```

Pure pure magic of BabelJs

With es2015 preset it also uses **regenerator runtime** by FB to make my code work in legacy environment.

```
var task = function () {
  var ref = _asyncToGenerator(regeneratorRuntime.mark(function _callee(timestamp) {
    return regeneratorRuntime.wrap(function _callee$(_context) {
      while (1) {
        switch (_context.prev = _context.next) {
          case 0:
            console.log('next', Date.now() - timestamp, 'ms');
            _context.next = 3;
            return sleep(1000);

          case 3:
            return _context.abrupt('return', timestamp);

          case 4:
          case 'end':
            return _context.stop();
        }
      }
    }, _callee, this);
  }));

  return function task(_x) {
    return ref.apply(this, arguments);
  };
}();

function _asyncToGenerator(fn) { return function () { var gen = fn.apply(th
```

BABEL

Conclusion

Now, async functions, as a language improvement proposal has already evolved up to 3rd maturity stage, candidate (https://github.com/tc39/ecma262).

- This means that their semantics, syntax and API have been completely described and are fully spec compliant.
- Also this means that ES2017 with extremely high degree of probability will understand async/await keywords.

So, shall we start using this feature right now?