bevacqua / **es6** Public

🌟 ES6 Overview in 350 Bullet Points

ponyfoo.com/articles/es6

MIT license

4.3k stars **261** forks Branches Tags Activity

Star Notifications

Code Issues Pull requests Actions Projects Security Insights

1 Branch **0** Tags Go to file Go to file Code ...

bevacqua Merge pull request #12 from mariusschulz/patch-1

6a11e1d · 7 years ago

license	initial commit	9 years ago
readme.markdown	Fixes tiny typo	7 years ago

ES6 Overview in 350 Bullet Points

My ES6 in Depth series consists of 24 articles covering most syntax changes and features coming in ES6. This article aims to summarize all of those, providing you with practical insight into most of ES6, so that you can quickly get started. I've also linked to the articles in ES6 in Depth so that you can easily go deeper on any topic you're interested in.

I heard you like bullet points, so I made an article containing hundreds of those bad boys. To kick things off, here's a table of contents with all the topics covered. It has bullet points in it -- **obviously**. Note that if you want these concepts to permeate your brain, you'll have a much better time learning the subject by going through the in-depth series and playing around, experimenting with ES6 code yourself.

Please enter your comment here, Markdown formatting is supported.

Table of Contents

Apologies about that long table of contents, and here we go.

Introduction

- ES6 -- also known as Harmony, `es-next` , ES2015 -- is the latest finalized specification of the language
- The ES6 specification was finalized in **June 2015**, *(hence ES2015)*
- Future versions of the specification will follow the `ES[YYYY]` pattern, e.g ES2016 for ES7
  - **Yearly release schedule**, features that don't make the cut take the next train
  - Since ES6 pre-dates that decision, most of us still call it ES6
  - Starting with ES2016 (ES7), we should start using the `ES[YYYY]` pattern to refer to newer versions

  - Top reason for naming scheme is to pressure browser vendors into quickly implementing newest features

(back to table of contents)

Tooling

- To get ES6 working today, you need a **JavaScript-to-JavaScript** *transpiler*
- Transpilers are here to stay
  - They allow you to compile code in the latest version into older versions of the language
  - As browser support gets better, we'll transpile ES2016 and ES2017 into ES6 and beyond
  - We'll need better source mapping functionality
  - They're the most reliable way to run ES6 source code in production today *(although browsers get ES5)*
- Babel *(a transpiler)* has a killer feature: **human-readable output**
- Use `babel` to transpile ES6 into ES5 for static builds
- Use `babelify` to incorporate `babel` into your Gulp, Grunt, or `npm run` build process
- Use Node.js `v4.x.x` or greater as they have decent ES6 support baked in, thanks to `v8`
- Use `babel-node` with any version of `node`, as it transpiles modules into ES5
- Babel has a thriving ecosystem that already supports some of ES2016 and has plugin support
- Read A Brief History of ES6 Tooling

(back to table of contents)

Assignment Destructuring

- `var {foo} = pony` is equivalent to `var foo = pony.foo`
- `var {foo: baz} = pony` is equivalent to `var baz = pony.foo`
- You can provide default values, `var {foo='bar'} = baz` yields `foo: 'bar'` if `baz.foo` is `undefined`
- You can pull as many properties as you like, aliased or not
  - `var {foo, bar: baz} = {foo: 0, bar: 1}` gets you `foo: 0` and `baz: 1`
- You can go deeper. `var {foo: {bar}} = { foo: { bar: 'baz' } }` gets you `bar: 'baz'`
- You can alias that too. `var {foo: {bar: deep}} = { foo: { bar: 'baz' } }` gets you `deep: 'baz'`
- Properties that aren't found yield `undefined` as usual, e.g: `var {foo} = {}`
- Deeply nested properties that aren't found yield an error, e.g: `var {foo: {bar}} = {}`
- It also works for arrays, `var [a, b] = [0, 1]` yields `a: 0` and `b: 1`

- You can skip items in an array, `var [a, , b] = [0, 1, 2]`, getting `a: 0` and `b: 2`
- You can swap without an "*aux*" variable, `[a, b] = [b, a]`
- You can also use destructuring in function parameters
  - Assign default values like `function foo (bar=2) {}`
  - Those defaults can be objects, too `function foo (bar={ a: 1, b: 2 }) {}`
  - Destructure `bar` completely, like `function foo ({ a=1, b=2 }) {}`
  - Default to an empty object if nothing is provided, like `function foo ({ a=1, b=2 } = {}) {}`
- Read ES6 JavaScript Destructuring in Depth

(back to table of contents)

Spread Operator and Rest Parameters

- Rest parameters is a better `arguments`
  - You declare it in the method signature like `function foo (...everything) {}`
  - `everything` is an array with all parameters passed to `foo`
  - You can name a few parameters before `...everything`, like `function foo (bar, ...rest) {}`
  - Named parameters are excluded from `...rest`
  - `...rest` must be the last parameter in the list
- Spread operator is better than magic, also denoted with `...` syntax
  - Avoids `.apply` when calling methods, `fn(...[1, 2, 3])` is equivalent to `fn(1, 2, 3)`
  - Easier concatenation `[1, 2, ...[3, 4, 5], 6, 7]`
  - Casts array-likes or iterables into an array, e.g `[...document.querySelectorAll('img')]`
  - Useful when destructuring too, `[a, , ...rest] = [1, 2, 3, 4, 5]` yields `a: 1` and `rest: [3, 4, 5]`
  - Makes `new + .apply` effortless, `new Date(...[2015, 31, 8])`
- Read ES6 Spread and Butter in Depth

(back to table of contents)

Arrow Functions

- Terse way to declare a function like `param => returnValue`
- Useful when doing functional stuff like `[1, 2].map(x => x * 2)`
- Several flavors are available, might take you some getting used to
  - `p1 => expr` is okay for a single parameter
  - `p1 => expr` has an implicit `return` statement for the provided `expr` expression

    - To return an object implicitly, wrap it in parenthesis `() => ({ foo: 'bar' })` or you'll get **an error**
    - Parenthesis are demanded when you have zero, two, or more parameters, `() => expr` or `(p1, p2) => expr`
    - Brackets in the right-hand side represent a code block that can have multiple statements, `() => {}`
    - When using a code block, there's no implicit `return`, you'll have to provide it -- `() => { return 'foo' }`
- You can't name arrow functions statically, but runtimes are now much better at inferring names for most methods
- Arrow functions are bound to their lexical scope
    - `this` is the same `this` context as in the parent scope
    - `this` can't be modified with `.call`, `.apply`, or similar "*reflection*"-*type* methods
    - `arguments` is also lexically scoped to the nearest normal function; use `(...args)` for local arguments
- Read ES6 Arrow Functions in Depth

(back to table of contents)

Template Literals

- You can declare strings with `` ` `` (backticks), in addition to `"` and `'`
- Strings wrapped in backticks are *template literals*
- Template literals can be multiline
- Template literals allow interpolation like `` `ponyfoo.com is ${rating}` `` where `rating` is a variable
- You can use any valid JavaScript expressions in the interpolation, such as `` `${2 * 3}` `` or `` `${foo()}` ``
- You can use tagged templates to change how expressions are interpolated
    - Add a `fn` prefix to `` fn`foo, ${bar} and ${baz}` ``
    - `fn` is called once with `template, ...expressions`
    - `template` is `['foo, ', ' and ', '']` and `expressions` is `[bar, baz]`
    - The result of `fn` becomes the value of the template literal
    - Possible use cases include input sanitization of expressions, parameter parsing, etc.
- Template literals are almost strictly better than strings wrapped in single or double quotes
- Read ES6 Template Literals in Depth

(back to table of contents)

Object Literals

- Instead of `{ foo: foo }`, you can just do `{ foo }` -- known as a *property value shorthand*
- Computed property names, `{ [prefix + 'Foo']: 'bar' }`, where `prefix: 'moz'`, yields `{ mozFoo: 'bar' }`
- You can't combine computed property names and property value shorthands, `{ [foo] }` is invalid
- Method definitions in an object literal can be declared using an alternative, more terse syntax, `{ foo () {} }`
- See also `Object` section
- Read ES6 Object Literal Features in Depth

(back to table of contents)

Classes

- Not "*traditional*" classes, syntax sugar on top of prototypal inheritance
- Syntax similar to declaring objects, `class Foo {}`
- Instance methods -- *`new Foo().bar`* -- are declared using the short object literal syntax, `class Foo { bar () {} }`
- Static methods -- *`Foo.isPonyFoo()`* -- need a `static` keyword prefix, `class Foo { static isPonyFoo () {} }`
- Constructor method `class Foo { constructor () { /* initialize instance */ } }`
- Prototypal inheritance with a simple syntax `class PonyFoo extends Foo {}`
- Read ES6 Classes in Depth

(back to table of contents)

Let and Const

- `let` and `const` are alternatives to `var` when declaring variables
- `let` is block-scoped instead of lexically scoped to a `function`
- `let` is hoisted to the top of the block, while `var` declarations are hoisted to top of the function
- "Temporal Dead Zone" -- TDZ for short
  - Starts at the beginning of the block where `let foo` was declared
  - Ends where the `let foo` statement was placed in user code *(hoisiting is irrelevant here)*
  - Attempts to access or assign to `foo` within the TDZ *(before the `let foo` statement is reached)* result in an error
  - Helps prevent mysterious bugs when a variable is manipulated before its declaration is reached
- `const` is also block-scoped, hoisted, and constrained by TDZ semantics

- `const` variables must be declared using an initializer, `const foo = 'bar'`
- Assigning to `const` after initialization fails silently (or **loudly** -- *with an exception* -- under strict mode)
- `const` variables don’t make the assigned value immutable
  - `const foo = { bar: 'baz' }` means `foo` will always reference the right-hand side object
  - `const foo = { bar: 'baz' }; foo.bar = 'boo'` won't throw
- Declaration of a variable by the same name will throw
- Meant to fix mistakes where you reassign a variable and lose a reference that was passed along somewhere else
- In ES6, **functions are block scoped**
  - Prevents leaking block-scoped secrets through hoisting, `{ let _foo = 'secret', bar = () => _foo; }`
  - Doesn't break user code in most situations, and typically what you wanted anyways
- Read [ES6 Let, Const and the “Temporal Dead Zone” (TDZ) in Depth]

[(back to table of contents)]

Symbols

- A new primitive type in ES6
- You can create your own symbols using `var symbol = Symbol()`
- You can add a description for debugging purposes, like `Symbol('ponyfoo')`
- Symbols are immutable and unique. `Symbol()`, `Symbol()`, `Symbol('foo')` and `Symbol('foo')` are all different
- Symbols are of type `symbol`, thus: `typeof Symbol() === 'symbol'`
- You can also create global symbols with `Symbol.for(key)`
  - If a symbol with the provided `key` already existed, you get that one back
  - Otherwise, a new symbol is created, using `key` as its description as well
  - `Symbol.keyFor(symbol)` is the inverse function, taking a `symbol` and returning its `key`
  - Global symbols are **as global as it gets**, or *cross-realm*. Single registry used to look up these symbols across the runtime
    - `window` context
    - `eval` context
    - `<iframe>` context, `Symbol.for('foo') === iframe.contentWindow.Symbol.for('foo')`
- There's also "well-known" symbols
  - Not on the global registry, accessible through `Symbol[name]`, e.g: `Symbol.iterator`
  - Cross-realm, meaning `Symbol.iterator === iframe.contentWindow.Symbol.iterator`

  - Used by specification to define protocols, such as the *iterable* protocol over `Symbol.iterator`
  - They're not **actually well-known** -- in colloquial terms
- Iterating over symbol properties is hard, but not impossible and definitely not private
  - Symbols are hidden to all pre-ES6 "reflection" methods
  - Symbols are accessible through `Object.getOwnPropertySymbols`
  - You won't stumble upon them but you **will** find them if *actively looking*
- Read ES6 Symbols in Depth

(back to table of contents)

Iterators

- Iterator and iterable protocol define how to iterate over any object, not just arrays and array-likes
- A well-known `Symbol` is used to assign an iterator to any object
- `var foo = { [Symbol.iterator]: iterable}`, or `foo[Symbol.iterator] = iterable`
- The `iterable` is a method that returns an `iterator` object that has a `next` method
- The `next` method returns objects with two properties, `value` and `done`
  - The `value` property indicates the current value in the sequence being iterated
  - The `done` property indicates whether there are any more items to iterate
- Objects that have a `[Symbol.iterator]` value are *iterable*, because they subscribe to the iterable protocol
- Some built-ins like `Array`, `String`, or `arguments` -- and `NodeList` in browsers -- are iterable by default in ES6
- Iterable objects can be looped over with `for..of`, such as `for (let el of document.querySelectorAll('a'))`
- Iterable objects can be synthesized using the spread operator, like `[...document.querySelectorAll('a')]`
- You can also use `Array.from(document.querySelectorAll('a'))` to synthesize an iterable sequence into an array
- Iterators are *lazy*, and those that produce an infinite sequence still can lead to valid programs
- Be careful not to attempt to synthesize an infinite sequence with `...` or `Array.from` as that **will** cause an infinite loop
- Read ES6 Iterators in Depth

(back to table of contents)

Generators

- Generator functions are a special kind of *iterator* that can be declared using the `function* generator () {}` syntax
- Generator functions use `yield` to emit an element sequence
- Generator functions can also use `yield*` to delegate to another generator function -- *or any iterable object*
- Generator functions return a generator object that adheres to both the *iterable* and *iterator* protocols
  - Given `g = generator()`, `g` adheres to the iterable protocol because `g[Symbol.iterator]` is a method
  - Given `g = generator()`, `g` adheres to the iterator protocol because `g.next` is a method
  - The iterator for a generator object `g` is the generator itself: `g[Symbol.iterator]() === g`
- Pull values using `Array.from(g)`, `[...g]`, `for (let item of g)`, or just calling `g.next()`
- Generator function execution is suspended, remembering the last position, in four different cases
  - A `yield` expression returning the next value in the sequence
  - A `return` statement returning the last value in the sequence
  - A `throw` statement halts execution in the generator entirely
  - Reaching the end of the generator function signals `{ done: true }`
- Once the `g` sequence has ended, `g.next()` simply returns `{ done: true }` and has no effect
- It's easy to make asynchronous flows feel synchronous
  - Take user-provided generator function
  - User code is suspended while asynchronous operations take place
  - Call `g.next()`, unsuspending execution in user code
- Read ES6 Generators in Depth

(back to table of contents)

Promises

- Follows the `Promises/A+` specification, was widely implemented in the wild before ES6 was standarized *(e.g `bluebird`)*
- Promises behave like a tree. Add branches with `p.then(handler)` and `p.catch(handler)`
- Create new `p` promises with `new Promise((resolve, reject) => { /* resolver */ })`
  - The `resolve(value)` callback will fulfill the promise with the provided `value`
  - The `reject(reason)` callback will reject `p` with a `reason` error
  - You can call those methods asynchronously, blocking deeper branches of the promise tree

- Each call to `p.then` and `p.catch` creates another promise that's blocked on `p` being settled
- Promises start out in *pending* state and are **settled** when they're either *fulfilled* or *rejected*
- Promises can only be settled once, and then they're settled. Settled promises unblock deeper branches
- You can tack as many promises as you want onto as many branches as you need
- Each branch will execute either `.then` handlers or `.catch` handlers, never both
- A `.then` callback can transform the result of the previous branch by returning a value
- A `.then` callback can block on another promise by returning it
- `p.catch(fn).catch(fn)` won't do what you want -- unless what you wanted is to catch errors in the error handler
- `Promise.resolve(value)` creates a promise that's fulfilled with the provided `value`
- `Promise.reject(reason)` creates a promise that's rejected with the provided `reason`
- `Promise.all(...promises)` creates a promise that settles when all `...promises` are fulfilled or 1 of them is rejected
- `Promise.race(...promises)` creates a promise that settles as soon as 1 of `...promises` is settled
- Use Promisees -- the promise visualization playground -- to better understand promises
- Read ES6 Promises in Depth

(back to table of contents)

Maps

- A replacement to the common pattern of creating a hash-map using plain JavaScript objects
  - Avoids security issues with user-provided keys
  - Allows keys to be arbitrary values, you can even use DOM elements or functions as the `key` to an entry
- `Map` adheres to *iterable* protocol
- Create a `map` using `new Map()`
- Initialize a map with an `iterable` like `[[key1, value1], [key2, value2]]` in `new Map(iterable)`
- Use `map.set(key, value)` to add entries
- Use `map.get(key)` to get an entry
- Check for a `key` using `map.has(key)`
- Remove entries with `map.delete(key)`
- Iterate over `map` with `for (let [key, value] of map)`, the spread operator, `Array.from`, etc
- Read ES6 Maps in Depth

(back to table of contents)

WeakMaps

- Similar to `Map`, but not quite the same
- `WeakMap` isn't iterable, so you don't get enumeration methods like `.forEach`, `.clear`, and others you had in `Map`
- `WeakMap` keys must be reference types. You can't use value types like symbols, numbers, or strings as keys
- `WeakMap` entries with a `key` that's the only reference to the referenced variable are subject to garbage collection
- That last point means `WeakMap` is great at keeping around metadata for objects, while those objects are still in use
- You avoid memory leaks, without manual reference counting -- think of `WeakMap` as `IDisposable` in .NET
- Read ES6 WeakMaps in Depth

(back to table of contents)

Sets

- Similar to `Map`, but not quite the same
- `Set` doesn't have keys, there's only values
- `set.set(value)` doesn't look right, so we have `set.add(value)` instead
- Sets can't have duplicate values because the values are also used as keys
- Read ES6 Sets in Depth

(back to table of contents)

WeakSets

README MIT license

- `WeakSet` values must be reference types
- `WeakSet` may be useful for a metadata table indicating whether a reference is actively in use or not
- Read ES6 WeakSets in Depth

(back to table of contents)

Proxies

- Proxies are created with `new Proxy(target, handler)`, where `target` is any object and `handler` is configuration
- The default behavior of a `proxy` acts as a passthrough to the underlying `target` object
- Handlers determine how the underlying `target` object is accessed on top of regular object property access semantics
- You pass off references to `proxy` and retain strict control over how `target` can be interacted with
- Handlers are also known as traps, these terms are used interchangeably
- You can create **revocable** proxies with `Proxy.revocable(target, handler)`
  - That method returns an object with `proxy` and `revoke` properties
  - You could destructure `var {proxy, revoke} = Proxy.revocable(target, handler)` for convenience
  - You can configure the `proxy` all the same as with `new Proxy(target, handler)`
  - After `revoke()` is called, the `proxy` will **throw** on *any operation*, making it convenient when you can't trust consumers
- `get` -- traps `proxy.prop` and `proxy['prop']`
- `set` -- traps `proxy.prop = value` and `proxy['prop'] = value`
- `has` -- traps `in` operator
- `deleteProperty` -- traps `delete` operator
- `defineProperty` -- traps `Object.defineProperty` and declarative alternatives
- `enumerate` -- traps `for..in` loops
- `ownKeys` -- traps `Object.keys` and related methods
- `apply` -- traps *function calls*
- `construct` -- traps usage of the `new` operator
- `getPrototypeOf` -- traps internal calls to `[[GetPrototypeOf]]`
- `setPrototypeOf` -- traps calls to `Object.setPrototypeOf`
- `isExtensible` -- traps calls to `Object.isExtensible`
- `preventExtensions` -- traps calls to `Object.preventExtensions`
- `getOwnPropertyDescriptor` -- traps calls to `Object.getOwnPropertyDescriptor`
- Read ES6 Proxies in Depth
- Read ES6 Proxy Traps in Depth
- Read More ES6 Proxy Traps in Depth

(back to table of contents)

Reflection

- `Reflection` is a new static built-in (think of `Math`) in ES6
- `Reflection` methods have sensible internals, e.g `Reflect.defineProperty` returns a boolean instead of throwing

There's a `Reflection` method for each proxy trap handler, and they represent the default behavior of each trap

Going forward, new reflection methods in the same vein as `Object.keys` will be placed in the `Reflection` namespace

Read ES6 Reflection in Depth

(back to table of contents)

Number

Use `0b` prefix for binary, and `0o` prefix for octal integer literals

`Number.isNaN` and `Number.isFinite` are like their global namesakes, except that they *don't* coerce input to `Number`

`Number.parseInt` and `Number.parseFloat` are exactly the same as their global namesakes

`Number.isInteger` checks if input is a `Number` value that doesn’t have a decimal part

`Number.EPSILON` helps figure out negligible differences between two numbers -- e.g. `0.1 + 0.2` and `0.3`

`Number.MAX_SAFE_INTEGER` is the largest integer that can be safely and precisely represented in JavaScript

`Number.MIN_SAFE_INTEGER` is the smallest integer that can be safely and precisely represented in JavaScript

`Number.isSafeInteger` checks whether an integer is within those bounds, able to be represented safely and precisely

Read ES6 `Number` Improvements in Depth

(back to table of contents)

Math

`Math.sign` -- sign function of a number

`Math.trunc` -- integer part of a number

`Math.cbrt` -- cubic root of value, or $\sqrt[3]{\texttt{value}}$

`Math.expm1` -- `e` to the `value` minus `1`, or e^{value} `- 1`

`Math.log1p` -- natural logarithm of `value + 1`, or $ln(value + 1)$

`Math.log10` -- base 10 logarithm of `value`, or $log_{10}(value)$

`Math.log2` -- base 2 logarithm of `value`, or $log_2(value)$

`Math.sinh` -- hyperbolic sine of a number

`Math.cosh` -- hyperbolic cosine of a number

`Math.tanh` -- hyperbolic tangent of a number

`Math.asinh` -- hyperbolic arc-sine of a number

- `Math.acosh` -- hyperbolic arc-cosine of a number
- `Math.atanh` -- hyperbolic arc-tangent of a number
- `Math.hypot` -- square root of the sum of squares
- `Math.clz32` -- leading zero bits in the 32-bit representation of a number
- `Math.imul` -- *C-like* 32-bit multiplication
- `Math.fround` -- nearest single-precision float representation of a number
- Read ES6 `Math` Additions in Depth

(back to table of contents)

Array

- `Array.from` -- create `Array` instances from arraylike objects like `arguments` or iterables
- `Array.of` -- similar to `new Array(...items)`, but without special cases
- `Array.prototype.copyWithin` -- copies a sequence of array elements into somewhere else in the array
- `Array.prototype.fill` -- fills all elements of an existing array with the provided value
- `Array.prototype.find` -- returns the first item to satisfy a callback
- `Array.prototype.findIndex` -- returns the index of the first item to satisfy a callback
- `Array.prototype.keys` -- returns an iterator that yields a sequence holding the keys for the array
- `Array.prototype.values` -- returns an iterator that yields a sequence holding the values for the array
- `Array.prototype.entries` -- returns an iterator that yields a sequence holding key value pairs for the array
- `Array.prototype[Symbol.iterator]` -- exactly the same as the `Array.prototype.values` method
- Read ES6 `Array` Extensions in Depth

(back to table of contents)

Object

- `Object.assign` -- recursive shallow overwrite for properties from `target, ...objects`
- `Object.is` -- like using the `===` operator, but `true` for `NaN` vs `NaN`, and `false` for `+0` vs `-0`
- `Object.getOwnPropertySymbols` -- returns all own property symbols found on an object
- `Object.setPrototypeOf` -- changes prototype. Equivalent to `Object.prototype.__proto__` setter
- See also Object Literals section
- Read ES6 `Object` Changes in Depth

(back to table of contents)

Strings and Unicode

- String Manipulation
  - `String.prototype.startsWith` -- whether the string starts with `value`
  - `String.prototype.endsWith` -- whether the string ends in `value`
  - `String.prototype.includes` -- whether the string contains `value` anywhere
  - `String.prototype.repeat` -- returns the string repeated `amount` times
  - `String.prototype[Symbol.iterator]` -- lets you iterate over a sequence of unicode code points *(not characters)*
- Unicode
  - `String.prototype.codePointAt` -- base-10 numeric representation of a code point at a given position in string
  - `String.fromCodePoint` -- given `...codepoints`, returns a string made of their unicode representations
  - `String.prototype.normalize` -- returns a normalized version of the string's unicode representation
- Read ES6 Strings and Unicode Additions in Depth

(back to table of contents)

Modules

- Strict Mode is turned on by default in the ES6 module system
- ES6 modules are files that `export` an API
- `export default value` exports a default binding
- `export var foo = 'bar'` exports a named binding
- Named exports are bindings that can be changed at any time from the module that's exporting them
- `export { foo, bar }` exports a list of named exports
- `export { foo as ponyfoo }` aliases the export to be referenced as `ponyfoo` instead
- `export { foo as default }` marks the named export as the default export
- As a best practice, `export default api` at the end of all your modules, where `api` is an object, avoids confusion
- Module loading is implementation-specific, allows interoperation with CommonJS
- `import 'foo'` loads the `foo` module into the current module
- `import foo from 'ponyfoo'` assigns the default export of `ponyfoo` to a local `foo` variable
- `import {foo, bar} from 'baz'` imports named exports `foo` and `bar` from the `baz` module

- `import {foo as bar} from 'baz'` imports named export `foo` but aliased as a `bar` variable
- `import {default} from 'foo'` also imports the default export
- `import {default as bar} from 'foo'` imports the default export aliased as `bar`
- `import foo, {bar, baz} from 'foo'` mixes default `foo` with named exports `bar` and `baz` in one declaration
- `import * as foo from 'foo'` imports the namespace object
  - Contains all named exports in `foo[name]`
  - Contains the default export in `foo.default`, if a default export was declared in the module

Releases

No releases published

Packages

No packages published

Contributors 5