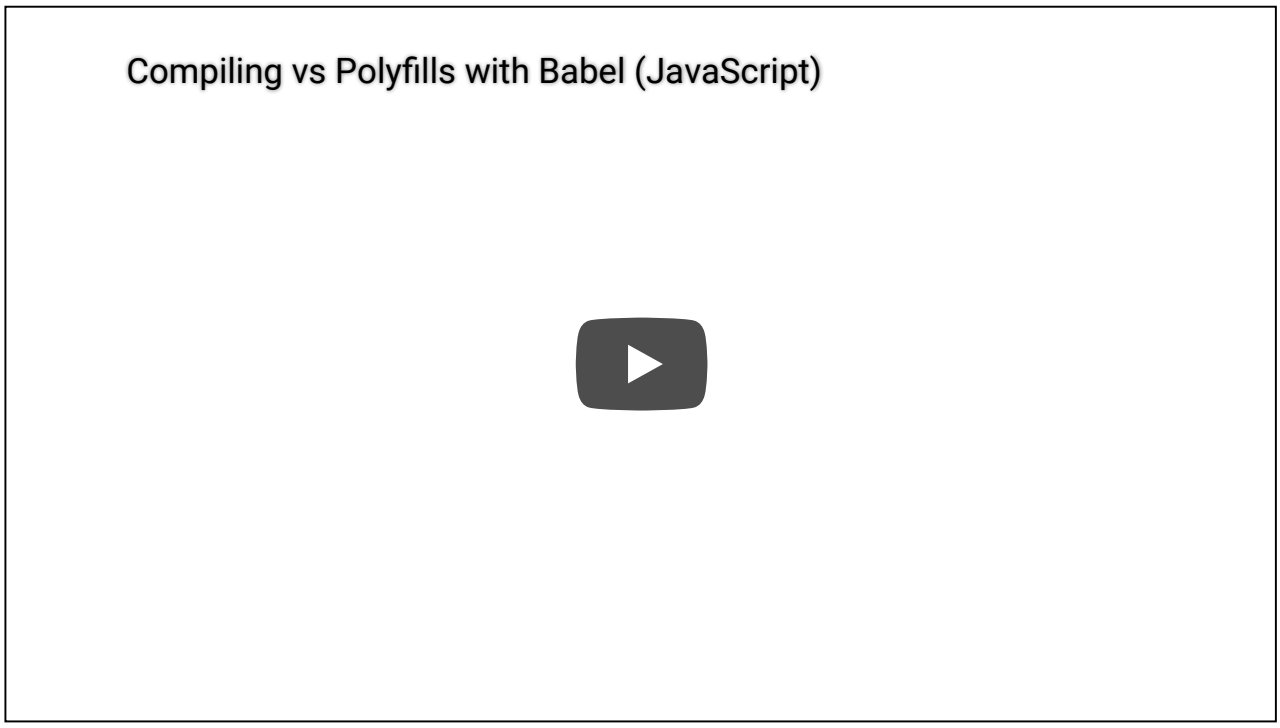
Compiling vs Polyfills with Babel (JavaScript)

October 3 2017. 4 min read.

by Tyler McGinnis 

This is part of our **Modern JavaScript** course. Check it out if you like this post.



JavaScript is a living language that's constantly progressing. As a developer, this is great because we're constantly learning and our tools are constantly improving. The downside of

this is that it typically takes browsers a few years to catch up. Whenever a new language proposal is created, it needs to go through five different stages before it's added to the official language specification. Once it's part of the official spec, it still actually needs to be implemented into every browser you think your users will use. Because of this delay, if you ever want to use the newest JavaScript features, you need to either wait for the latest browsers to implement them (and then hope your users don't use older browsers) or you need to use a tool like Babel to compile your new, modern code back to code that older browsers can understand. When it's framed like that, it's almost certain that a compiler like Babel will always be a fundamental part of building JavaScript applications, assuming the language continues to progress. Again, the purpose of Babel is to take your code which uses new features that browsers may not support yet, and transform it into code that any browser you care about can understand.

So for example, code that looks like this,

```
const getProfile = username => {
  return fetch(`https://api.github.com/users/${username}`)
    .then((response) => response.json())
    .then(({ data }) => ({
      name: data.name,
      location: data.location,
      company: data.company,
      blog: data.blog.includes('https') ? data.blog : null
    }))
    .catch((e) => console.warn(e))
}
```

would get compiled into code that looks like this,

```
var getProfile = function getProfile(username) {
  return fetch('https://api.github.com/users/' + username).then(function (response) {
    return response.json();
  }).then(function (_ref) {
    var data = _ref.data;
    return {
      name: data.name,
      location: data.location,
      company: data.company,
      blog: data.blog.includes('https') ? data.blog : null
    };
  }).catch(function (e) {
    return console.warn(e);
```

```
  });
};
```

You'll notice that most of the ES6 code like the Arrow Functions and Template Strings have been compiled into regular old ES5 JavaScript. There are, however, two features that didn't get compiled: `fetch` and `includes`. The whole goal of Babel is to take our "next generation" code (as the Babel website says) and make it work in all the browsers we care about. You would think that `includes` and `fetch` would get compiled into something native like `indexOf` and `XMLHttpRequest`, but that's not the case. So now the question becomes, why didn't `fetch` or `includes` get compiled? `fetch` isn't actually part of ES6 so that one at least makes a little bit of sense assuming we're only having Babel compile our ES6 code. `includes`, however, is part of ES6 but still didn't get compiled. What this tells us is that compiling only gets our code part of the way there. There's still another step, that if we're using certain new features, we need to take - polyfilling.

What's the difference between compiling and polyfilling? When Babel compiles your code, what it's doing is taking your syntax and running it through various syntax transforms in order to get browser compatible syntax. What it's not doing is adding any new JavaScript primitives or any properties you may need to the browser's global namespace. **One way you can think about it is that when you compile your code, you're transforming it. When you add a polyfill, you're adding new functionality to the browser.**

If this is still fuzzy, here are a list of new language features. Try to figure out if they are compiled or if they need to be polyfilled.

```
Arrow Functions
Classes
Promises
Destructuring
Fetch
String.includes
```

Arrow functions: Babel can transform arrow functions into regular functions, so, they can be compiled.

Classes: Like Arrow functions, `Class` can be transformed into functions with prototypes, so they can be compiled as well.

Promises: There's nothing Babel can do to transform promises into native syntax that browsers understand. More important, compiling won't add new properties, like `Promise`, to the global namespace so Promises need to be polyfilled.

Destructuring: Babel can transform every destructured object into normal variables using dot notation. So, compiled.

Fetch: `fetch` needs to be polyfilled because, by the definition mentioned earlier, when you compile code you're not adding any new global or primitive properties that you may need. `fetch` would be a new property on the global namespace, therefore, it needs to be polyfilled.

String.includes: This one is tricky because it doesn't follow our typical routine. One could argue that `includes` should be transformed to use `indexOf`, however, again, compiling doesn't add new properties to any primitives, so it needs to be polyfilled.

Here's a pretty extensive list from the Babel website as to what features are compiled and what features need to be polyfilled.

Features that need to be compiled

```
Arrow functions
Async functions
Async generator functions
Block scoping
Block scoped functions
Classes
Class properties
Computed property names
Constants
Decorators
Default parameters
Destructuring
Do expressions
Exponentiation operator
For-of
Function bind
Generators
Modules
Module export extensions
New literals
Object rest/spread
Property method assignment
Property name shorthand
Rest parameters
```

```
Spread
Sticky regex
Template literals
Trailing function commas
Type annotations
Unicode regex
```

Features that need to be polyfilled

```
ArrayBuffer
Array.from
Array.of
Array#copyWithin
Array#fill
Array#find
Array#findIndex
Function#name
Map
Math.acosh
Math.hypot
Math.imul
Number.isNaN
Number.isInteger
Object.assign
Object.getOwnPropertyDescriptors
Object.is
Object.entries
Object.values
Object.setPrototypeOf
Promise
Reflect
RegExp#flags
Set
String#codePointAt
String#endsWith
String.fromCodePoint
String#includes
String.raw
String#repeat
String#startsWith
String#padStart
String#padEnd
Symbol
```

```
WeakMap
WeakSet
```

Because adding new features to your project isn't done very often, instead of trying to memorize what's compiled and what's polyfilled, I think it's important to understand the general rules behind the two concepts, then if you need to know if you should include a polyfill for a specific feature, check Babel's website.

Liked this post? Share it 🕺