
Migrating to TypeScript

You’ve heard that TypeScript is great. You also know from painful experience that maintaining your 15-year-old, 100,000-line JavaScript library isn’t. If only it could become a TypeScript library!

This chapter offers some advice about migrating your JavaScript project to TypeScript without losing your sanity and abandoning the effort.

Only the smallest codebases can be migrated in one fell swoop. The key for larger projects is to migrate gradually. **Item 60** discusses how to do this. For a long migration, it’s essential to track your progress and make sure you don’t backslide. This creates a sense of momentum and inevitability to the change. **Item 61** discusses ways to do this.

Migrating a large project to TypeScript won’t necessarily be easy, but it does offer a huge potential upside. A 2017 study found that 15% of bugs fixed in JavaScript projects on GitHub could have been prevented with TypeScript.¹ Even more impressive, a survey of six months’ worth of postmortems at AirBnb found that 38% of them could have been prevented by TypeScript.² If you’re advocating for TypeScript at your organization, stats like these will help! So will running some experiments and finding early adopters. **Item 59** discusses how to experiment with TypeScript before you begin migration.

Since this chapter is largely about JavaScript, many of the code samples are either pure JavaScript (and not expected to pass the type checker) or checked with looser settings (e.g., with `noImplicitAny` off).

¹ Z. Gao, C. Bird, and E. T. Barr, “To Type or Not to Type: Quantifying Detectable Bugs in JavaScript,” ICSE 2017, <http://earlbarr.com/publications/typestudy.pdf>.

² Brie Bunge, “Adopting TypeScript at Scale,” JSConf Hawaii 2019, <https://youtu.be/P-J9Eg7hJwE>.

Item 58: Write Modern JavaScript

In addition to checking your code for type safety, TypeScript compiles your TypeScript code to any version of JavaScript code, all the way back to 1999 vintage ES3. Since TypeScript is a superset of the *latest* version of JavaScript, this means that you can use `tsc` as a “transpiler”: something that takes new JavaScript and converts it to older, more widely supported JavaScript.

Taking a different perspective, this means that when you decide to convert an existing JavaScript codebase to TypeScript, there’s no downside to adopting all the latest JavaScript features. In fact, there’s quite a bit of upside: because TypeScript is designed to work with modern JavaScript, modernizing your JS is a great first step toward adopting TypeScript.

And because TypeScript is a superset of JavaScript, learning to write more modern and idiomatic JavaScript means you’re learning to write better TypeScript, too.

This item gives a quick tour of some of the features in modern JavaScript, which I’m defining here as everything introduced in ES2015 (aka ES6) and after. This material is covered in much greater detail in other books and online. If any of the topics mentioned here are unfamiliar, you owe it to yourself to learn more about them. TypeScript can be tremendously helpful when you’re learning a new language feature like `async/await`: it almost certainly understands the feature better than you do and can guide you toward correct usage.

These are all worth understanding, but by far the most important for adopting TypeScript are ECMAScript Modules and ES2015 classes.

Use ECMAScript Modules

Before the 2015 version of ECMAScript there was no standard way to break your code into separate modules. There were many solutions, from multiple `<script>` tags, manual concatenation, and Makefiles to node.js-style `require` statements or AMD-style `define` callbacks. TypeScript even had its own module system ([Item 53](#)).

Today there is one standard: ECMAScript modules, aka `import` and `export`. If your JavaScript codebase is still a single file, if you use concatenation or one of the other module systems, it’s time to switch to ES modules. This may require setting up a tool like `webpack` or `ts-node`. TypeScript works best with ES modules, and adopting them will facilitate your transition, not least because it will allow you to migrate modules one at a time (see [Item 61](#)).

The details will vary depending on your setup, but if you’re using CommonJS like this:

```
// CommonJS
// a.js
const b = require('./b');
console.log(b.name);

// b.js
const name = 'Module B';
module.exports = {name};
```

then the ES module equivalent would look like:

```
// ECMAScript module
// a.ts
import * as b from './b';
console.log(b.name);

// b.ts
export const name = 'Module B';
```

Use Classes Instead of Prototypes

JavaScript has a flexible prototype-based object model. But by and large JS developers have ignored this in favor of a more rigid class-based model. This was officially enshrined into the language with the introduction of the `class` keyword in ES2015.

If your code uses prototypes in a straightforward way, switch to using classes. That is, instead of:

```
function Person(first, last) {
  this.first = first;
  this.last = last;
}

Person.prototype.getName = function() {
  return this.first + ' ' + this.last;
}

const marie = new Person('Marie', 'Curie');
const personName = marie.getName();
```

write:

```
class Person {
  first: string;
  last: string;

  constructor(first: string, last: string) {
    this.first = first;
    this.last = last;
  }

  getName() {
    return this.first + ' ' + this.last;
  }
}
```

```

    }
  }

  const marie = new Person('Marie', 'Curie');
  const personName = marie.getName();

```

TypeScript struggles with the prototype version of `Person` but understands the class-based version with minimal annotations. If you're unfamiliar with the syntax, TypeScript will help you get it right.

For code that uses older-style classes, the TypeScript language service offers a “Convert function to an ES2015 class” quick fix that can speed this up ([Figure 8-1](#)).

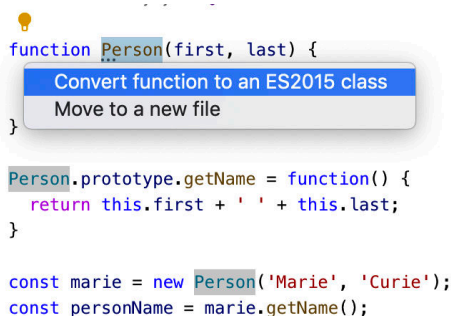


Figure 8-1. The TypeScript language service offers a quick fix to convert older-style classes to ES2015 classes.

Use `let/const` Instead of `var`

JavaScript's `var` has some famously quirky scoping rules. If you're curious to learn more about them, read *Effective JavaScript*. But better to avoid `var` and not worry! Instead, use `let` and `const`. They're truly block-scoped and work in much more intuitive ways than `var`.

Again, TypeScript will help you here. If changing `var` to `let` results in an error, then you're almost certainly doing something you shouldn't be.

Nested function statements also have `var`-like scoping rules:

```

function foo() {
  bar();
  function bar() {
    console.log('hello');
  }
}

```

When you call `foo()`, it logs `hello` because the definition of `bar` is hoisted to the top of `foo`. This is surprising! Prefer function expressions (`const bar = () => { ... }`) instead.

Use for-of or Array Methods Instead of for(;;)

In classic JavaScript you used a C-style for loop to iterate over an array:

```
for (var i = 0; i < array.length; i++) {  
  const el = array[i];  
  // ...  
}
```

In modern JavaScript you can use a for-of loop instead:

```
for (const el of array) {  
  // ...  
}
```

This is less prone to typos and doesn't introduce an index variable. If you want the index variable, you can use `forEach`:

```
array.forEach((el, i) => {  
  // ...  
});
```

Avoid using the `for-in` construct to iterate over arrays as it has many surprises (see [Item 16](#)).

Prefer Arrow Functions Over Function Expressions

The `this` keyword is one of the most famously confusing aspects of JavaScript because it has different scoping rules than other variables:

```
class Foo {  
  method() {  
    console.log(this);  
    [1, 2].forEach(function(i) {  
      console.log(this);  
    });  
  }  
}  
  
const f = new Foo();  
f.method();  
// Prints Foo, undefined, undefined in strict mode  
// Prints Foo, window, window (!) in non-strict mode
```

Generally you want `this` to refer to the relevant instance of whichever class you're in. Arrow functions help you do that by keeping the `this` value from their enclosing scope:

```

class Foo {
  method() {
    console.log(this);
    [1, 2].forEach(i => {
      console.log(this);
    });
  }
}
const f = new Foo();
f.method();
// Always prints Foo, Foo, Foo

```

In addition to having simpler semantics, arrow functions are more concise. You should use them whenever possible. For more on this binding, see [Item 49](#). With the `noImplicitThis` (or `strict`) compiler option, TypeScript will help you get your this-binding right.

Use Compact Object Literals and Destructuring Assignment

Instead of writing:

```

const x = 1, y = 2, z = 3;
const pt = {
  x: x,
  y: y,
  z: z
};

```

you can simply write:

```

const x = 1, y = 2, z = 3;
const pt = { x, y, z };

```

In addition to being more concise, this encourages consistent naming of variables and properties, something your human readers will appreciate as well ([Item 36](#)).

To return an object literal from an arrow function, wrap it in parentheses:

```

['A', 'B', 'C'].map((char, idx) => ({char, idx}));
// [ { char: 'A', idx: 0 }, { char: 'B', idx: 1 }, { char: 'C', idx: 2 } ]

```

There is also shorthand for properties whose values are functions:

```

const obj = {
  onClickLong: function(e) {
    // ...
  },
  onClickCompact(e) {
    // ...
  }
};

```

The inverse of compact object literals is object destructuring. Instead of writing:

```
const props = obj.props;
const a = props.a;
const b = props.b;
```

you can write:

```
const {props} = obj;
const {a, b} = props;
```

or even:

```
const {props: {a, b}} = obj;
```

In this last example only `a` and `b` become variables, not `props`.

You may specify default values when destructuring. Instead of writing:

```
let {a} = obj.props;
if (a === undefined) a = 'default';
```

write this:

```
const {a = 'default'} = obj.props;
```

You can also destructure arrays. This is particularly useful with tuple types:

```
const point = [1, 2, 3];
const [x, y, z] = point;
const [, a, b] = point; // Ignore the first one
```

Destructuring can also be used in function parameters:

```
const points = [
  [1, 2, 3],
  [4, 5, 6],
];
points.forEach(([x, y, z]) => console.log(x + y + z));
// Logs 6, 15
```

As with compact object literal syntax, destructuring is concise and encourages consistent variable naming. Use it!

Use Default Function Parameters

In JavaScript, all function parameters are optional:

```
function log2(a, b) {
  console.log(a, b);
}
log2();
```

This outputs:

```
undefined undefined
```

This is often used to implement default values for parameters:

```
function parseNum(str, base) {
  base = base || 10;
  return parseInt(str, base);
}
```

In modern JavaScript, you can specify the default value directly in the parameter list:

```
function parseNum(str, base=10) {
  return parseInt(str, base);
}
```

In addition to being more concise, this makes it clear that `base` is an optional parameter. Default parameters have another benefit when you migrate to TypeScript: they help the type checker infer the type of the parameter, removing the need for a type annotation. See [Item 19](#).

Use `async/await` Instead of Raw Promises or Callbacks

[Item 25](#) explains why `async` and `await` are preferable, but the gist is that they'll simplify your code, prevent bugs, and help types flow through your asynchronous code.

Instead of either of these:

```
function getJSON(url: string) {
  return fetch(url).then(response => response.json());
}
function getJSONCallback(url: string, cb: (result: unknown) => void) {
  // ...
}
```

write this:

```
async function getJSON(url: string) {
  const response = await fetch(url);
  return response.json();
}
```

Don't Put `use strict` in TypeScript

ES5 introduced “strict mode” to make some suspect patterns more explicit errors. You enable it by putting `'use strict'` in your code:

```
'use strict';
function foo() {
  x = 10; // Throws in strict mode, defines a global in non-strict.
}
```

If you've never used strict mode in your JavaScript codebase, then give it a try. The errors it finds are likely to be ones that the TypeScript compiler will find, too.

But as you transition to TypeScript, there's not much value in keeping `'use strict'` in your source code. By and large, the sanity checks that TypeScript provides are far stricter than those offered by strict mode.

There is some value in having a `'use strict'` in the JavaScript that tsc emits. If you set the `alwaysStrict` or `strict` compiler options, TypeScript will parse your code in strict mode and put a `'use strict'` in the JavaScript output for you.

In short, don't write `'use strict'` in your TypeScript. Use `alwaysStrict` instead.

These are just a few of the many new JavaScript features that TypeScript lets you use. TC39, the body that governs JS standards, is very active, and new features are added year to year. The TypeScript team is currently committed to implementing any feature that reaches stage 3 (out of 4) in the standardization process, so you don't even have to wait for the ink to dry. Check out the TC39 GitHub repo³ for the latest. As of this writing, the pipeline and decorators proposals in particular have great potential to impact TypeScript.

Things to Remember

- TypeScript lets you write modern JavaScript whatever your runtime environment. Take advantage of this by using the language features it enables. In addition to improving your codebase, this will help TypeScript understand your code.
- Use TypeScript to learn language features like classes, destructuring, and `async/await`.
- Don't bother with `'use strict'`: TypeScript is stricter.
- Check the TC39 GitHub repo and TypeScript release notes to learn about all the latest language features.

Item 59: Use `@ts-check` and JSDoc to Experiment with TypeScript

Before you begin the process of converting your source files from JavaScript to TypeScript ([Item 60](#)), you may want to experiment with type checking to get an initial read on the sorts of issues that will come up. TypeScript's `@ts-check` directive lets you do exactly this. It directs the type checker to analyze a single file and report whatever issues it finds. You can think of it as an extremely loose version of type checking: looser even than TypeScript with `noImplicitAny` off ([Item 2](#)).

³ <https://github.com/tc39/proposals>

Here's how it works:

```
// @ts-check
const person = {first: 'Grace', last: 'Hopper'};
2 * person.first
// ~~~~~~ The right-hand side of an arithmetic operation must be of type
//         'any', 'number', 'bigint', or an enum type
```

TypeScript infers the type of `person.first` as `string`, so `2 * person.first` is a type error, no type annotations required.

While it may surface this sort of blatant type error, or functions called with too many arguments, in practice, `// @ts-check` tends to turn up a few specific types of errors:

Undeclared Globals

If these are symbols that you're defining, then declare them with `let` or `const`. If they are “ambient” symbols that are defined elsewhere (in a `<script>` tag in an HTML file, for instance), then you can create a type declarations file to describe them.

For example, if you have JavaScript like this:

```
// @ts-check
console.log(user.firstName);
// ~~~~ Cannot find name 'user'
```

then you could create a file called `types.d.ts`:

```
interface UserData {
  firstName: string;
  lastName: string;
}
declare let user: UserData;
```

Creating this file on its own may fix the issue. If it does not, you may need to explicitly import it with a “triple-slash” reference:

```
// @ts-check
/// <reference path="./types.d.ts" />
console.log(user.firstName); // OK
```

This `types.d.ts` file is a valuable artifact that will become the basis for your project's type declarations.

Unknown Libraries

If you're using a third-party library, TypeScript needs to know about it. For example, you might use jQuery to set the size of an HTML element. With `@ts-check`, TypeScript will flag an error:

```
// @ts-check
$('#graph').style({'width': '100px', 'height': '100px'});
// ~ Cannot find name '$'
```

The solution is to install the type declarations for jQuery:

```
$ npm install --save-dev @types/jquery
```

Now the error is specific to jQuery:

```
// @ts-check
$('#graph').style({'width': '100px', 'height': '100px'});
// ~~~~~ Property 'style' does not exist on type 'jQuery<HTMLElement>'
```

In fact, it should be `.css`, not `.style`.

@ts-check lets you take advantage of the TypeScript declarations for popular JavaScript libraries without migrating to TypeScript yourself. This is one of the best reasons to use it.

DOM Issues

Assuming you're writing code that runs in a web browser, TypeScript is likely to flag issues around your handling of DOM elements. For example:

```
// @ts-check
const ageEl = document.getElementById('age');
ageEl.value = '12';
// ~~~~~ Property 'value' does not exist on type 'HTMLElement'
```

The issue is that only `HTMLInputElement`s have a `value` property, but `document.getElementById` returns the more generic `HTMLElement` (see [Item 55](#)). If you know that the `#age` element really is an input element, then this is an appropriate time to use a type assertion ([Item 9](#)). But this is still a JS file, so you can't write as `HTMLInputElement`. Instead, you can assert a type using JSDoc:

```
// @ts-check
const ageEl = /** @type {HTMLInputElement} */(document.getElementById('age'));
ageEl.value = '12'; // OK
```

If you mouse over `ageEl` in your editor, you'll see that TypeScript now considers it an `HTMLInputElement`. Take care as you type the JSDoc `@type` annotation: the parentheses after the comment are required.

This leads to another type of error that comes up with @ts-check, inaccurate JSDoc, as explained next.

Inaccurate JSDoc

If your project already has JSDoc-style comments, TypeScript will begin checking them when you flip on @ts-check. If you previously used a system like the Closure

Compiler that used these comments to enforce type safety, then this shouldn't cause major headaches. But you may be in for some surprises if your comments were more like "aspirational JSDoc":

```
// @ts-check
/**
 * Gets the size (in pixels) of an element.
 * @param {Node} el The element
 * @return {{w: number, h: number}} The size
 */
function getSize(el) {
  const bounds = el.getBoundingBoxRect();
  // ~~~~~ Property 'getBoundingBoxRect'
  // ~~~~~ does not exist on type 'Node'
  return {width: bounds.width, height: bounds.height};
  // ~~~~~ Type '{ width: any; height: any; }' is not
  // ~~~~~ assignable to type '{ w: number; h: number; }'
}
```

The first issue is a misunderstanding of the DOM: `getBoundingBoxRect()` is defined on `Element`, not `Node`. So the `@param` tag should be updated. The second is a mismatch between properties specified in the `@return` tag and the implementation. Presumably the rest of the project uses the `width` and `height` properties, so the `@return` tag should be updated.

You can use JSDoc to gradually add type annotations to your project. The TypeScript language service will offer to infer type annotations as a quick fix for code where it's clear from usage, as shown here and in [Figure 8-2](#):

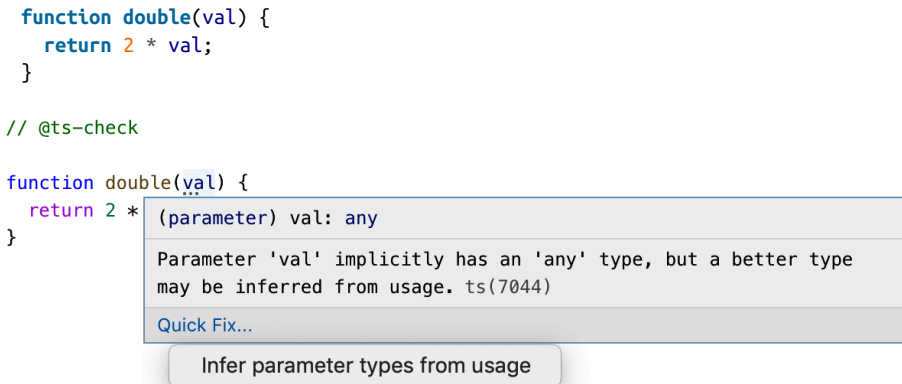


Figure 8-2. The TypeScript Language Services offer a quick fix to infer parameter types from usage.

This results in a correct JSDoc annotation:

```
// @ts-check
/**
 * @param {number} val
 */
function double(val) {
  return 2 * val;
}
```

This can be helpful to encourage types to flow through your code with `@ts-check`. But it doesn't always work so well. For instance:

```
function loadData(data) {
  data.files.forEach(async file => {
    // ...
  });
}
```

If you use the quick fix to annotate `data`, you'll wind up with:

```
/**
 * @param {{
 *   files: { forEach: (arg0: (file: any) => Promise<void>) => void; };
 * }} data
 */
function loadData(data) {
  // ...
}
```

This is structural typing gone awry (Item 4). While the function would technically work on any sort of object with a `forEach` method with that signature, the intent was most likely for the parameter to be `{files: string[]}`.

You can get much of the TypeScript experience in a JavaScript project using JSDoc annotations and `@ts-check`. This is appealing because it requires no changes in your tooling. But it's best not to go too far in this direction. Comment boilerplate has real costs: it's easy for your logic to get lost in a sea of JSDoc. TypeScript works best with `.ts` files, not `.js` files. The goal is ultimately to convert your project to TypeScript, not to JavaScript with JSDoc annotations. But `@ts-check` can be a useful way to experiment with types and find some initial errors, especially for projects that already have extensive JSDoc annotations.

Things to Remember

- Add `///
@ts-check` to the top of a JavaScript file to enable type checking.
- Recognize common errors. Know how to declare globals and add type declarations for third-party libraries.
- Use JSDoc annotations for type assertions and better type inference.

- Don't spend too much time getting your code perfectly typed with JSDoc. Remember that the goal is to convert to *.ts*!

Item 60: Use `allowJs` to Mix TypeScript and JavaScript

For a small project, you may be able to convert from JavaScript to TypeScript in one fell swoop. But for a larger project this “stop the world” approach won't work. You need to be able to transition gradually. That means you need a way for TypeScript and JavaScript to coexist.

The key to this is the `allowJs` compiler option. With `allowJs`, TypeScript files and JavaScript files may import one another. For JavaScript files this mode is extremely permissive. Unless you use `@ts-check` (Item 59), the only errors you'll see are syntax errors. This is “TypeScript is a superset of JavaScript” in the most trivial sense.

While it's unlikely to catch errors, `allowJs` does give you an opportunity to introduce TypeScript into your build chain before you start making code changes. This is a good idea because you'll want to be able to run your tests as you convert modules to TypeScript (Item 61).

If your bundler includes TypeScript integration or has a plug-in available, that's usually the easiest path forward. With `browserify`, for instance, you run `npm install --save-dev tsify` and add it as a plug-in:

```
$ browserify index.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Most unit testing tools have an option like this as well. With the `jest` tool, for instance, you install `ts-jest` and pass TypeScript sources through it by specifying a `jest.config.js` like:

```
module.exports = {
  transform: {
    '^.+\\.tsx?$': 'ts-jest',
  },
};
```

If your build chain is custom, your task will be more involved. But there's always a good fallback option: when you specify the `outDir` option, TypeScript will generate pure JavaScript sources in a directory that parallels your source tree. Usually your existing build chain can be run over that. You may need to tweak TypeScript's JavaScript output so that it closely matches your original JavaScript source, (e.g., by specifying the `target` and `module` options).

Adding TypeScript into your build and test process may not be the most enjoyable task, but it is an essential one that will let you begin to migrate your code with confidence.

Things to Remember

- Use the `allowJs` compiler option to support mixed JavaScript and TypeScript as you transition your project.
- Get your tests and build chain working with TypeScript before beginning large-scale migration.

Item 61: Convert Module by Module Up Your Dependency Graph

You've adopted modern JavaScript, converting your project to use ECMAScript modules and classes ([Item 58](#)). You've integrated TypeScript into your build chain and have all your tests passing ([Item 60](#)). Now for the fun part: converting your JavaScript to TypeScript. But where to begin?

When you add types to a module, it's likely to surface new type errors in all the modules that depend on it. Ideally you'd like to convert each module once and be done with it. This implies that you should convert modules going *up* the dependency graph: starting with the leaves (modules that depend on no others) and moving up to the root.

The very first modules to migrate are your third-party dependencies since, by definition, you depend on them but they do not depend on you. Usually this means installing `@types` modules. If you use the `lodash` utility library, for example, you'd run `npm install --save-dev @types/lodash`. These typings will help types flow through your code and surface issues in your use of the libraries.

If your code calls external APIs, you may also want to add type declarations for these early on. Although these calls may happen anywhere in your code, this is still in the spirit of moving up the dependency graph since you depend on the APIs but they do not depend on you. Many types flow from API calls, and these are generally difficult to infer from context. If you can find a spec for the API, generate types from that (see [Item 35](#)).

As you migrate your own modules, it's helpful to visualize the dependency graph. [Figure 8-3](#) shows an example graph from a medium-sized JavaScript project, made using the excellent `madge` tool.

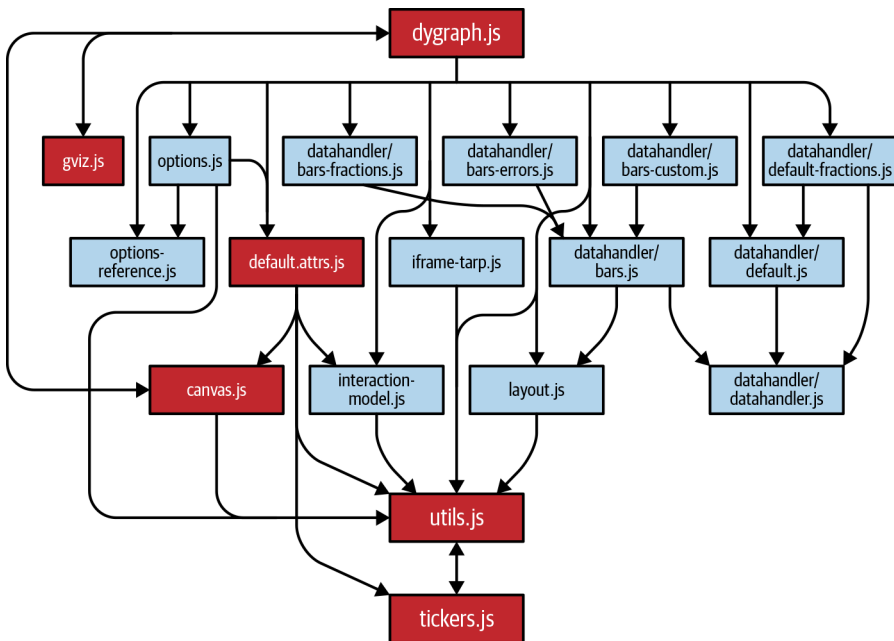


Figure 8-3. The dependency graph for a medium-sized JavaScript project. Arrows indicate dependencies. Darker-shaded boxes indicate that a module is involved in a circular dependency.

The bottom of this dependency graph is the circular dependency between *utils.js* and *tickers.js*. There are many modules that depend on these two, but they only depend on one another. This pattern is quite common: most projects will have some sort of utility module at the bottom of the dependency graph.

As you migrate your code, focus on adding types rather than refactoring. If this is an old project, you're likely to notice some strange things and want to fix them. Resist this urge! The immediate goal is to convert your project to TypeScript, not to improve its design. Instead, write down code smells as you detect them and make a list of future refactors.

There are a few common errors you'll run into as you convert to TypeScript. Some of these were covered in [Item 59](#), but new ones include:

Undeclared Class Members

Classes in JavaScript do not need to declare their members, but classes in TypeScript do. When you rename a class's *.js* file to *.ts*, it's likely to show errors for every single property you reference:


```

class Greeting {
  constructor(name) {
    this.greeting = 'Hello';
    // ~~~~~ Property 'greeting' does not exist on type 'Greeting'
    this.name = name;
    // ~~~~ Property 'name' does not exist on type 'Greeting'
  }
  greet() {
    return this.greeting + ' ' + this.name;
    // ~~~~~ ~~~~ Property ... does not exist
  }
}

```

There's a helpful quick fix (see [Figure 8-4](#)) for this that you should take advantage of.



Figure 8-4. The quick fix to add declarations for missing members is particularly helpful in converting a class to TypeScript.

This will add declarations for the missing members based on usage:

```

class Greeting {
  greeting: string;
  name: any;
  constructor(name) {
    this.greeting = 'Hello';
    this.name = name;
  }
  greet() {
    return this.greeting + ' ' + this.name;
  }
}

```

TypeScript was able to get the type for greeting correct, but not the type for name. After applying this quick fix, you should look through the property list and fix the any types.

If this is the first time you've seen the full property list for your class, you may be in for a shock. When I converted the main class in *dygraph.js* (the root module in

Figure 8-3), I discovered that it had no fewer than 45 member variables! Migrating to TypeScript has a way of surfacing bad designs like this that were previously implicit. It's harder to justify a bad design if you have to look at it. But again, resist the urge to refactor now. Note the oddity and think about how you'd fix it some other day.

Values with Changing Types

TypeScript will complain about code like this:

```
const state = {};  
state.name = 'New York';  
    // ~~~~ Property 'name' does not exist on type '{}'  
state.capital = 'Albany';  
    // ~~~~~~ Property 'capital' does not exist on type '{}'
```

This topic is covered in more depth in [Item 23](#), so you may want to brush up on that item if you run into this error. If the fix is trivial, you can build the object all at once:

```
const state = {  
  name: 'New York',  
  capital: 'Albany',  
}; // OK
```

If it is not, then this is an appropriate time to use a type assertion:

```
interface State {  
  name: string;  
  capital: string;  
}  
const state = {} as State;  
state.name = 'New York'; // OK  
state.capital = 'Albany'; // OK
```

You should fix this eventually (see [Item 9](#)), but this is expedient and will help you keep the migration going.

If you've been using JSDoc and `@ts-check` ([Item 59](#)), be aware that you can actually *lose* type safety by converting to TypeScript. For instance, TypeScript flags an error in this JavaScript:

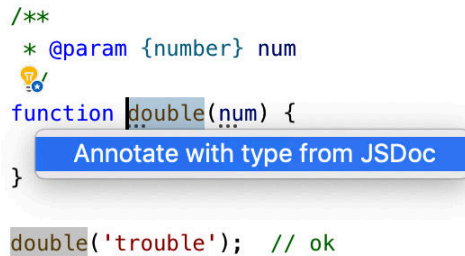
```
// @ts-check  
/**  
 * @param {number} num  
 */  
function double(num) {  
  return 2 * num;  
}  
  
double('trouble');  
    // ~~~~~~ Argument of type '"trouble"' is not assignable to  
    //         parameter of type 'number'
```

When you convert to TypeScript, the @ts-check and JSDoc stop being enforced. This means the type of num is implicitly any, so there's no error:

```
/**
 * @param {number} num
 */
function double(num) {
  return 2 * num;
}

double('trouble'); // OK
```

Fortunately there's a quick fix available to move JSDoc types to TypeScript types. If you have any JSDoc, you should use what's shown in [Figure 8-5](#).



The screenshot shows a code editor with the following code:

```
/**
 * @param {number} num
 */
function double(num) {
}

double('trouble'); // ok
```

A tooltip is visible over the `double` function signature, displaying the text "Annotate with type from JSDoc".

Figure 8-5. Quick fix to copy JSDoc annotations to TypeScript type annotations

Once you've copied type annotations to TypeScript, make sure to remove them from the JSDoc to avoid redundancy (see [Item 30](#)):

```
function double(num: number) {
  return 2 * num;
}

double('trouble');
// ~~~~~ Argument of type '"trouble"' is not assignable to
//       parameter of type 'number'
```

This issue will also be caught when you turn on `noImplicitAny`, but you may as well add the types now.

Migrate your tests last. They should be at the top of your dependency graph (since your code doesn't depend on them), and it's extremely helpful to know that your tests continue to pass during the migration despite your not having changed them at all.

Things to Remember

- Start migration by adding `@types` for third-party modules and external API calls.

- Begin migrating your modules from the bottom of the dependency graph upwards. The first module will usually be some sort of utility code. Consider visualizing the dependency graph to help you track progress.
- Resist the urge to refactor your code as you uncover odd designs. Keep a list of ideas for future refactors, but stay focused on TypeScript conversion.
- Be aware of common errors that come up during conversion. Copy JSDoc annotations if necessary to avoid losing type safety as you convert.

Item 62: Don't Consider Migration Complete Until You Enable `noImplicitAny`

Converting your whole project to `.ts` is a big accomplishment. But your work isn't done quite yet. Your next goal is to turn on the `noImplicitAny` option ([Item 2](#)). TypeScript code without `noImplicitAny` is best thought of as transitional because it can mask real errors you've made in your type declarations.

For example, perhaps you've used the "Add all missing members" quick fix to add property declarations to a class ([Item 61](#)). You're left with an `any` type and would like to fix it:

```
class Chart {
  indices: any;

  // ...
}
```

`indices` sounds like it should be an array of numbers, so you plug in that type:

```
class Chart {
  indices: number[];

  // ...
}
```

No new errors result, so you then keep moving. Unfortunately, you've made a mistake: `number[]` is the wrong type. Here's some code from elsewhere in the class:

```
getRanges() {
  for (const r of this.indices) {
    const low = r[0]; // Type is any
    const high = r[1]; // Type is any
    // ...
  }
}
```

Clearly `number[][]` or `[number, number][]` would be a more accurate type. Does it surprise you that indexing into a number is allowed? Take this as an indication of just how loose TypeScript can be without `noImplicitAny`.

When you turn on `noImplicitAny`, this becomes an error:

```
getRanges() {  
  for (const r of this.indices) {  
    const low = r[0];  
    // ~~~~ Element implicitly has an 'any' type because  
    //       type 'Number' has no index signature  
    const high = r[1];  
    // ~~~~ Element implicitly has an 'any' type because  
    //       type 'Number' has no index signature  
    // ...  
  }  
}
```

A good strategy for enabling `noImplicitAny` is to set it in your local client and start fixing errors. The number of errors you get from the type checker gives you a good sense of your progress. You can commit the type corrections without committing the `tsconfig.json` change until you get the number of errors down to zero.

There are many other knobs you can turn to increase the strictness of type checking, culminating with `"strict": true`. But `noImplicitAny` is the most important one and your project will get most of the benefits of TypeScript even if you don't adopt other settings like `strictNullChecks`. Give everyone on your team a chance to get used to TypeScript before you adopt stricter settings.

Things to Remember

- Don't consider your TypeScript migration done until you adopt `noImplicitAny`. Loose type checking can mask real mistakes in type declarations.
- Fix type errors gradually before enforcing `noImplicitAny`. Give your team a chance to get comfortable with TypeScript before adopting stricter checks.