CHAPTER 7

Writing and Running Your Code

This chapter is a bit of a grab bag: it covers some issues that come up in writing code (not types) as well as issues you may run into when you run your code.

Item 53: Prefer ECMAScript Features to TypeScript Features

The relationship between TypeScript and JavaScript has changed over time. When Microsoft first started work on TypeScript in 2010, the prevailing attitude around JavaScript was that it was a problematic language that needed to be fixed. It was common for frameworks and source-to-source compilers to add missing features like classes, decorators, and a module system to JavaScript. TypeScript was no different. Early versions included home-grown versions of classes, enums, and modules.

Over time TC39, the standards body that governs JavaScript, added many of these same features to the core JavaScript language. And the features they added were not compatible with the versions that existed in TypeScript. This left the TypeScript team in an awkward predicament: adopt the new features from the standard or break existing code?

TypeScript has largely chosen to do the latter and eventually articulated its current governing principle: TC39 defines the runtime while TypeScript innovates solely in the type space.

There are a few remaining features from before this decision. It's important to recognize and understand these, because they don't fit the pattern of the rest of the language. In general, I recommend avoiding them to keep the relationship between TypeScript and JavaScript as clear as possible.

Enums

Many languages model types that can take on a small set of values using *enumerations* or *enums*. TypeScript adds them to JavaScript:

```
enum Flavor {
  VANILLA = 0,
  CHOCOLATE = 1,
  STRAWBERRY = 2,
}

let flavor = Flavor.CHOCOLATE;  // Type is Flavor

Flavor  // Autocomplete shows: VANILLA, CHOCOLATE, STRAWBERRY
Flavor[0]  // Value is "VANILLA"
```

The argument for enums is that they provide more safety and transparency than bare numbers. But enums in TypeScript have some quirks. There are actually several variants on enums that all have subtly different behaviors:

- A number-valued enum (like `Flavor`). Any number is assignable to this, so it's not very safe. (It was designed this way to make bit flag structures possible.)
- A string-valued enum. This does offer type safety, and also more transparent values at runtime. But it's not structurally typed, unlike every other type in TypeScript (more on this momentarily).
- `const enum`. Unlike regular enums, const enums go away completely at runtime. If you changed to `const enum Flavor` in the previous example, the compiler would rewrite `Flavor.CHOCOLATE` as `0`. This also breaks our expectations around how the compiler behaves and still has the divergent behaviors between `string` and `number`-valued enums.
- `const enum` with the `preserveConstEnums` flag set. This emits runtime code for `const enum`s, just like for a regular `enum`.

That string-valued enums are nominally typed comes as a particular surprise, since every other type in TypeScript uses structural typing for assignability (see Item 4):

```
enum Flavor {
  VANILLA = 'vanilla',
  CHOCOLATE = 'chocolate',
  STRAWBERRY = 'strawberry',
}

let flavor = Flavor.CHOCOLATE;  // Type is Flavor
    flavor = 'strawberry';
 // ~~~~~~ Type '"strawberry"' is not assignable to type 'Flavor'
```

This has implications when you publish a library. Suppose you have a function that takes a `Flavor`:

```
function scoop(flavor: Flavor) { /* ... */ }
```

Because a `Flavor` at runtime is really just a string, it's fine for your JavaScript users to call it with one:

```
scoop('vanilla');  // OK in JavaScript
```

but your TypeScript users will need to import the `enum` and use that instead:

```
scoop('vanilla');
   // ~~~~~~~~~ '"vanilla"' is not assignable to parameter of type 'Flavor'

import {Flavor} from 'ice-cream';
scoop(Flavor.VANILLA);  // OK
```

These divergent experiences for JavaScript and TypeScript users are a reason to avoid string-valued enums.

TypeScript offers an alternative to enums that is less common in other languages: a union of literal types.

```
type Flavor = 'vanilla' | 'chocolate' | 'strawberry';

let flavor: Flavor = 'chocolate';  // OK
    flavor = 'mint chip';
 // ~~~~~~ Type '"mint chip"' is not assignable to type 'Flavor'
```

This offers as much safety as the enum and has the advantage of translating more directly to JavaScript. It also offers similarly strong autocomplete in your editor:

```
function scoop(flavor: Flavor) {
  if (flavor === 'v
                  // Autocomplete here suggests 'vanilla'
}
```

For more on this approach, see Item 33.

Parameter Properties

It's common to assign properties to a constructor parameter when initializing a class:

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}
```

TypeScript provides a more compact syntax for this:

```
class Person {
  constructor(public name: string) {}
}
```

This is called a "parameter property," and it is equivalent to the code in the first example. There are a few issues to be aware of with parameter properties:

- They are one of the few constructs which generates code when you compile to JavaScript (`enum`s are another). Generally compilation just involves erasing types.
- Because the parameter is only used in generated code, the source looks like it has unused parameters.
- A mix of parameter and non-parameter properties can hide the design of your classes.

For example:

```
class Person {
  first: string;
  last: string;
  constructor(public name: string) {
    [this.first, this.last] = name.split(' ');
  }
}
```

This class has three properties (`first`, `last`, `name`), but this is hard to read off the code because only two are listed before the constructor. This gets worse if the constructor takes other parameters, too.

If your class consists *only* of parameter properties and no methods, you might consider making it an `interface` and using object literals. Remember that the two are assignable to one another because of structural typing Item 4:

```
class Person {
  constructor(public name: string) {}
}
const p: Person = {name: 'Jed Bartlet'};  // OK
```

Opinions are divided on parameter properties. While I generally avoid them, others appreciate the saved keystrokes. Be aware that they do not fit the pattern of the rest of TypeScript, and may in fact obscure that pattern for new developers. Try to avoid hiding the design of your class by using a mix of parameter and non-parameter properties.

Namespaces and Triple-Slash Imports

Before ECMAScript 2015, JavaScript didn't have an official module system. Different environments added this missing feature in different ways: Node.js used `require` and `module.exports` whereas AMD used a `define` function with a callback.

TypeScript also filled this gap with its own module system. This was done using a `module` keyword and "triple-slash" imports. After ECMAScript 2015 added an official module system, TypeScript added `namespace` as a synonym for `module`, to avoid confusion:

```
namespace foo {
  function bar() {}
}

/// <reference path="other.ts"/>
foo.bar();
```

Outside of type declarations, triple-slash imports and the `module` keyword are just a historical curiosity. In your own code, you should use ECMAScript 2015–style modules (`import` and `export`). See Item 58.

Decorators

Decorators can be used to annotate or modify classes, methods, and properties. For example, you could define a `logged` annotation that logs all calls to a method on a class:

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  @logged
  greet() {
    return "Hello, " + this.greeting;
  }
}

function logged(target: any, name: string, descriptor: PropertyDescriptor) {
  const fn = target[name];
  descriptor.value = function() {
    console.log(`Calling ${name}`);
    return fn.apply(this, arguments);
  };
}

console.log(new Greeter('Dave').greet());
// Logs:
// Calling greet
// Hello, Dave
```

This feature was initially added to support the Angular framework and requires the `experimentalDecorators` property to be set in `tsconfig.json`. Their implementation has not yet been standardized by TC39 at the time of this writing, so any code you write today using decorators is liable to break or become non-standard in the

future. Unless you're using Angular or another framework that requires annotations and until they're standardized, don't use TypeScript's decorators.

Things to Remember

- By and large, you can convert TypeScript to JavaScript by removing all the types from your code.
- Enums, parameter properties, triple-slash imports, and decorators are historical exceptions to this rule.
- In order to keep TypeScript's role in your codebase as clear as possible, I recommend avoiding these features.

Item 54: Know How to Iterate Over Objects

This code runs fine, and yet TypeScript flags an error in it. Why?

```
const obj = {
  one: 'uno',
  two: 'dos',
  three: 'tres',
};
for (const k in obj) {
  const v = obj[k];
        // ~~~~~~ Element implicitly has an 'any' type
        //        because type ... has no index signature
}
```

Inspecting the `obj` and `k` symbols gives a clue:

```
const obj = { /* ... */ };
// const obj: {
//     one: string;
//     two: string;
//     three: string;
// }
for (const k in obj) {  // const k: string
  // ...
}
```

The type of `k` is `string`, but you're trying to index into an object whose type only has three specific keys: `'one'`, `'two'`, and `'three'`. There are strings other than these three, so this has to fail.

Plugging in a narrower type declaration for `k` fixes the issue:

```
let k: keyof typeof obj;  // Type is "one" | "two" | "three"
for (k in obj) {
```

```
  const v = obj[k];  // OK
}
```

So the real question is: why is the type of `k` in the first example inferred as `string` rather than `"one" | "two" | "three"`?

To understand, let's look at a slightly different example involving an interface and a function:

```
interface ABC {
  a: string;
  b: string;
  c: number;
}

function foo(abc: ABC) {
  for (const k in abc) {  // const k: string
    const v = abc[k];
          // ~~~~~~ Element implicitly has an 'any' type
          //        because type 'ABC' has no index signature
  }
}
```

It's the same error as before. And you can "fix" it using the same sort of declaration (`let k: keyof ABC`). But in this case TypeScript is right to complain. Here's why:

```
const x = {a: 'a', b: 'b', c: 2, d: new Date()};
foo(x);  // OK
```

The function `foo` can be called with any value *assignable* to `ABC`, not just a value with "a," "b," and "c" properties. It's entirely possible that the value will have other properties, too (see Item 4). To allow for this, TypeScript gives `k` the only type it can be confident of, namely, `string`.

Using the `keyof` declaration would have another downside here:

```
function foo(abc: ABC) {
  let k: keyof ABC;
  for (k in abc) {  // let k: "a" | "b" | "c"
    const v = abc[k];  // Type is string | number
  }
}
```

If `"a" | "b" | "c"` is too narrow for `k`, then `string | number` is certainly too narrow for `v`. In the preceding example one of the values is a `Date`, but it could be anything. The types here give a false sense of certainty that could lead to chaos at runtime.

So what if you just want to iterate over the object's keys and values without type errors? `Object.entries` lets you iterate over both simultaneously:

```
function foo(abc: ABC) {
  for (const [k, v] of Object.entries(abc)) {
    k  // Type is string
    v  // Type is any
  }
}
```

While these types may be hard to work with, they are at least honest!

You should also be aware of the possibility of *prototype pollution*. Even in the case of an object literal that you define, for-in can produce additional keys:

```
> Object.prototype.z = 3; // Please don't do this!
> const obj = {x: 1, y: 2};
> for (const k in obj) { console.log(k); }
x
y
z
```

Hopefully this doesn't happen in a nonadversarial environment (you should never add enumerable properties to `Object.prototype`), but it is another reason that for-in produces `string` keys even for object literals.

If you want to iterate over the keys and values in an object, use either a `keyof` declaration (`let k: keyof T`) or `Object.entries`. The former is appropriate for constants or other situations where you know that the object won't have additional keys and you want precise types. The latter is more generally appropriate, though the key and value types are more difficult to work with.

Things to Remember

- Use `let k: keyof T` and a for-in loop to iterate objects when you know exactly what the keys will be. Be aware that any objects your function receives as parameters might have additional keys.
- Use `Object.entries` to iterate over the keys and values of any object.

Item 55: Understand the DOM hierarchy

Most of the items in this book are agnostic about where you run your TypeScript: in a web browser, on a server, on a phone. This one is different. If you're not working in a browser, skip ahead!

The DOM hierarchy is always present when you're running JavaScript in a web browser. When you use `document.getElementById` to get an element or `document.createElement` to create one, it's always a particular kind of element, even if

you're not entirely familiar with the taxonomy. You call the methods and use the properties that you want and hope for the best.

With TypeScript, the hierarchy of DOM elements becomes more visible. Knowing your `Nodes` from your `Elements` and `EventTargets` will help you debug type errors and decide when type assertions are appropriate. Because so many APIs are based on the DOM, this is relevant even if you're using a framework like React or d3.

Suppose you want to track a user's mouse as they drag it across a `<div>`. You write some seemingly innocuous JavaScript:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
  const dragStart = [eDown.clientX, eDown.clientY];
  const handleUp = (eUp: Event) => {
    targetEl.classList.remove('dragging');
    targetEl.removeEventListener('mouseup', handleUp);
    const dragEnd = [eUp.clientX, eUp.clientY];
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
}
const div = document.getElementById('surface');
div.addEventListener('mousedown', handleDrag);
```

TypeScript's type checker flags no fewer than 11 errors in these 14 lines of code:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
// ~~~~~~~           Object is possibly 'null'.
//         ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
  const dragStart = [
     eDown.clientX, eDown.clientY];
        // ~~~~~~~               Property 'clientX' does not exist on 'Event'
        //               ~~~~~~~ Property 'clientY' does not exist on 'Event'
  const handleUp = (eUp: Event) => {
    targetEl.classList.remove('dragging');
//  ~~~~~~~~          Object is possibly 'null'.
//           ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
    targetEl.removeEventListener('mouseup', handleUp);
//  ~~~~~~~~ Object is possibly 'null'
    const dragEnd = [
       eUp.clientX, eUp.clientY];
        // ~~~~~~~               Property 'clientX' does not exist on 'Event'
        //             ~~~~~~~   Property 'clientY' does not exist on 'Event'
    console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
  }
  targetEl.addEventListener('mouseup', handleUp);
// ~~~~~~~ Object is possibly 'null'
}
```

```
  const div = document.getElementById('surface');
  div.addEventListener('mousedown', handleDrag);
// ~~~ Object is possibly 'null'
```

What went wrong? What's this `EventTarget`? And why might everything be `null`?

To understand the `EventTarget` errors it helps to dig into the DOM hierarchy a bit. Here's some HTML:

```
<p id="quote">and <i>yet</i> it moves</p>
```

If you open your browser's JavaScript console and get a reference to the `p` element, you'll see that it's an `HTMLParagraphElement`:

```
const p = document.getElementsByTagName('p')[0];
p instanceof HTMLParagraphElement
// True
```

An `HTMLParagraphElement` is a subtype of `HTMLElement`, which is a subtype of `Element`, which is a subtype of `Node`, which is a subtype of `EventTarget`. Here are some examples of types along the hierarchy:

Table 7-1. Types in the DOM Hierarchy

Type	Examples
EventTarget	`window`, `XMLHttpRequest`
Node	`document`, `Text`, `Comment`
Element	*includes HTMLElements, SVGElements*
HTMLElement	`<i>`, `<b>`
HTMLButtonElement	`<button>`

An `EventTarget` is the most generic of DOM types. All you can do with it is add event listeners, remove them, and dispatch events. With this in mind, the `classList` errors start to make a bit more sense:

```
function handleDrag(eDown: Event) {
  const targetEl = eDown.currentTarget;
  targetEl.classList.add('dragging');
// ~~~~~~~           Object is possibly 'null'
//         ~~~~~~~~~ Property 'classList' does not exist on type 'EventTarget'
  // ...
}
```

As its name implies, an `Event`'s `currentTarget` property is an `EventTarget`. It could even be `null`. TypeScript has no reason to believe that it has a `classList` property. While an `EventTargets` *could* be an `HTMLElement` in practice, from the type system's perspective there's no reason it couldn't be `window` or `XMLHTTPRequest`.

Moving up the hierarchy we come to `Node`. A couple of examples of `Node`s that are not `Element`s are text fragments and comments. For instance, in this HTML:

```
<p>
  And <i>yet</i> it moves
  <!-- quote from Galileo -->
</p>
```

the outermost element is an `HTMLParagraphElement`. As you can see here, it has `children` and `childNodes`:

```
> p.children
HTMLCollection [i]
> p.childNodes
NodeList(5) [text, i, text, comment, text]
```

`children` returns an `HTMLCollection`, an array-like structure containing just the child `Element`s (`<i>yet</i>`). `childNodes` returns a `NodeList`, an Array-like collection of `Node`s. This includes not just `Element`s (`<i>yet</i>`) but also text fragments ("And," "it moves") and comments ("quote from Galileo").

What's the difference between an `Element` and an `HTMLElement`? There are non-HTML `Element`s including the whole hierarchy of SVG tags. These are `SVGElement`s, which are another type of `Element`. What's the type of an `<html>` or `<svg>` tag? They're `HTMLHtmlElement` and `SVGSvgElement`.

Sometimes these specialized classes will have properties of their own—for example, an `HTMLImageElement` has a `src` property, and an `HTMLInputElement` has a `value` property. If you want to read one of these properties off a value, its type must be specific enough to have that property.

TypeScript's type declarations for the DOM make liberal use of literal types to try to get you the most specific type possible. For example:

```
document.getElementsByTagName('p')[0];  // HTMLParagraphElement
document.createElement('button');  // HTMLButtonElement
document.querySelector('div');  // HTMLDivElement
```

but this is not always possible, notably with `document.getElementById`:

```
document.getElementById('my-div');  // HTMLElement
```

While type assertions are generally frowned upon (Item 9), this is a case where you know more than TypeScript does and so they are appropriate. There's nothing wrong with this, so long as you know that `#my-div` is a div:

```
document.getElementById('my-div') as HTMLDivElement;
```

with `strictNullChecks` enabled, you will need to consider the case that `document.getElementById` returns `null`. Depending on whether this can really happen, you can either add an if statement or an assertion (`!`):

```
const div = document.getElementById('my-div')!;
```

These types are not specific to TypeScript. Rather, they are generated from the formal specification of the DOM. This is an example of the advice of Item 35 to generate types from specs when possible.

So much for the DOM hierarchy. What about the `clientX` and `clientY` errors?

```
function handleDrag(eDown: Event) {
  // ...
  const dragStart = [
     eDown.clientX, eDown.clientY];
        // ~~~~~~~               Property 'clientX' does not exist on 'Event'
        //                ~~~~~~~ Property 'clientY' does not exist on 'Event'
  // ...
}
```

In addition to the hierarchy for `Nodes` and `Elements`, there is also a hierarchy for `Events`. The Mozilla documentation currently lists no fewer than 52 types of `Event`!

Plain `Event` is the most generic type of event. More specific types include:

`UIEvent`
: Any sort of user interface event

`MouseEvent`
: An event triggered by the mouse such as a click

`TouchEvent`
: A touch event on a mobile device

`WheelEvent`
: An event triggered by rotating the scroll wheel

`KeyboardEvent`
: A key press

The problem in `handleDrag` is that the events are declared as `Event`, while `clientX` and `clientY` exist only on the more specific `MouseEvent` type.

So how can you fix the example from the start of this item? TypeScript's declarations for the DOM make extensive use of context (Item 26). Inlining the mousedown handler gives TypeScript more information to work with and removes most of the errors. You can also declare the parameter type to be `MouseEvent` rather than `Event`. Here's a version that uses both techniques to fix the errors:

```
function addDragHandler(el: HTMLElement) {
  el.addEventListener('mousedown', eDown => {
    const dragStart = [eDown.clientX, eDown.clientY];
    const handleUp = (eUp: MouseEvent) => {
      el.classList.remove('dragging');
```

```
      el.removeEventListener('mouseup', handleUp);
      const dragEnd = [eUp.clientX, eUp.clientY];
      console.log('dx, dy = ', [0, 1].map(i => dragEnd[i] - dragStart[i]));
    }
    el.addEventListener('mouseup', handleUp);
  });
}

const div = document.getElementById('surface');
if (div) {
  addDragHandler(div);
}
```

The `if` statement at the end handles the possibility that there is no `#surface` element. If you know that this element exists, you could use an assertion instead (`div!`). `addDragHandler` requires a non-null `HTMLElement`, so this is an example of pushing `null` values to the perimeter (Item 31).

Things to Remember

- The DOM has a type hierarchy that you can usually ignore while writing JavaScript. But these types become more important in TypeScript. Understanding them will help you write TypeScript for the browser.
- Know the differences between `Node`, `Element`, `HTMLElement`, and `EventTarget`, as well as those between `Event` and `MouseEvent`.
- Either use a specific enough type for DOM elements and Events in your code or give TypeScript the context to infer it.

Item 56: Don't Rely on Private to Hide Information

JavaScript has historically lacked a way to make properties of a class private. The usual workaround is a convention of prefixing fields that are not part of a public API with underscores:

```
class Foo {
  _private = 'secret123';
}
```

But this only discourages users from accessing private data. It is easy to circumvent:

```
const f = new Foo();
f._private;  // 'secret123'
```

TypeScript adds `public`, `protected`, and `private` field modifiers that seem to provide some enforcement:

```
class Diary {
  private secret = 'cheated on my English test';
}

const diary = new Diary();
diary.secret
   // ~~~~~~ Property 'secret' is private and only
   //        accessible within class 'Diary'
```

But `private` is a feature of the type system and, like all features of the type system, it goes away at runtime (see Item 3). Here's what this snippet looks like when TypeScript compiles it to JavaScript (with `target=ES2017`):

```
class Diary {
  constructor() {
    this.secret = 'cheated on my English test';
  }
}
const diary = new Diary();
diary.secret;
```

The `private` indicator is gone, and your secret is out! Much like the `_private` convention, TypeScript's access modifiers only discourage you from accessing private data. With a type assertion, you can even access a private property from within TypeScript:

```
class Diary {
  private secret = 'cheated on my English test';
}

const diary = new Diary();
(diary as any).secret  // OK
```

In other words, *don't rely on `private` to hide information!*

So what should you do if you want something more robust? The traditional answer has been to take advantage of one of JavaScript's most reliable ways to hide information: closures. You can create one in a constructor:

```
declare function hash(text: string): number;

class PasswordChecker {
  checkPassword: (password: string) => boolean;
  constructor(passwordHash: number) {
    this.checkPassword = (password: string) => {
      return hash(password) === passwordHash;
    }
  }
}

const checker = new PasswordChecker(hash('s3cret'));
checker.checkPassword('s3cret');  // Returns true
```

JavaScript offers no way to access the `passwordHash` variable from outside of the constructor of `PasswordChecker`. This does have a few downsides, however: specifically, because `passwordHash` can't be seen outside the constructor, every method that uses it also has to be defined there. This results in a copy of each method being created for every class instance, which will lead to higher memory use. It also prevents other instances of the same class from accessing private data. Closures may be inconvenient, but they will certainly keep your data private!

A newer option is to use private fields, a proposed language feature that is solidifying as this book goes to print. In this proposal, to make a field private both for type checking and at runtime, prefix it with a `#`:

```
class PasswordChecker {
  #passwordHash: number;

  constructor(passwordHash: number) {
    this.#passwordHash = passwordHash;
  }

  checkPassword(password: string) {
    return hash(password) === this.#passwordHash;
  }
}

const checker = new PasswordChecker(hash('s3cret'));
checker.checkPassword('secret');  // Returns false
checker.checkPassword('s3cret');  // Returns true
```

The `#passwordHash` property is not accessible from outside the class. In contrast to the closure technique, it *is* accessible from class methods and from other instances of the same class. For ECMAScript targets that don't natively support private fields, a fallback implementation using `WeakMaps` is used instead. The upshot is that your data is still private. This proposal was stage 3 and support was being added to TypeScript as this book went to print. If you'd like to use it, check the TypeScript release notes to see if it's generally available.

Finally, if you are worried about *security*, rather than just encapsulation, then there are others concerns to be aware of such as modifications to built-in prototypes and functions.

Things to Remember

- The `private` access modifier is only enforced through the type system. It has no effect at runtime and can be bypassed with an assertion. Don't assume it will keep data hidden.
- For more reliable information hiding, use a closure.

Item 57: Use Source Maps to Debug TypeScript

When you run TypeScript code, you're actually running the JavaScript that the TypeScript compiler generates. This is true of any source-to-source compiler, be it a minifier, a compiler, or a preprocessor. The hope is that this is mostly transparent, that you can pretend that the TypeScript source code is being executed without ever having to look at the JavaScript.

This works well until you have to debug your code. Debuggers generally work on the code you're executing and don't know about the translation process it went through. Since JavaScript is such a popular target language, browser vendors collaborated to solve this problem. The result is source maps. They map positions and symbols in a generated file back to the corresponding positions and symbols in the original source. Most browsers and many IDEs support them. If you're not using them to debug your TypeScript, you're missing out!

Suppose you've created a small script to add a button to an HTML page that increments every time you click it:

```
function addCounter(el: HTMLElement) {
  let clickCount = 0;
  const button = document.createElement('button');
  button.textContent = 'Click me';
  button.addEventListener('click', () => {
    clickCount++;
    button.textContent = `Click me (${clickCount})`;
  });
  el.appendChild(button);
}

addCounter(document.body);
```

If you load this in your browser and open the debugger, you'll see the generated JavaScript. This closely matches the original source, so debugging isn't too difficult, as you can see in Figure 7-1.

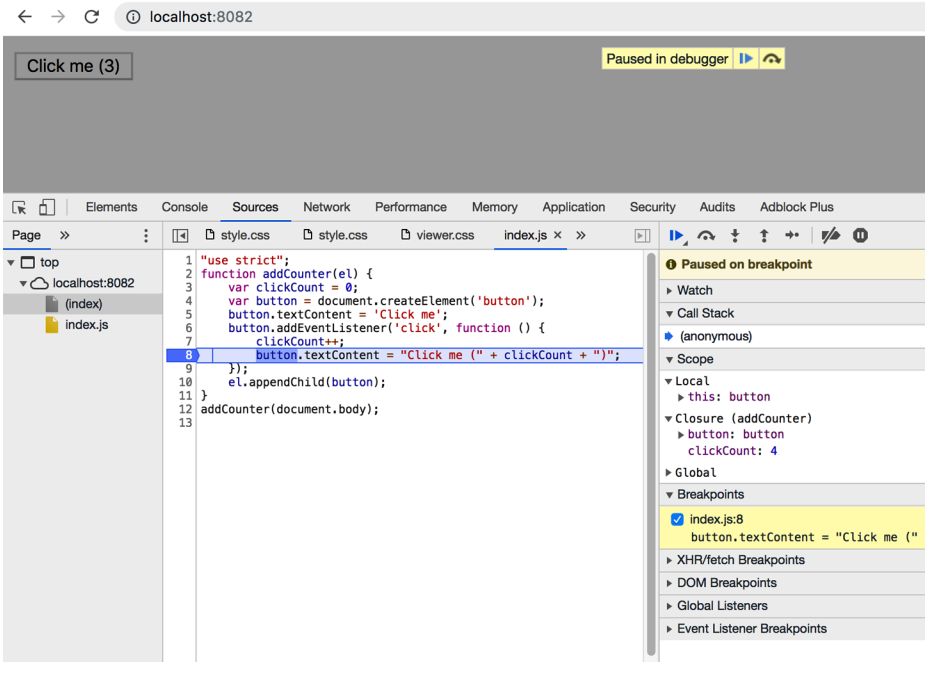

Figure 7-1. Debugging generated JavaScript using Chrome's developer tools. For this simple example, the generated JavaScript closely resembles the TypeScript source.

Let's make the page more fun by fetching an interesting fact about each number from numbersapi.com:

```
function addCounter(el: HTMLElement) {
  let clickCount = 0;
  const triviaEl = document.createElement('p');
  const button = document.createElement('button');
  button.textContent = 'Click me';
  button.addEventListener('click', async () => {
    clickCount++;
    const response = await fetch(`http://numbersapi.com/${clickCount}`);
    const trivia = await response.text();
    triviaEl.textContent = trivia;
    button.textContent = `Click me (${clickCount})`;
  });
  el.appendChild(triviaEl);
  el.appendChild(button);
}
```

If you open up your browser's debugger now, you'll see that the generated source has gotten dramatically more complicated (see Figure 7-2).

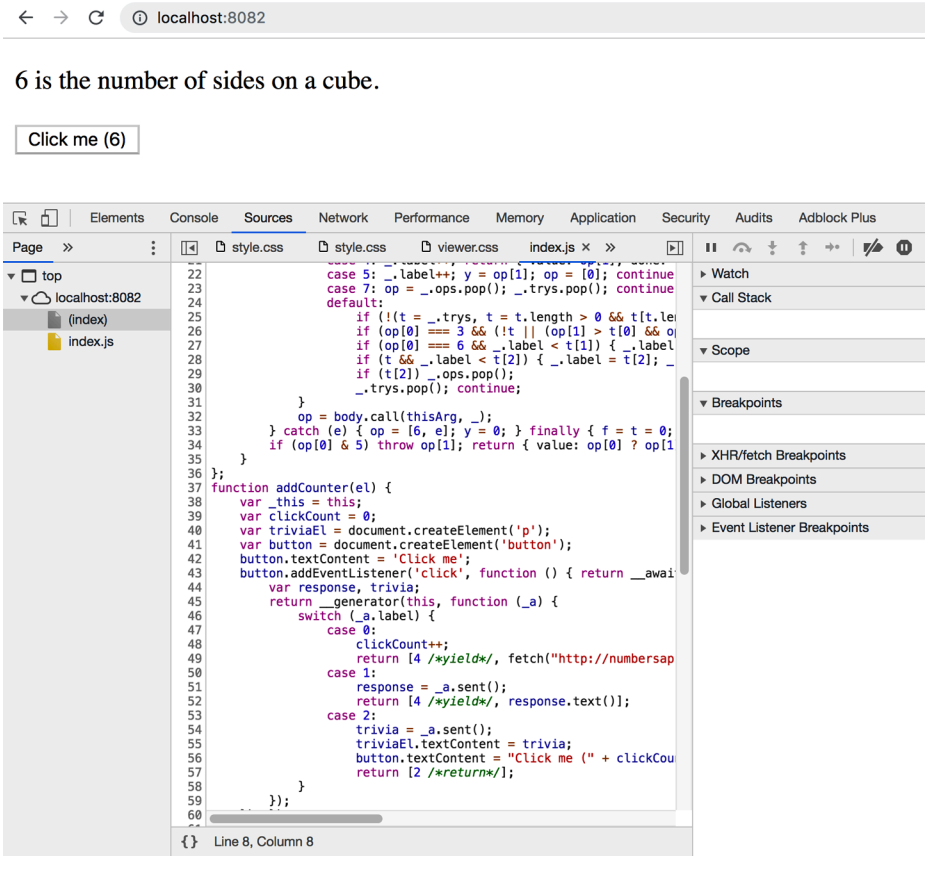

Figure 7-2. In this case the TypeScript compiler has generated JavaScript that doesn't closely resemble the original TypeScript source. This will make debugging more difficult.

To support `async` and `await` in older browsers, TypeScript has rewritten the event handler as a state machine. This has the same behavior, but the code no longer bears such a close resemblance to the original source.

This is where source maps can help. To tell TypeScript to generate one, set the `sourceMap` option in your *tsconfig.json*:

```
{
  "compilerOptions": {
    "sourceMap": true
  }
}
```

Now when you run `tsc`, it generates two output files for each *.ts* file: a *.js* file and a *.js.map* file. The latter is the source map.

With this file in place, a new *index.ts* file appears in your browser's debugger. You can set breakpoints and inspect variables in it, just as you'd hope (see Figure 7-3).

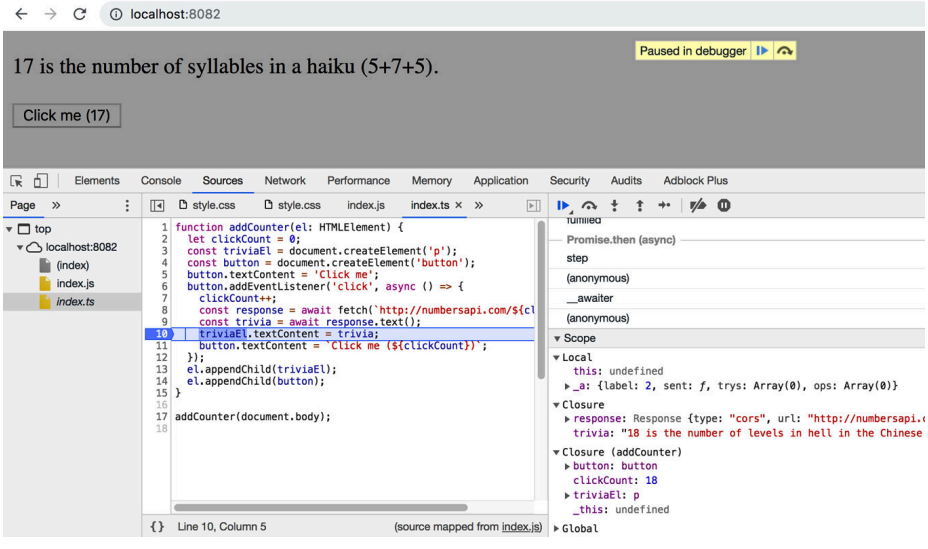

Figure 7-3. When a source map is present, you can work with the original TypeScript source in your debugger, rather than the generated JavaScript.

Note that *index.ts* appears in italics in the file list on the left. This indicates that it isn't a "real" file in the sense that the web page included it. Rather, it was included via the source map. Depending on your settings, *index.js.map* will contain either a reference to *index.ts* (in which case the browser loads it over the network) or an inline copy of it (in which case no request is needed).

There are a few things to be aware of with source maps:

- If you are using a bundler or minifier with TypeScript, it may generate a source map of its own. To get the best debugging experience, you want this to map all the way back to the original TypeScript sources, not the generated JavaScript. If your bundler has built-in support for TypeScript, then this should just work. If not, you may need to hunt down some flags to make it read source map inputs.
- Be aware of whether you're serving source maps in production. The browser won't load source maps unless the debugger is open, so there's no performance impact for end users. But if the source map contains an inline copy of your original source code, then there may be content that you didn't intend to publicize. Does the world really need to see your snarky comments or internal bug tracker URLs?

You can also debug NodeJS programs using source maps. This is typically done via your editor or by connecting to your node process from a browser's debugger. Consult the Node docs for details.

The type checker can catch many errors before you run your code, but it is no substitute for a good debugger. Use source maps to get a great TypeScript debugging experience.

Things to Remember

- Don't debug generated JavaScript. Use source maps to debug your TypeScript code at runtime.
- Make sure that your source maps are mapped all the way through to the code that you run.
- Depending on your settings, your source maps might contain an inline copy of your original code. Don't publish them unless you know what you're doing!