CHAPTER 6

Types Declarations and @types

Dependency management can be confusing in any language, and TypeScript is no exception. This chapter will help you build a mental model for how dependencies work in TypeScript and show you how to work through some of the issues that can come up with them. It will also help you craft your own type declaration files to publish and share with others. By writing great type declarations, you can help not just your own project but the entire TypeScript community.

Item 45: Put TypeScript and @types in devDependencies

The Node Package Manager, npm, is ubiquitous in the JavaScript world. It provides both a repository of JavaScript libraries (the npm registry) and a way to specify which versions of them you depend on (*package.json*).

npm draws a distinction between a few types of dependencies, each of which goes in a separate section of *package.json*:

`dependencies`
: These are packages that are required to run your JavaScript. If you import `lodash` at runtime, then it should go in `dependencies`. When you publish your code on npm and another user installs it, it will also install these dependencies. (These are known as transitive dependencies.)

`devDependencies`
: These packages are used to develop and test your code but are not required at runtime. Your test framework would be an example of a `devDependency`. Unlike `dependencies`, these are *not* installed transitively with your packages.

`peerDependencies`
: These are packages that you require at runtime but don't want to be responsible for tracking. The canonical example is a plug-in. Your jQuery plug-in is compatible with a range of versions of jQuery itself, but you'd prefer that the user select one, rather than you choosing for them.

Of these, `dependencies` and `devDependencies` are by far the most common. As you use TypeScript, be aware of which type of dependency you're adding. Because TypeScript is a development tool and TypeScript types do not exist at runtime (Item 3), packages related to TypeScript generally belong in `devDependencies`.

The first dependency to consider is TypeScript itself. It is possible to install TypeScript system-wide, but this is generally a bad idea for two reasons:

- There's no guarantee that you and your coworkers will always have the same version installed.
- It adds a step to your project setup.

Make TypeScript a `devDependency` instead. That way you and your coworkers will always get the correct version when you run `npm install`. And updating your TypeScript version follows the same pattern as updating any other package.

Your IDE and build tools will happily discover a version of TypeScript installed in this way. On the command line you can use `npx` to run the version of `tsc` installed by npm:

```
$ npx tsc
```

The next type of dependency to consider is *type dependencies* or `@types`. If a library itself does not come with TypeScript type declarations, then you may still be able to find typings on DefinitelyTyped, a community-maintained collection of type definitions for JavaScript libraries. Type definitions from DefinitelyTyped are published on the npm registry under the `@types` scope: `@types/jquery` has type definitions for the jQuery, `@types/lodash` has types for Lodash, and so on. These `@types` packages only contain the *types*. They don't contain the implementation.

Your `@types` dependencies should also be `devDependencies`, even if the package itself is a direct dependency. For example, to depend on React and its type declarations, you might run:

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

This will result in a *package.json* file that looks something like this:

```
{
  "devDependencies": {
    "@types/lodash": "^16.8.19",
    "typescript": "^3.5.3"
  },
  "dependencies": {
    "react": "^16.8.6"
  }
}
```

The idea here is that you should publish JavaScript, not TypeScript, and your JavaScript does not depend on the `@types` when you run it. There are a few things that can go wrong with `@types` dependencies, and the next item will delve deeper into this topic.

Things to Remember

- Avoid installing TypeScript system-wide. Make TypeScript a `devDependency` of your project to ensure that everyone on the team is using a consistent version.
- Put `@types` dependencies in `devDependencies`, not `dependencies`. If you need `@types` at runtime, then you may want to rework your process.

Item 46: Understand the Three Versions Involved in Type Declarations

Dependency management rarely conjures up happy feelings for software developers. Usually you just want to use a library and not think too much about whether its transitive dependencies are compatible with yours.

The bad news is that TypeScript doesn't make this any better. In fact, it makes dependency management quite a bit *more* complicated. This is because instead of having a single version to worry about, you now have three:

- The version of the package
- The version of its type declarations (`@types`)
- The version of TypeScript

If any of these versions get out of sync with one another, you can run into errors that may not be clearly related to dependency management. But as the saying goes, "make things as simple as possible but no simpler." Understanding the full complexity of TypeScript package management will help you diagnose and fix problems. And it will help you make more informed decisions when it comes time to publish type declarations of your own.

Here's how dependencies in TypeScript are supposed to work. You install a package as a direct dependency, and you install its types as a dev dependency (see Item 45):

```
$ npm install react
+ react@16.8.6

$ npm install --save-dev @types/react
+ @types/react@16.8.19
```

Note that the major and minor versions (`16.8`) match but that the patch versions (`.6` and `.19`) do not. This is exactly what you want to see. The `16.8` in the `@types` version means that these type declarations describe the API of version `16.8` of `react`. Assuming the `react` module follows good semantic versioning hygiene, the patch versions (`16.8.1`, `16.8.2`, ...) will not change its public API and will not require updates to the type declarations. But the type declarations *themselves* might have bugs or omissions. The patch versions of the `@types` module correspond to these sorts of fixes and additions. In this case, there were many more updates to the type declarations than the library itself (19 versus 6).

This can go wrong in a few ways.

First, you might update a library but forget to update its type declarations. In this case you'll get type errors whenever you try to use new features of the library. If there were breaking changes to the library, you might get runtime errors despite your code passing the type checker.

The solution is usually to update your type declarations so that the versions are back in sync. If the type declarations have not been updated, you have a few options. You can use an augmentation in your own project to add new functions and methods that you'd like to use. Or you can contribute updated type declarations back to the community.

Second, your type declarations might get ahead of your library. This can happen if you've been using a library without its typings (perhaps you gave it an `any` type using `declare module`) and try to install them later. If there have been new releases of the library and its type declarations, your versions might be out of sync. The symptoms of this are similar to the first problem, just in reverse. The type checker will be comparing your code against the latest API, while you'll be using an older one at runtime. The solution is to either upgrade the library or downgrade the type declarations until they match.

Third, the type declarations might require a newer version of TypeScript than you're using in your project. Much of the development of TypeScript's type system has been motivated by an attempt to more precisely type popular JavaScript libraries like Lodash, React, and Ramda. It makes sense that the type declarations for these libraries would want to use the latest and greatest features to get you better type safety.

If this happens, you'll experience it as type errors in the `@types` declarations themselves. The solution is to either upgrade your TypeScript version, use an older version of the type declarations, or, if you really can't update TypeScript, stub out the types with `declare module`. It is possible for a library to provide different type declarations for different versions of TypeScript via `typesVersions`, but this is rare: at the time of this writing, fewer than 1% of the packages on DefinitelyTyped did so.

To install `@types` for a specific version of TypeScript, you can use:

```
npm install --save-dev @types/lodash@ts3.1
```

The version matching between libraries and their types is best effort and may not always be correct. But the more popular the library is, the more likely it is that its type declarations will get this right.

Fourth, you can wind up with duplicate `@types` dependencies. Say you depend on `@types/foo` and `@types/bar`. If `@types/bar` depends on an incompatible version of `@types/foo`, then npm will attempt to resolve this by installing both versions, one in a nested folder:

```
node_modules/
  @types/
    foo/
      index.d.ts @1.2.3
    bar/
      index.d.ts
      node_modules/
        @types/
          foo/
            index.d.ts @2.3.4
```

While this is sometimes OK for node modules that are used at runtime, it almost certainly won't be OK for type declarations, which live in a flat global namespace. You'll see this as errors about duplicate declarations or declarations that cannot be merged. You can track down why you have a duplicate type declaration by running `npm ls @types/foo`. The solution is typically to update your dependency on `@types/foo` or `@types/bar` so that they are compatible. Transitive `@types` dependencies like these are often a source of trouble. If you're publishing types, see Item 51 for ways to avoid them.

Some packages, particularly those written in TypeScript, choose to bundle their own type declarations. This is usually indicated by a `"types"` field in their *package.json* which points to a *.d.ts* file:

```
{
  "name": "left-pad",
  "version": "1.3.0",
  "description": "String left pad",
  "main": "index.js",
  "types": "index.d.ts",
  // ...
}
```

Does this solve all our problems? Would I even be asking if the answer was "yes"?

Bundling types *does* solve the problem of version mismatch, particularly if the library itself is written in TypeScript and the type declarations are generated by `tsc`. But bundling has some problems of its own.

First, what if there's an error in the bundled types that can't be fixed through augmentation? Or the types worked fine when they were published, but a new TypeScript version has since been released which flags an error. With `@types` you could depend on the library's implementation but not its type declarations. But with bundled types, you lose this option. One bad type declaration might keep you stuck on an old version of TypeScript. Contrast this with DefinitelyTyped: as TypeScript is developed, Microsoft runs it against all the type declarations on DefinitelyTyped. Breaks are fixed quickly.

Second, what if your types depend on another library's type declarations? Usually this would be a `devDependency` (Item 45). But if you publish your module and another user installs it, they won't get your `devDependencies`. Type errors will result. On the other hand, you probably don't want to make it a direct dependency either, since then your JavaScript users will install `@types` modules for no reason. Item 51 discusses the standard workaround for this situation. But if you publish your types on DefinitelyTyped, this is not a problem at all: you declare your type dependency there and only your TypeScript users will get it.

Third, what if you need to fix an issue with the type declarations of an old version of your library? Would you be able to go back and release a patch update? DefinitelyTyped has mechanisms for simultaneously maintaining type declarations for different versions of the same library, something that might be hard for you to do in your own project.

Fourth, how committed to accepting patches for type declarations are you? Remember the versions of `react` and `@types/react` from the start of this item. There were three times more patch updates to the type declarations than the library itself. DefinitelyTyped is community-maintained and is able to handle this volume. In particular, if a library maintainer doesn't look at a patch within five days, a global maintainer will. Can you commit to a similar turnaround time for your library?

Managing dependencies in TypeScript can be challenging, but it does come with rewards: well-written type declarations can help you learn how to use libraries correctly and can greatly improve your productivity with them. As you run into issues with dependency management, keep the three versions in mind.

If you are publishing packages, weigh the pros and cons of bundling type declarations versus publishing them on DefinitelyTyped. The official recommendation is to bundle type declarations only if the library is written in TypeScript. This works well in practice since `tsc` can automatically generate type declarations for you (using the `declaration` compiler option). For JavaScript libraries, handcrafted type declarations are more likely to contain errors, and they'll require more updates. If you publish your type declarations on DefinitelyTyped, the community will help you support and maintain them.

Things to Remember

- There are three versions involved in an `@types` dependency: the library version, the `@types` version, and the TypeScript version.
- If you update a library, make sure you update the corresponding `@types`.
- Understand the pros and cons of bundling types versus publishing them on DefinitelyTyped. Prefer bundling types if your library is written in TypeScript and DefinitelyTyped if it is not.

Item 47: Export All Types That Appear in Public APIs

Use TypeScript long enough and you'll eventually find yourself wanting to use a `type` or `interface` from a third-party module only to find that it isn't exported. Fortunately TypeScript's tools for mapping between types are rich enough that, as a library user, you can almost always find a way to reference the type you want. As a library author, this means that you ought to just export your types to begin with. If a type ever appears in a function declaration, it is effectively exported. So you may as well make things explicit.

Suppose you want to create some secret, unexported types:

```
interface SecretName {
  first: string;
  last: string;
}

interface SecretSanta {
  name: SecretName;
  gift: string;
}

export function getGift(name: SecretName, gift: string): SecretSanta {
  // ...
}
```

As a user of your module, I cannot directly import `SecretName` or `SecretSanta`, only `getGift`. But this is no barrier: because those types appear in an exported function signature, I can extract them. One way is to use the `Parameters` and `ReturnType` generic types:

```
type MySanta = ReturnType<typeof getGift>;  // SecretSanta
type MyName = Parameters<typeof getGift>[0];  // SecretName
```

If your goal in not exporting these types was to preserve flexibility, then the jig is up! You've already committed to them by putting them in a public API. Do your users a favor and export them.

Things to Remember

- Export types that appear in any form in any public method. Your users will be able to extract them anyway, so you may as well make it easy for them.

Item 48: Use TSDoc for API Comments

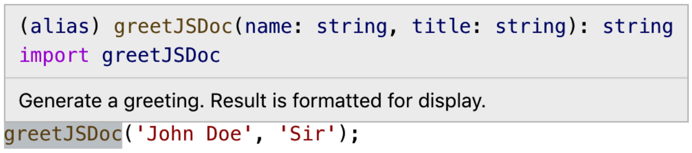
Here's a TypeScript function to generate a greeting:

```
// Generate a greeting. Result is formatted for display.
function greet(name: string, title: string) {
  return `Hello ${title} ${name}`;
}
```

The author was kind enough to leave a comment describing what this function does. But for documentation intended to be read by users of your functions, it's better to use JSDoc-style comments:

```
/** Generate a greeting. Result is formatted for display. */
function greetJSDoc(name: string, title: string) {
  return `Hello ${title} ${name}`;
}
```

The reason is that there is a nearly universal convention in editors to surface JSDoc-style comments when the function is called (see Figure 6-1).



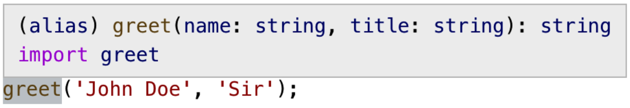
```
(alias) greetJSDoc(name: string, title: string): string
import greetJSDoc
```

Generate a greeting. Result is formatted for display.

```
greetJSDoc('John Doe', 'Sir');
```

Figure 6-1. JSDoc-style comments are typically surfaced in tooltips in your editor.

Whereas the inline comment gets no such treatment (see Figure 6-2).



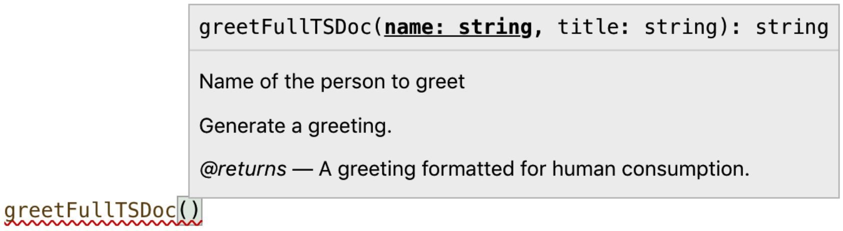
```
(alias) greet(name: string, title: string): string
import greet
greet('John Doe', 'Sir');
```

Figure 6-2. Inline comments are typically not shown in tooltips.

The TypeScript language service supports this convention, and you should take advantage of it. If a comment describes a public API, it should be JSDoc. In the context of TypeScript, these comments are sometimes called TSDoc. You can use many of the usual conventions like `@param` and `@returns`:

```
/**
 * Generate a greeting.
 * @param name Name of the person to greet
 * @param salutation The person's title
 * @returns A greeting formatted for human consumption.
 */
function greetFullTSDoc(name: string, title: string) {
  return `Hello ${title} ${name}`;
}
```

This lets editors show the relevant documentation for each parameter as you're writing out a function call (as shown in Figure 6-3).



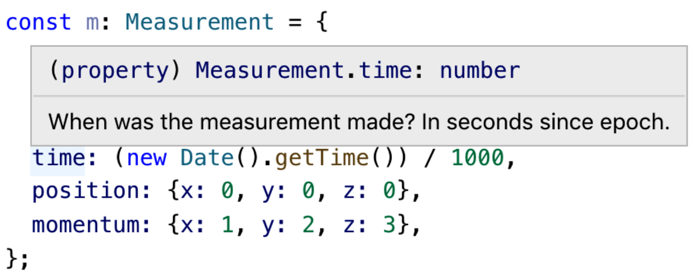
```
greetFullTSDoc(name: string, title: string): string

Name of the person to greet

Generate a greeting.

@returns — A greeting formatted for human consumption.
greetFullTSDoc()
```

Figure 6-3. An @param annotation lets your editor show documentation for the current parameter as you type it.

You can also use TSDoc with type definitions:

```
/** A measurement performed at a time and place. */
interface Measurement {
  /** Where was the measurement made? */
  position: Vector3D;
  /** When was the measurement made? In seconds since epoch. */
  time: number;
  /** Observed momentum */
  momentum: Vector3D;
}
```

As you inspect individual fields in a `Measurement` object, you'll get contextual documentation (see Figure 6-4).

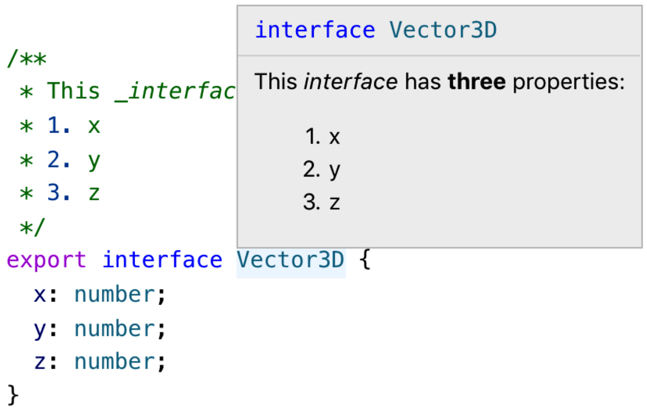


```
const m: Measurement = {
  (property) Measurement.time: number
  When was the measurement made? In seconds since epoch.
  time: (new Date().getTime()) / 1000,
  position: {x: 0, y: 0, z: 0},
  momentum: {x: 1, y: 2, z: 3},
};
```

Figure 6-4. TSDoc for a field is shown when you mouse over that field in your editor.

TSDoc comments are formatted using Markdown, so if you want to use bold, italic, or bulleted lists, you can (see Figure 6-5):

```
/**
 * This _interface_ has **three** properties:
 * 1. x
 * 2. y
 * 3. z
 */
interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```



```
/**
 * This _interfac
 * 1. x
 * 2. y
 * 3. z
 */
export interface Vector3D {
  x: number;
  y: number;
  z: number;
}
```

interface Vector3D

This *interface* has **three** properties:

1. x
2. y
3. z

Figure 6-5. TSDoc comments

Try to avoid writing essays in your documentation, though: the best comments are short and to the point.

JSDoc includes some conventions for specifying type information (`@param {string} name ...`), but you should avoid these in favor of TypeScript types (Item 30).

Things to Remember

- Use JSDoc-/TSDoc-formatted comments to document exported functions, classes, and types. This helps editors surface information for your users when it's most relevant.
- Use `@param`, `@returns`, and Markdown for formatting.
- Avoid including type information in documentation (see Item 30).

Item 49: Provide a Type for this in Callbacks

JavaScript's `this` keyword is one of the most notoriously confusing parts of the language. Unlike variables declared with `let` or `const`, which are lexically scoped, `this` is dynamically scoped: its value depends not on the way in which it was *defined* but on the way in which it was *called*.

`this` is most often used in classes, where it typically references the current instance of an object:

```
class C {
  vals = [1, 2, 3];
  logSquares() {
    for (const val of this.vals) {
      console.log(val * val);
    }
  }
}

const c = new C();
c.logSquares();
```

This logs:

```
1
4
9
```

Now look what happens if you try to put `logSquares` in a variable and call that:

```
const c = new C();
const method = c.logSquares;
method();
```

This version throws an error at runtime:

```
Uncaught TypeError: Cannot read property 'vals' of undefined
```

The problem is that `c.logSquares()` actually does two things: it calls `C.prototype.logSquares` *and* it binds the value of `this` in that function to `c`. By pulling out a reference to `logSquares`, you've separated these, and `this` gets set to `undefined`.

JavaScript gives you complete control over `this` binding. You can use `call` to explicitly set `this` and fix the problem:

```
const c = new C();
const method = c.logSquares;
method.call(c);  // Logs the squares again
```

There's no reason that `this` had to be bound to an instance of `C`. It could have been bound to anything. So libraries can, and do, make the value of `this` part of their APIs. Even the DOM makes use of this. In an event handler, for instance:

```
document.querySelector('input')!.addEventListener('change', function(e) {
  console.log(this);  // Logs the input element on which the event fired.
});
```

`this` binding often comes up in the context of callbacks like this one. If you want to define an `onClick` handler in a class, for example, you might try this:

```
class ResetButton {
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick() {
    alert(`Reset ${this}`);
  }
}
```

When `Button` calls `onClick`, it will alert "Reset undefined." Oops! As usual, the culprit is `this` binding. A common solution is to create a bound version of the method in the constructor:

```
class ResetButton {
  constructor() {
    this.onClick = this.onClick.bind(this);
  }
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick() {
    alert(`Reset ${this}`);
  }
}
```

The `onClick() { ... }` definition defines a property on `ResetButton.prototype`. This is shared by all instances of `ResetButton`. When you bind `this.onClick = ...` in the constructor, it creates a property called `onClick` on the instance of `ResetButton` with `this` bound to that instance. The `onClick` instance property comes before the `onClick` prototype property in the lookup sequence, so `this.onClick` refers to the bound function in the `render()` method.

There is a shorthand for binding that can sometimes be convenient:

```
class ResetButton {
  render() {
    return makeButton({text: 'Reset', onClick: this.onClick});
  }
  onClick = () => {
    alert(`Reset ${this}`);  // "this" always refers to the ResetButton instance.
  }
}
```

Here we've replaced `onClick` with an arrow function. This will define a new function every time a `ResetButton` is constructed with `this` set to the appropriate value. It's instructive to look at the JavaScript that this generates:

```
class ResetButton {
  constructor() {
    var _this = this;
    this.onClick = function () {
      alert("Reset " + _this);
    };
  }
  render() {
    return makeButton({ text: 'Reset', onClick: this.onClick });
  }
}
```

So what does this all have to do with TypeScript? Because `this` binding is part of JavaScript, TypeScript models it. This means that if you're writing (or typing) a library that sets the value of `this` on callbacks, then you should model this, too.

You do this by adding a `this` parameter to your callback:

```
function addKeyListener(
  el: HTMLElement,
  fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
  el.addEventListener('keydown', e => {
    fn.call(el, e);
  });
}
```

The this parameter is special: it's not just another positional argument. You can see this if you try to call it with two parameters:

```
function addKeyListener(
  el: HTMLElement,
  fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
  el.addEventListener('keydown', e => {
    fn(el, e);
        // ~ Expected 1 arguments, but got 2
  });
}
```

Even better, TypeScript will enforce that you call the function with the correct this context:

```
function addKeyListener(
  el: HTMLElement,
  fn: (this: HTMLElement, e: KeyboardEvent) => void
) {
  el.addEventListener('keydown', e => {
    fn(e);
 // ~~~~~ The 'this' context of type 'void' is not assignable
 //       to method's 'this' of type 'HTMLElement'
  });
}
```

As a user of this function, you can reference this in the callback and get full type safety:

```
declare let el: HTMLElement;
addKeyListener(el, function(e) {
  this.innerHTML;  // OK, "this" has type of HTMLElement
});
```

Of course, if you use an arrow function here, you'll override the value of this. TypeScript will catch the issue:

```
class Foo {
  registerHandler(el: HTMLElement) {
    addKeyListener(el, e => {
      this.innerHTML;
        // ~~~~~~~~~ Property 'innerHTML' does not exist on type 'Foo'
    });
  }
}
```

Don't forget about this! If you set the value of this in your callbacks, then it's part of your API, and you should include it in your type declarations.

Things to Remember

- Understand how `this` binding works.
- Provide a type for `this` in callbacks when it's part of your API.

Item 50: Prefer Conditional Types to Overloaded Declarations

How would you write a type declaration for this JavaScript function?

```
function double(x) {
  return x + x;
}
```

`double` can be passed either a `string` or a `number`. So you might use a union type:

```
function double(x: number|string): number|string;
function double(x: any) { return x + x; }
```

(These examples all make use of TypeScript's concept of function overloading. For a refresher, see Item 3.)

While this declaration is accurate, it's a bit imprecise:

```
const num = double(12);  // string | number
const str = double('x');  // string | number
```

When `double` is passed a `number`, it returns a `number`. And when it's passed a `string`, it returns a `string`. This declaration misses that nuance and will produce types that are hard to work with.

You might try to capture this relationship using a generic:

```
function double<T extends number|string>(x: T): T;
function double(x: any) { return x + x; }

const num = double(12);  // Type is 12
const str = double('x');  // Type is "x"
```

Unfortunately, in our zeal for precision we've overshot. The types are now a little *too* precise. When passed a `string` type, this `double` declaration will result in a `string` type, which is correct. But when passed a string *literal* type, the return type is the same string literal type. This is wrong: doubling `'x'` results in `'xx'`, not `'x'`.

Another option is to provide multiple type declarations. While TypeScript only allows you to write one implementation of a function, it allows you to write any number of type declarations. You can use this to improve the type of `double`:

```
function double(x: number): number;
function double(x: string): string;
function double(x: any) { return x + x; }

const num = double(12);  // Type is number
const str = double('x');  // Type is string
```

This is progress! But is this declaration correct? Unfortunately there's still a subtle bug. This type declaration will work with values that are either a `string` or a `number`, but not with values that could be either:

```
function f(x: number|string) {
  return double(x);
           // ~ Argument of type 'string | number' is not assignable
           //   to parameter of type 'string'
}
```

This call to `double` is safe and should return `string|number`. When you overload type declarations, TypeScript processes them one by one until it finds a match. The error you're seeing is a result of the last overload (the `string` version) failing, because `string|number` is not assignable to `string`.

While you could patch this issue by adding a third `string|number` overload, the best solution is to use a *conditional type*. Conditional types are like if statements (conditionals) in type space. They're perfect for situations like this one where there are a few possibilities that you need to cover:

```
function double<T extends number | string>(
  x: T
): T extends string ? string : number;
function double(x: any) { return x + x; }
```

This is similar to the first attempt to type `double` using a generic, but with a more elaborate return type. You read the conditional type like you'd read a ternary (`?:`) operator in JavaScript:

- If `T` is a subset of `string` (e.g., `string` or a string literal or a union of string literals), then the return type is `string`.
- Otherwise return `number`.

With this declaration, all of our examples work:

```
const num = double(12);  // number
const str = double('x');  // string

// function f(x: string | number): string | number
function f(x: number|string) {
  return double(x);
}
```

The `number|string` example works because conditional types distribute over unions. When `T` is `number|string`, TypeScript resolves the conditional type as follows:

```
   (number|string) extends string ? string : number
-> (number extends string ? string : number) |
   (string extends string ? string : number)
-> number | string
```

While the type declaration using overloading was simpler to write, the version using conditional types is more correct because it generalizes to the union of the individual cases. This is often the case for overloads. Whereas overloads are treated independently, the type checker can analyze conditional types as a single expression, distributing them over unions. If you find yourself writing an overloaded type declarations, consider whether it might be better expressed using a conditional type.

Things to Remember

- Prefer conditional types to overloaded type declarations. By distributing over unions, conditional types allow your declarations to support union types without additional overloads.

Item 51: Mirror Types to Sever Dependencies

Suppose you've written a library for parsing CSV files. Its API is simple: you pass in the contents of the CSV file and get back a list of objects mapping column names to values. As a convenience for your NodeJS users, you allow the contents to be either a `string` or a NodeJS `Buffer`:

```
function parseCSV(contents: string | Buffer): {[column: string]: string}[]  {
  if (typeof contents === 'object') {
    // It's a buffer
    return parseCSV(contents.toString('utf8'));
  }
  // ...
}
```

The type definition for `Buffer` comes from the NodeJS type declarations, which you must install:

```
npm install --save-dev @types/node
```

When you publish your CSV parsing library, you include the type declarations with it. Since your type declarations depend on the NodeJS types, you include these as a `devDependency` (Item 45). If you do this, you're liable to get complaints from two groups of users:

- JavaScript developers who wonder what these `@types` modules are that they're depending on.
- TypeScript web developers who wonder why they're depending on NodeJS.

These complaints are reasonable. The `Buffer` behavior isn't essential and is only relevant for users who are using NodeJS already. And the declaration in `@types/node` is only relevant to NodeJS users who are also using TypeScript.

TypeScript's structural typing (Item 4) can help you out of the jam. Rather than using the declaration of `Buffer` from `@types/node`, you can write your own with just the methods and properties you need. In this case that's just a `toString` method that accepts an encoding:

```
interface CsvBuffer {
  toString(encoding: string): string;
}
function parseCSV(contents: string | CsvBuffer): {[column: string]: string}[]  {
  // ...
}
```

This interface is dramatically shorter than the complete one, but it does capture our (simple) needs from a `Buffer`. In a NodeJS project, calling `parseCSV` with a real `Buffer` is still OK because the types are compatible:

```
parseCSV(new Buffer("column1,column2\nval1,val2", "utf-8"));  // OK
```

If your library only depends on the types for another library, rather than its implementation, consider mirroring just the declarations you need into your own code. This will result in a similar experience for your TypeScript users and an improved experience for everyone else.

If you depend on the implementation of a library, you may still be able to apply the same trick to avoid depending on its typings. But this becomes increasingly difficult as the dependence grows larger and more essential. If you're copying a large portion of the type declarations for another library, you may want to formalize the relationship by making the `@types` dependency explicit.

This technique is also helpful for severing dependencies between your unit tests and production systems. See the `getAuthors` example in Item 4.

Things to Remember

- Use structural typing to sever dependencies that are nonessential.
- Don't force JavaScript users to depend on `@types`. Don't force web developers to depend on NodeJS.

Item 52: Be Aware of the Pitfalls of Testing Types

You wouldn't publish code without writing tests for it (I hope!), and you shouldn't publish type declarations without writing tests for them, either. But how do you test types? If you're authoring type declarations, testing is an essential but surprisingly fraught undertaking. It's tempting to make assertions about types inside the type system using the tools that TypeScript provides. But there are several pitfalls with this approach. Ultimately it's safer and more straightforward to use `dtslint` or a similar tool that inspects types from outside of the type system.

Suppose you've written a type declaration for a `map` function provided by a utility library (the popular Lodash and Underscore libraries both provide such a function):

```
declare function map<U, V>(array: U[], fn: (u: U) => V): V[];
```

How can you check that this type declaration results in the expected types? (Presumably there are separate tests for the implementation.) One common technique is to write a test file that calls the function:

```
map(['2017', '2018', '2019'], v => Number(v));
```

This will do some blunt error checking: if your declaration of `map` only listed a single parameter, this would catch the mistake. But does it feel like something is missing here?

The equivalent of this style of test for runtime behavior might look something like this:

```
test('square a number', () => {
  square(1);
  square(2);
});
```

Sure, this tests that the `square` function doesn't throw an error. But it's missing any checks on the return value, so there's no real test of the behavior. An incorrect implementation of `square` would still pass this test.

This approach is common in testing type declaration files because it's simple to copy over existing unit tests for a library. And while it does provide some value, it would be much better to actually check some types!

One way is to assign the result to a variable with a specific type:

```
const lengths: number[] = map(['john', 'paul'], name => name.length);
```

This is exactly the sort of superfluous type declaration that Item 19 would encourage you to remove. But here it plays an essential role: it provides some confidence that the `map` declaration is at least doing something sensible with the types. And indeed you can find many type declarations in DefinitelyTyped that use exactly this approach for testing. But, as we'll see, there are a few fundamental problems with using assignment for testing.

One is that you have to create a named variable that is likely to be unused. This adds boilerplate, but also means that you'll have to disable some forms of linting.

A common workaround is to define a helper:

```
function assertType<T>(x: T) {}

assertType<number[]>(map(['john', 'paul'], name => name.length));
```

This eliminates the unused variable issue, but there are still surprises.

A second issue is that we're checking *assignability* of the two types rather than equality. Often this works as you'd expect. For example:

```
const n = 12;
assertType<number>(n);  // OK
```

If you inspect the `n` symbol, you'll see that its type is actually `12`, a numeric literal type. This is a subtype of `number` and so the assignability check passes, just as you'd expect.

So far so good. But things get murkier when you start checking the types of objects:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
assertType<{name: string}[]>(
  map(beatles, name => ({
    name,
    inYellowSubmarine: name === 'ringo'
  })));  // OK
```

The `map` call returns an array of `{name: string, inYellowSubmarine: boolean}` objects. This is assignable to `{name: string}[]`, sure, but shouldn't we be forced to acknowledge the yellow submarine? Depending on the context you may or may not really want to check for type equality.

If your function returns another function, you may be surprised at what's considered assignable:

```
const add = (a: number, b: number) => a + b;
assertType<(a: number, b: number) => number>(add);  // OK

const double = (x: number) => 2 * x;
assertType<(a: number, b: number) => number>(double);  // OK!?
```

Are you surprised that the second assertion succeeds? The reason is that a function in TypeScript is assignable to a function type, which takes fewer parameters:

```
const g: (x: string) => any = () => 12;  // OK
```

This reflects the fact that it's perfectly fine to call a JavaScript function with more parameters than it's declared to take. TypeScript chooses to model this behavior rather than bar it, largely because it is pervasive in callbacks. The callback in the Lodash `map` function, for example, takes up to three parameters:

```
map(array, (name, index, array) => { /* ... */ });
```

While all three are available, it's very common to use only one or sometimes two, as we have so far in this item. In fact, it's quite rare to use all three. By disallowing this assignment, TypeScript would report errors in an enormous amount of JavaScript code.

So what can you do? You could break apart the function type and test its pieces using the generic `Parameters` and `ReturnType` types:

```
const double = (x: number) => 2 * x;
let p: Parameters<typeof double> = null!;
assertType<[number, number]>(p);
//                           ~ Argument of type '[number]' is not
//                             assignable to parameter of type [number, number]
let r: ReturnType<typeof double> = null!;
assertType<number>(r);  // OK
```

But if "this" isn't complicated enough, there's another issue: `map` sets the value of `this` for its callback. TypeScript can model this behavior (see Item 49), so your type declaration should do so. And you should test it. How can we do that?

Our tests of `map` so far have been a bit black box in style: we've run an array and function through `map` and tested the type of the result, but we haven't tested the details of the intermediate steps. We can do so by filling out the callback function and verifying the types of its parameters and `this` directly:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
assertType<number[]>(map(
  beatles,
  function(name, i, array) {
// ~~~~~~~ Argument of type '(name: any, i: any, array: any) => any' is
//         not assignable to parameter of type '(u: string) => any'
    assertType<string>(name);
    assertType<number>(i);
    assertType<string[]>(array);
    assertType<string[]>(this);
                      // ~~~~ 'this' implicitly has type 'any'
    return name.length;
  }
));
```

This surfaced a few issues with our declaration of `map`. Note the use of a non-arrow function so that we could test the type of `this`.

Here is a declaration that passes the checks:

```
declare function map<U, V>(
  array: U[],
  fn: (this: U[], u: U, i: number, array: U[]) => V
): V[];
```

There remains a final issue, however, and it is a major one. Here's a complete type declaration file for our module that will pass even the most stringent tests for `map` but is worse than useless:

```
declare module 'overbar';
```

This assigns an `any` type to the *entire module*. Your tests will all pass, but you won't have any type safety. What's worse, every call to a function in this module will quietly produce an `any` type, contagiously destroying type safety throughout your code. Even with `noImplicitAny`, you can still get `any` types through type declarations.

Barring some advanced trickery, it's quite difficult to detect an `any` type from within the type system. This is why the preferred method for testing type declarations is to use a tool that operates *outside* the type checker.

For type declarations in the DefinitelyTyped repository, this tool is `dtslint`. It operates through specially formatted comments. Here's how you might write the last test for the `map` function using `dtslint`:

```
const beatles = ['john', 'paul', 'george', 'ringo'];
map(beatles, function(
  name,  // $ExpectType string
  i,     // $ExpectType number
  array  // $ExpectType string[]
) {
  this   // $ExpectType string[]
  return name.length;
});  // $ExpectType number[]
```

Rather than checking assignability, `dtslint` inspects the type of each symbol and does a textual comparison. This matches how you'd manually test the type declarations in your editor: `dtslint` essentially automates this process. This approach does have some drawbacks: `number|string` and `string|number` are textually different but the same type. But so are `string` and `any`, despite being assignable to each other, which is really the point.

Testing type declarations is tricky business. You *should* test them. But be aware of the pitfalls of some of the common techniques and consider using a tool like `dtslint` to avoid them.

Things to Remember

- When testing types, be aware of the difference between equality and assignability, particularly for function types.

- For functions that use callbacks, test the inferred types of the callback parameters. Don't forget to test the type of `this` if it's part of your API.
- Be wary of `any` in tests involving types. Consider using a tool like `dtslint` for stricter, less error-prone checking.