Appendix D. Recipes for Writing Declaration Files for Third-Party JavaScript Modules

This appendix covers a few key building blocks and patterns that come up over and over again when typing third-party modules. For a deeper discussion of typing third-party code, head over to "JavaScript That Doesn't Have Type Declarations on DefinitelyTyped".

Since module declaration files have to live in *.d.ts* files and so can't contain values, when you declare module types you need to use the `declare` keyword to affirm that values of the given type really are exported by your module. Table D-1 provides a short summary of regular declarations and their type declaration equivalents.

Table D-1. TypeScript and its type-only equivalents

.ts	*.d.ts*
`var a = 1`	`declare var a: number`
`let a = 1`	`declare let a: number`
`const a = 1`	`declare const a: 1`
`function a(b) { return b.toFixed() }`	`declare function a(b: number): string`
`class A { b() { return 3 } }`	`declare class A { b(): number }`
`namespace A {}`	`declare namespace A {}`
`type A = number`	`type A = number`
`interface A { b?: string }`	`interface A { b?: string }`

Types of Exports

Whether your module uses global, ES2015, or CommonJS exports will affect how you write your declaration files.

Globals

If your module only assigns values to the global namespace and doesn't actually export anything, you can just create a script-mode file (see "Module Mode Versus Script Mode") and prefix your variable, function, and class declarations with `declare` (every other kind of declaration— `enum`, `type`, and so on—remains unchanged):

```
// Global variable
declare let someGlobal: GlobalType

// Global class
declare class GlobalClass {}
```

```
// Global function
declare function globalFunction(): string

// Global enum
enum GlobalEnum {A, B, C}

// Global namespace
namespace GlobalNamespace {}

// Global type alias
type GlobalType = number

// Global interface
interface GlobalInterface {}
```

Each of these declarations will be globally available to every file in your project without requiring an explicit import. Here, you could use `someGlobal` in any file in your project without importing it first, but at runtime, `someGlobal` would need to be assigned to the global namespace (`window` in browsers or `global` in NodeJS).

Be careful to avoid `import` s and `export` s in your declaration file in order to keep your file in script mode.

ES2015 Exports

If your module uses ES2015 exports—that is, the `export` keyword—simply replace `declare` (which affirms that a global variable is defined) with `export` (which affirms that an ES2015 binding is exported):

```
// Default export
declare let defaultExport: SomeType
export default defaultExport

// Named export
export class SomeExport {
  a: SomeOtherType
}

// Class export
export class ExportedClass {}

// Function export
```

```
export function exportedFunction(): string

// Enum export
enum ExportedEnum {A, B, C}

// Namespace export
export namespace SomeNamespace {
  let someNamespacedExport: number
}

// Type export
export type SomeType = {
  a: number
}

// Interface export
export interface SomeOtherType {
  b: string
}
```

CommonJS Exports

CommonJS was the de facto module standard before ES2015, and is still the standard for NodeJS at the time of writing. It also uses the `export` keyword, but the syntax is a bit different:

```
declare let defaultExport: SomeType
export = defaultExport
```

Notice how we assigned our exports to `export`, rather than using `export` as a modifier (like we do for ES2015 exports).

A type declaration for a third-party CommonJS module can contain exactly one export. To export multiple things, we take advantage of declaration merging (see Appendix C).

For example, to type multiple exports and no default export, we export a single `namespace`:

```
declare namespace MyNamedExports {
  export let someExport: SomeType
  export type SomeType = number
```

```
  export class OtherExport {
    otherType: string
  }
}
export = MyNamedExports
```

What about a CommonJS module that has both a default export and named exports? We take advantage of declaration merging:

```
declare namespace MyExports {
  export let someExport: SomeType
  export type SomeType = number
}
declare function MyExports(a: number): string
export = MyExports
```

UMD Exports

Typing a UMD module is nearly identical to typing an ES2015 module. The only difference is that if you want to make your module globally available to script-mode files (see "Module Mode Versus Script Mode"), you use the special `export as namespace` syntax. For example:

```
// Default export
declare let defaultExport: SomeType
export default defaultExport

// Named export
export class SomeExport {
  a: SomeType
}

// Type export
export type SomeType = {
  a: number
}

export as namespace MyModule
```

Notice that last line—if you have a script-mode file in your project, you can now use that module directly (without importing it first) on the global `MyModule` namespace:

```
let a = new MyModule.SomeExport
```

Extending a Module

Extending a module's type declaration is less common than typing a module, but it might come up if you write a JQuery plugin or a Lodash mixin. Try to avoid doing it when possible; instead, consider using a separate module. That is, instead of a Lodash mixin use a regular function, and instead of a JQuery plugin—wait, why are you still using JQuery?

Globals

If you want to extend another module's global namespace or interface, just create a script-mode file (see "Module Mode Versus Script Mode"), and augment it. Note that this only works for interfaces and namespaces because TypeScript will take care of merging them for you.

For example, let's add an awesome new `marquee` method to JQuery. We'll start by installing `jquery` itself:

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

We'll then create a new file—say *jquery-extensions.d.ts*—in our project, and add `marquee` to JQuery's global `JQuery` interface (I found that JQuery defines its methods on the `JQuery` interface by sleuthing through its type declarations):

```
interface JQuery {
  marquee(speed: number): JQuery<HTMLElement>
}
```

Now, in any file where we use JQuery, we can use `marquee` (of course, we'll want to add a runtime implementation for `marquee` too):

```
import $ from 'jquery'
$(myElement).marquee(3)
```

Note that this is the same technique we used to extend built-in globals in "Safely Extending the Prototype".

Modules

Extending module exports is a bit trickier, and has more pitfalls: you need to type your extension correctly, load your modules in the correct order at runtime, and make sure to update your extension's types when the structure of the type declarations for the module you're extending changes.

As an example, let's type a new export for React. We'll start by installing React and its type declarations:

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

Then we'll take advantage of module merging (see "Declaration Merging") and simply declare a module with the same name as our React module:

```
import {ReactNode} from 'react'

declare module 'react' {
  export function inspect(element: ReactNode): void
}
```

Note that unlike in our example for extending globals, it doesn't matter whether our extension file is in module mode or script mode.

What about extending a specific export from a module? Inspired by ReasonReact, let's say we want to add a built-in reducer for our React components (a reducer is a way to declare an explicit set of state transitions for a React component). At the time of writing, React's type declarations declare the `React.Component` type as an interface and a class that get merged together into a single UMD export:

```
export = React
export as namespace React

declare namespace React {
  interface Component<P = {}, S = {}, SS = any>
```

```
    extends ComponentLifecycle<P, S, SS> {}
  class Component<P, S> {
    constructor(props: Readonly<P>)
    // ...
  }
  // ...
}
```

Let's extend `Component` with our `reducer` method. We can do this by entering the following in a *react-extensions.d.ts* file in the project root:

```
import 'react'                          ❶


declare module 'react' {                ❷

  interface Component<P, S> {           ❸

    reducer(action: object, state: S): S
                                        ❹

  }
}
```

❶ We import `'react'`, switching our extension file into script mode, which we need to be in to consume a React module. Note that there are other ways we could have switched to script mode, like importing something else, exporting something, or exporting an empty object (`export {}`)—we didn't have to import `'react'` specifically.

❷ We declare the `'react'` module, indicating to TypeScript that we want to declare types for that specific `import` path. Because we already installed `@types/react` (which defines an export for the same exact `'react'` path), TypeScript will merge this module declaration with the one provided by `@types/react`.

❸ We augment the `Component` interface provided by React by declaring our own `Component` interface. Following the rules of interface merging ("Declaration Merging"), we have to use the same exact signature in our declaration as the one in `@types/react`.

❹ Finally, we declare our `reducer` method.

After declaring these types (and assuming we've implemented the runtime behavior to support this update somewhere), we can now declare React components with built-in `reducers` in a typesafe way:

```
import * as React from 'react'

type Props = {
  // ...
}

type State = {
  count: number
  item: string
}

type Action =
  | {type: 'SET_ITEM', value: string}
  | {type: 'INCREMENT_COUNT'}
  | {type: 'DECREMENT_COUNT'}

class ShoppingBasket extends React.Component<Props, Sta
  reducer(action: Action, state: State): State {
    switch (action.type) {
      case 'SET_ITEM':
        return {...state, item: action.value}
      case 'INCREMENT_COUNT':
        return {...state, count: state.count + 1}
      case 'DECREMENT_COUNT':
        return {...state, count: state.count - 1}
    }
  }
}
```

As noted at the start of this section, it's good practice to avoid this pattern when possible (even though it's cool) because it can make your modules brittle and dependent on load order. Instead, try to use composition so that your module extensions consume the module they're extending, and export a wrapper rather than modifying that module.