# Chapter 11. Interoperating with JavaScript

We don’t live in a perfect world. Your coffee can be too hot and burn your mouth a little when you drink it, your parents might call and leave you voicemails a little too often, that pothole by your driveway is still there no matter how many times you call the city, and your code might not be completely covered with static types.

Most of us are in this boat: though once in a while you’ll have the leeway to start a greenfield project in TypeScript, most of the time it will start as a little island of safety, embedded in a larger, less safe codebase. Maybe you have a well-isolated component that you want to try TypeScript on even though your company uses regular ES6 JavaScript everywhere else, or maybe you’re fed up with getting paged at 6 A.M. because you refactored some code and forgot to update a call site (it’s now 7 A.M., and you’re ninja-merging TSC into your codebase before your coworkers wake up). Either way, you will probably start with an island of TypeScript in a type-less sea.

So far in this book I’ve been teaching you to write TypeScript the right way. This chapter is about writing TypeScript the practical way, in real codebases that are in the process of migrating away from untyped languages, that use third-party JavaScript libraries, that at times sacrifice type safety for a quick hot patch to unbreak prod. This chapter is dedicated to working with JavaScript. We’ll explore:

- Using type declarations
- Gradually migrating from JavaScript to TypeScript
- Using third-party JavaScript and TypeScript

## Type Declarations

A *type declaration* is a file with the extension *.d.ts*. Along with JSDoc annotations (see “Step 2b: Add JSDoc Annotations (Optional)”), it’s a way to attach TypeScript types to JavaScript code that would otherwise be untyped.

Type declarations have a similar syntax to regular TypeScript, with a few differences:

- Type declarations can only contain types, and can't contain values. That means no function, class, object, or variable implementations, and no default values for parameters.
- While type declarations can't define values, they can declare that there *exists* a value defined somewhere in your JavaScript. We use the special `declare` keyword for this.
- Type declarations only declare types for things that are visible to consumers. We don't include things like types that aren't exported, or types for local variables inside of function bodies.

Let's jump into an example, and take a look at a piece of TypeScript (*.ts*) code and its equivalent type declaration (*.d.ts*). This example is a fairly involved piece of code from the popular RxJS library; feel free to gloss over the details of what it's doing, and instead pay attention to which language features it's using (imports, classes, interfaces, class fields, function overloads, and so on):

```
import {Subscriber} from './Subscriber'
import {Subscription} from './Subscription'
import {PartialObserver, Subscribable, TeardownLogic} f

export class Observable<T> implements Subscribable<T> {
  public _isScalar: boolean = false
  constructor(
    subscribe?: (
      this: Observable<T>,
      subscriber: Subscriber<T>
    ) => TeardownLogic
  ) {
    if (subscribe) {
      this._subscribe = subscribe
    }
  }
  static create<T>(subscribe?: (subscriber: Subscriber<
    return new Observable<T>(subscribe)
  }
  subscribe(observer?: PartialObserver<T>): Subscriptic
  subscribe(
    next?: (value: T) => void,
    error?: (error: any) => void,
    complete?: () => void
  ): Subscription
  subscribe(
    observerOrNext?: PartialObserver<T> | ((value: T) =
    error?: (error: any) => void,
    complete?: () => void
  ): Subscription {
    // ...
  }
}
```

Running this code through TSC with the `declarations` flag enabled ( `tsc -d Observable.ts` ) yields the following *Observable.d.ts* type declaration:

```
import {Subscriber} from './Subscriber'
import {Subscription} from './Subscription'
import {PartialObserver, Subscribable, TeardownLogic} f

export declare class Observable<T> implements Subscriba
                                   ❶

  _isScalar: boolean
  constructor(
    subscribe?: (
      this: Observable<T>,
      subscriber: Subscriber<T>
    ) => TeardownLogic
  );
  static create<T>(
    subscribe?: (subscriber: Subscriber<T>) => Teardowr
  ): Observable<T>
  subscribe(observer?: PartialObserver<T>): Subscriptic
  subscribe(
    next?: (value: T) => void,
    error?: (error: any) => void,
    complete?: () => void
  ): Subscription
                                   ❷

}
```

❶ Notice the `declare` keyword before `class` . We can't actually define a class in a type declaration, but we can *declare* that we defined a class in the *.d.ts* file's corresponding JavaScript file. Think of

`declare` like an affirmation: “I swear that my JavaScript exports a class of this type.”
Because type declarations don’t contain implementations, we only keep the two overloads for `subscribe`, and not the signature for its implementation.

Notice how *Observable.d.ts* is just *Observable.ts*, minus the implementations. In other words, it’s just the types from *Observable.ts*.

This type declaration isn’t useful to other files in the RxJS library that use *Observable.ts*, since they have access to the *Observable.ts* source TypeScript file itself and can use it directly. It is useful, however, if you consume RxJS in your TypeScript application.

Think about it: if the authors of RxJS wanted to package in typing information on NPM for their TypeScript-wielding users (RxJS can be used in both TypeScript and JavaScript applications), they would have two options: package *both* source TypeScript files (for TypeScript users) and compiled JavaScript files (for JavaScript users), or ship compiled JavaScript files along with type declarations for TypeScript users. The latter reduces the file size, and makes it unambiguous what the correct import to use is. It also helps keep compile times for your application fast, since your TSC instance doesn’t have to recompile RxJS every time you compile your own app (in fact, it’s the reason the optimization strategy we introduce in “Project References” works!).

Type declaration files have a few uses:

1. When someone else uses your compiled TypeScript from their TypeScript application, their TSC instance will look for *.d.ts* files corresponding to your generated JavaScript files. This tells TypeScript what the types are for your project.
2. Code editors with TypeScript support (like VSCode) will read these *.d.ts* files to give your users helpful type hints as they type, even if they don’t use TypeScript.
3. They speed up compile times significantly by avoiding unnecessarily re-compiling your TypeScript code.

A type declaration is a way to tell TypeScript, “There exists this thing that’s defined in JavaScript, and I’m going to describe it to you.” When we talk about type declarations, we often call them *ambient* in order to differentiate them from regular declarations that contain values; for example, an *ambient variable declaration* uses the `declare` keyword to declare that a variable is defined somewhere in JavaScript, while a regular nonambient variable declaration is a normal `let` or `const` declaration that declares a variable without using the `declare` keyword.

You can use type declarations for a few things:

- To tell TypeScript about a global variable that's defined in JavaScript somewhere. For example, if you polyfilled the `Promise` global or defined `process.env` in a browser environment, you might use an *ambient variable declaration* to give TypeScript a heads-up.
- To define a type that's globally available everywhere in your project, so to use it you don't have to import it first (we call this an ambient type declaration).
- To tell TypeScript about a third-party module that you installed with NPM (an *ambient module declaration*).

A type declaration, regardless of what you're using it for, has to live in a script-mode *.ts* or *.d.ts* file (recall our earlier discussion of script mode in "Module Mode Versus Script Mode"). By convention, we give our file a *.d.ts* extension if it has a corresponding *.js* file; otherwise, we use a *.ts* extension. It doesn't matter what you call the file—for example, I like to stick to a single top-level *types.ts* until it gets unwieldy—and a single type declaration file can contain as many type declarations as you want.

Finally, while top-level values in a type declaration file need the `declare` keyword (`declare let`, `declare function`, `declare class`, and so on), top-level types and interfaces do not.

With those ground rules out of the way, let's briefly look at some examples of each kind of type declaration.

## Ambient Variable Declarations

An ambient variable declaration is a way to tell TypeScript about a global variable that can be used in any *.ts* or *.d.ts* file in your project without explicitly importing it first.

Let's say you're running a NodeJS program in your browser, and the program checks `process.env.NODE_ENV` (which is either `"development"` or `"production"`) at some point. When you run the program, you get an ugly runtime error:

```
Uncaught ReferenceError: process is not defined.
```

You sleuth around Stack Overflow a bit, and realize that the quickest hack to get your program running is to polyfill `process.env.NODE_ENV` yourself and hardcode it. So you create a new file, *polyfills.ts*, and define a global `process.env`:

```
process = {
  env: {
    NODE_ENV: 'production'
  }
}
```

Of course, TypeScript then comes to the rescue, throwing a red squiggly at you to try to save you from the mistake you're clearly making by augmenting the `window` global:

```
Error TS2304: Cannot find name 'process'.
```

But in this case, TypeScript is being overprotective. You really do want to augment `window`, and you want to do it safely.

So what do you do? You pop open *polyfills.ts* in Vim (you see where this is going) and type:

```
declare let process: {
  env: {
    NODE_ENV: 'development' | 'production'
  }
}

process = {
  env: {
    NODE_ENV: 'production'
    }
  }
```

You're declaring to TypeScript that there's a global object `process` that has a single property `env`, that has a property `NODE_ENV`. Once you tell TypeScript about that, the red squiggly disappears and you can safely define your `process` global.

---

**TSC SETTING: LIB**

TypeScript comes with a set of type declarations for describing the JavaScript standard library that includes built-in JavaScript types, like `Array` and `Promise`, and methods on built-in types, like `''.toUpperCase`. It also includes global objects like `window` and `document` (in a browser environment), and `onmessage` (in a Web Worker environment).

You can pull in TypeScript's built-in type declarations using your *tsconfig.json*'s `lib` field. Jump ahead to "lib" for a deep dive into how to dial in your project's `lib` setting.

---

## Ambient Type Declarations

Ambient type declarations follow the same rules as ambient variable declarations: the declaration has to live in a script-mode *.ts* or *.d.ts* file, and it'll be available globally to the other files in your project without an explicit import. For example, let's declare a global utility type `ToArray<T>` that lifts `T` to an array, if it isn't an array already. We can define this type in any script-mode file in our project—for this example, let's define it in a top-level *types.ts* file:

```
type ToArray<T> = T extends unknown[] ? T : T[]
```

We can now use this type from any project file, without an explicit import:

```
function toArray<T>(a: T): ToArray<T> {
  // ...
}
```

Consider using ambient type declarations to model data types that are used throughout your application. For example, you might use them to make the

UserID type we developed in “Simulating Nominal Types” globally available:

```
type UserID = string & {readonly brand: unique symbol}
```

Now, you can use UserID anywhere in your application without having to explicitly import it first.

## Ambient Module Declarations

When you consume a JavaScript module and want to quickly declare some types for it so you can use it safely—without having to contribute the type declarations back to the JavaScript module’s GitHub repository or DefinitelyTyped first—ambient module declarations are the tool to use.

An ambient module declaration is a regular type declaration, surrounded by a special `declare module` syntax:

```
declare module 'module-name' {
  export type MyType = number
  export type MyDefaultType = {a: string}
  export let myExport: MyType
  let myDefaultExport: MyDefaultType
  export default myDefaultExport
}
```

A module name ( `'module-name'` in this example) corresponds to an exact `import` path. When you import that path, your ambient module declaration tells TypeScript what’s available:

```
import ModuleName from 'module-name'
ModuleName.a  // string
```

If you have a nested module, make sure you include the whole `import` path in its declaration:

```
declare module '@most/core' {
  // Type declaration
}
```

If you just want to quickly tell TypeScript “I’m importing this module—I’ll type it later, just assume it’s an `any` for now,” keep the header but omit the actual declaration:

```
// Declare a module that can be imported, where each of
declare module 'unsafe-module-name'
```

Now if you consume this module, it’s less safe:

```
import {x} from 'unsafe-module-name'
x  // any
```

Module declarations support wildcard imports, so you can give a type to any `import` path that matches the given pattern. Use a wildcard ( `*` ) to match an `import` path:[1]

```
// Type JSON files imported with Webpack's json-loader
declare module 'json!*' {
  let value: object
  export default value
}

// Type CSS files imported with Webpack's style-loader
declare module '*.css' {
  let css: CSSRuleList
  export default css
}
```

Now, you can load JSON and CSS files:

```
import a from 'json!myFile'
a  // object

import b from './widget.css'
b  // CSSRuleList
```

**NOTE**

For the last two examples to work, you'll need to configure your build system to load *.json* and *.css* files. You can declare to TypeScript that these path patterns are safe to import, but TypeScript won't be able to build them by itself.

Jump ahead to "JavaScript That Doesn't Have Type Declarations on DefinitelyTyped" for an example of how to use ambient module declarations to declare types for untyped third-party JavaScript.

## Gradually Migrating from JavaScript to TypeScript

TypeScript was designed with JavaScript interoperability in mind, not as an afterthought. So while it's not painless, migrating to TypeScript is a good experience, letting you convert your codebase over a file at a time, opting into stricter levels of safety as you migrate, showing your boss and your coworkers just how impactful statically typing your code can be, one commit at a time.

At a high level, here's where you want to end up: your codebase should be completely written in TypeScript with strict type coverage, and third-party JavaScript libraries you depend on should come with high-quality, strict types of their own. Any bugs that could be caught at compile time are, and TypeScript's rich autocompletion halves the time it takes to write each line of code. You might have to take a few baby steps to get there:

- Add TSC to your project.
- Start typechecking your existing JavaScript code.
- Migrate your JavaScript code to TypeScript, a file at a time.
- Install type declarations for your dependencies, either stubbing out types for dependencies that don't have types or writing type declarations for untyped dependencies and contributing them back to DefinitelyTyped.[2]
- Flip on `strict` mode for your codebase.

This process can take a while, but you will see safety and productivity gains right away, and uncover more gains as you keep going. Let's walk through the steps one at a time.

## Step 1: Add TSC

When working on a codebase that combines TypeScript and JavaScript, start by letting TSC compile JavaScript files alongside your TypeScript. In your *tsconfig.json*:

```
{
  "compilerOptions": {
  "allowJs": true
}
```

With this one change, you can now use TSC to compile your JavaScript. Just add TSC to your build process, and either run every existing JavaScript file through TSC,[3] or continue running legacy JavaScript files through your existing build process and run new TypeScript files through TSC.

With `allowJs` set to `true`, TypeScript won't typecheck your existing JavaScript code, but it will transpile it (to ES3, ES5, or whatever `target` is set to in your *tsconfig.json*) using the module system you asked for (in your *tsconfig.json*'s `module` field). First step, done. Commit it, and give yourself a pat on the back—your codebase now uses TypeScript!

## Step 2a: Enable Typechecking for JavaScript (Optional)

Now that TSC is processing your JavaScript, why not typecheck it too? While you might not have explicit type annotations in your JavaScript, remember how great TypeScript is at inferring types for you; it can infer types in your JavaScript the same way it does in your TypeScript code. Enable this in your *tsconfig.json*:

```
{
  "compilerOptions": {
  "allowJs": true,
  "checkJs": true
}
```

Now, whenever TypeScript compiles a JavaScript file it'll do its best to infer types and typecheck as it goes, like it does for regular TypeScript code.

If your codebase is big and flipping on `checkJs` reports too many type errors at once, turn it off, and instead enable checking for a JavaScript file at a time by adding the `// @ts-check` directive (a regular comment at the top of the file). Or, if a few big files are throwing the bulk of your errors and you don't want to fix them just yet, keep `checkJs` on and add the `// @ts-nocheck` directive to just those files.

---

**NOTE**

Because TypeScript can't infer everything (e.g., function parameter types), it will infer a lot of types in your JavaScript code as `any`. If you have `strict` mode enabled in your *tsconfig.json* (you should!), you may want to temporarily allow implicit `any` s while you migrate. In your *tsconfig.json*, add:

```
{
  "compilerOptions": {
  "allowJs": true,
  "checkJs": true,
  "noImplicitAny": false
}
```

Don't forget to turn `noImplicitAny` on again when you've migrated a critical mass of code to TypeScript! It will probably reveal a bunch of real errors that you missed (unless, of course, you're Xenithar, disciple of Bathmorda the JavaScript witch, able to typecheck in your mind's eye with the help of nothing but a cauldronful of mugwort).

---

When TypeScript runs over JavaScript code, it uses a more lenient inference algorithm than it does for TypeScript code. Specifically:

- All function parameters are optional.
- The types of properties on functions and classes are inferred from usage (rather than having to be declared up front):

```
class A {
  x = 0 // number | string | string[], inferred from u
  method() {
    this.x = 'foo'
  }
  otherMethod() {
    this.x = ['array', 'of', 'strings']
    }
  }
```

- After declaring an object, class, or function, you can assign extra properties to it. Under the hood, TypeScript does this by generating a corresponding namespace for each class and function declaration, and automatically adding an index signature to every object literal.

## Step 2b: Add JSDoc Annotations (Optional)

Maybe you're in a hurry, and just need to add a single type annotation for a new function you added to an old JavaScript file. Until you get a chance to convert that file to TypeScript, you can use a JSDoc annotation to type your new function.

You've probably seen JSDoc before; it's those funny-looking comments above some JavaScript and TypeScript code with `@`-annotations like `@param`, `@returns`, and so on. TypeScript understands JSDoc, and uses it as input to its typechecker the same way that it uses explicit type annotations in TypeScript code.

Let's say you have a 3,000-line utilities file (yes, I know your "friend" wrote it). You add a new utility function to it:

```
export function toPascalCase(word) {
  return word.replace(
    /\w+/g,
    ([a, ...b]) => a.toUpperCase() + b.join('').toLower
  )
}
```

Without converting *utils.js* to TypeScript full sail—which would probably catch a bunch of bugs you'd then have to fix—you can annotate just your `toPascalCase` function, carving out a little island of safety in a sea of untyped JavaScript:

```
/**
 * @param word {string} An input string to convert
 * @returns {string} The string in PascalCase
 */
export function toPascalCase(word) {
  return word.replace(
    /\w+/g,
    ([a, ...b]) => a.toUpperCase() + b.join('').toLower
  )
}
```

Without that JSDoc annotation, TypeScript would have inferred `toPascalCase`'s type as `(word: any) => string`. Now, when TypeScript compiles your code it knows `toPascalCase`'s type is `(word: string) => string`. And you got some nice documentation out of it!

Head over to the TypeScript Wiki to learn more about supported JSDoc annotations.

## Step 3: Rename Your Files to .ts

Once you've added TSC to your build process, and optionally started typechecking and annotating JavaScript where possible, it's time to start switching over to TypeScript.

One file at a time, update your files' extensions from *.js* (or *.coffee*, *.es6*, etc.) to *.ts*. As soon as you rename a file in your code editor, you'll see your friends the red squigglies appear (the TypeError, not the kids' TV show), uncovering type errors, missed cases, forgotten `null` checks, and misspelled variable names. There are two strategies for fixing these errors:

1. Do it right. Take your time to type shapes, fields, and functions correctly, so you can catch errors in all the files that consume them. If you have `checkJs` enabled, turn on `noImplicitAny` in your *tsconfig.json* to uncover `any` s and type them, then turn it back off to make the output of typechecking your remaining JavaScript files less noisy.
2. Do it fast. Mass-rename your JavaScript files to the *.ts* extension, and keep your *tsconfig.json* settings lax (meaning `strict` set to `false`) to throw as few type errors as possible after renaming. Type complex types as `any` to appease the typechecker. Fix whatever type errors remain, and commit. Once this is done, flip on the `strict` mode flags ( `noImplicitAny`, `noImplicitThis`, `strictNullChecks`, and so on) one by one, and fix the errors that pop up. (See Appendix F for a full list of these flags.)

TIP

If you choose to go the quick-and-dirty route, a useful trick is to define an ambient type declaration `TODO` as a type alias for `any`, and use that instead of `any` so that you can more easily find and track missing types. You can also call it something more specific, so it's easier to find in a project-wide code search:

```
// globals.ts
type TODO_FROM_JS_TO_TS_MIGRATION = any

// MyMigratedUtil.ts
export function mergeWidgets(
  widget1: TODO_FROM_JS_TO_TS_MIGRATION,
  widget2: TODO_FROM_JS_TO_TS_MIGRATION
): number {
  // ...
}
```

Both ways of doing it are fair game, and it's up to you which you want to go with. Because TypeScript is a gradually typed language, it's built from the ground up to interoperate with untyped JavaScript code as safely as possible. Regardless of whether you're interoperating strictly typed TypeScript with untyped JavaScript or strictly typed TypeScript with loosely typed TypeScript, TypeScript will do its best to make sure that you're doing it as safely as possible, and that on the strictly typed island that you've so carefully built, everything is as safe as it can be.

## Step 4: Make It strict

Once you've migrated a critical mass of your JavaScript over to TypeScript, you'll want to make your code as safe as possible by opting into TSC's more stringent flags one by one (see Appendix F for a full list of flags).

Finally, you can disable TSC's JavaScript interoperability flags, enforcing that all of your code is written in strictly typed TypeScript:

```
{
  "compilerOptions": {
  "allowJs": false,
  "checkJs": false
}
```

This will surface the final rounds of type-related errors. Fix these, and you're left with a pristine, safe codebase that the most hardcore OCaml engineer would give you a pat on the back for (were you to ask nicely).

Following these steps will get you far when adding types to JavaScript you control, but what about JavaScript you don't control, like code you installed from NPM? To get there, we're first going to have to take a small detour…

## Type Lookup for JavaScript

When you import a JavaScript file from a TypeScript file, TypeScript follows an algorithm that looks like this to look up type declarations for your JavaScript code (remember that "file" and "module" are interchangeable when we talk about TypeScript):[4]

1. Look for a sibling *.d.ts* file with the same name as your *.js* file. If it exists, use it as the type declaration for the *.js* file.
   For example, say you have the following folder structure:

   ```
   my-app/
   ├──src/
   │ ├──index.ts
   │ └──legacy/
   │   ├──old-file.js
   │   └──old-file.d.ts
   ```

   You then import *old-file* from *index.ts*:

   ```
   // index.ts
   import './legacy/old-file'
   ```

   TypeScript will use *src/legacy/old-file.d.ts* as the source of type declarations for *./legacy/old-file*.
2. Otherwise, if `allowJs` and `checkJs` are true, infer the *.js* file's types (informed by any JSDoc annotations in the *.js* file).
3. Otherwise, treat the whole module as an `any`.

When importing a third-party JavaScript module—that is, an NPM package that you installed to *node modules*—TypeScript uses a slightly different algorithm:

1. Look for a local type declaration for the module. If it exists, use it.

   For example, say your app's folder structure looks like this:

   ```
   my-app/
   ├──node_modules/
   │ └──foo/
   ├──src/
   │ ├──index.ts
   │ └──types.d.ts
   ```

   And *types.d.ts* looks like this:

   ```
   // types.d.ts
   declare module 'foo' {
     let bar: {}
     export default bar
   }
   ```

   If you then import `foo` , TypeScript will use the ambient module declaration in *types.d.ts* as the source of types for `foo` :

   ```
   // index.ts
   import bar from 'foo'
   ```

2. Otherwise, look at the module's *package.json*. If it defines a field called `types` or `typings` , use the *.d.ts* file that field points to as the source of type declarations for the module.
3. Otherwise, traverse out a directory at a time, and look for a *node modules/@types* directory that has type declarations for the module.

   For example, say you installed React:

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

   ```
   my-app/
   ├──node_modules/
   │ ├──@types/
   ```

```
│ │ └──react/
│ └──react/
├──src/
│ └──index.ts
```

When you import React, TypeScript will find the *@types/react* folder and use that as the source of type declarations for React:

```
// index.ts
import * as React from 'react'
```

4. Otherwise, proceed to steps 1–3 of the local type lookup algorithm.

That was a lot of steps, but it's remarkably intuitive once you get the hang of it.

---

#### TSC SETTINGS: TYPES AND TYPEROOTS

By default, TypeScript looks in *node modules/@types* in your project's folder and containing folders (*../node modules/@types* and so on) for third-party type declarations. Most of the time, you want to leave this behavior as is.

To override this default behavior for global type declarations, configure `typeRoots` in your *tsconfig.json* with an array of folders to look in for type declarations. For example, you can tell TypeScript to look for type declarations in the *typings* folder as well as *node modules/@types*:

```
{
  "compilerOptions": {
    "typeRoots" : ["./typings", "./node modules/@types"]
  }
}
```

For even more granular control, use the `types` option in your *tsconfig.json* to specify which packages you want TypeScript to look up types for. For example, the following config ignores all third-party type declarations except the ones for React:

```
{
  "compilerOptions": {
    "types" : ["react"]
  }
}
```

---

# Using Third-Party JavaScript

**NOTE**

I'll assume you're using a package manager like NPM or Yarn to install third-party JavaScript. And if you're one of those people that prefers to copy and paste code manually instead—shame on you.

When you `npm install` third-party JavaScript code into your project, there are three possible scenarios:

1. The code you installed comes with type declarations out of the box.
2. The code you installed doesn't come with type declarations, but declarations are available on DefinitelyTyped.
3. The code you installed doesn't come with type declarations, and declarations are not available on DefinitelyTyped.

Let's dig into each of these.

## JavaScript That Comes with Type Declarations

You know that a package comes with type declarations out of the box if you `import` it with `{"noImplicitAny": true}` and TypeScript doesn't throw a red squiggly at you.

If the code you're installing is compiled from TypeScript, or its authors were kind enough to include type declarations in its NPM package, then you're in luck. Just install the code and start using it with full type support.

Some examples of NPM packages that come with built-in type declarations are:

```
npm install rxjs
npm install ava
npm install @angular/cli
```

WARNING

Unless the code you're installing was actually compiled from TypeScript, you always run the risk that the type declarations it comes with don't match up to the code those declarations describe. When type declarations come packaged with source code the risk of this happening is pretty low (especially for popular packages), but it's something to be aware of.

## JavaScript That Has Type Declarations on DefinitelyTyped

Even if the third-party code you're importing doesn't come with type declarations, declarations for it are probably available on DefinitelyTyped, TypeScript's community-maintained, centralized repository for ambient module declarations for open source projects.

To check if the package you installed has type declarations available on DefinitelyTyped, either search on TypeSearch or just try installing the declarations. All DefinitelyTyped type declarations are published to NPM under the `@types` scope, so you can just `npm install` from that scope:

```
npm install lodash --save              # Install Lodash
npm install @types/lodash --save-dev # Install type dec
```

Most of the time, you'll want to use `npm install`'s `--save-dev` flag to add your installed type declarations to your *package.json*'s `devDependencies` field.

NOTE

Since type declarations on DefinitelyTyped are community-maintained, they run the risk of being incomplete, inaccurate, or stale. While most popular packages have well-maintained type declarations, if you find that the declarations you're using can be improved, take the time to improve them and contribute them back to DefinitelyTyped so other TypeScript users can take advantage of your hard work.

## JavaScript That Doesn't Have Type Declarations on

## DefinitelyTyped

This is the least common case of the three. You have several options here, from the cheapest and least safe to the most time-intensive and safest:

1. *Whitelist the specific import* by adding a `// @ts-ignore` directive above your untyped import. TypeScript will let you use the untyped module, but the module and all of its contents will be typed as `any` :

   ```
   // @ts-ignore
   import Unsafe from 'untyped-module'

   Unsafe  // any
   ```

2. *Whitelist all usages of this module* by creating an empty type declaration file and stubbing out the module. For example, if you installed the rarely used package `nearby-ferret-alerter` , you could make a new type declaration (e.g., *types.d.ts*) and add to it the ambient type declaration:

   ```
   // types.d.ts
   declare module 'nearby-ferret-alerter'
   ```

   This tells TypeScript that there exists a module that you can import ( `import alert from 'nearby-ferret-alerter'` ), but it doesn't tell TypeScript anything about the types contained in that module. This approach is a slightly better alternative to the first, in that now there's a central *types.d.ts* file that enumerates all the untyped modules in your application, but it's equally unsafe because `nearby-ferret-alerter` and all of its exports will still be typed as `any` .
3. *Create an ambient module declaration*. Like in the previous approach, create a file called *types.d.ts* and add an empty declaration ( `declare module 'nearby-ferret-alerter'` ). Now, fill in the type declaration. For example, the result might look like this:

   ```
   // types.d.ts
   declare module 'nearby-ferret-alerter' {
     export default function alert(loudness: 'soft' | '
     export function getFerretCount(): Promise<number>
   }
   ```

   Now when you `import alert from 'nearby-ferret-alerter'`, TypeScript will know exactly what `alert`'s type is. It's no longer an `any`, but `(loudness: 'quiet' | 'loud') => Promise<void>`.

4. *Create a type declaration and contribute it back to NPM*. If you got as far as the third option and now have a local type declaration for your module, consider contributing it back to NPM so the next person that needs type declarations for the awesome `nearby-ferret-alerter` package can use it too. To do this you can either submit a pull request to the `nearby-ferret-alerter` Git repository and contribute the type declarations directly, or, if the maintainers of that repository don't want to be on the hook for maintaining TypeScript type declarations, contribute your declarations to DefinitelyTyped instead.

Writing type declarations for third-party JavaScript is straightforward, but how it's done depends on the type of module you're typing. There are a few common patterns that come up when typing different kinds of JavaScript modules (from NodeJS modules to jQuery augmentations and Lodash mixins to React and Angular components). Head over to Appendix D for a list of recipes for typing third-party JavaScript modules.

**NOTE**

Automatically generating type declarations for untyped JavaScript is an active area of research. Check out `dts-gen` for a way to automatically generate type declaration scaffolding for any third-party JavaScript module.

## Summary

There are a few ways to use JavaScript from TypeScript. Table 11-1 summarizes the options.

Table 11-1. Ways to use JavaScript from TypeScript

| Approach | tsconfig.json flags | Type safety |
|---|---|---|
| Import untyped JavaScript | `{"allowJs": true}` | Poor |
| Import and check JavaScript | `{"allowJs": true, "checkJs": true}` | OK |
| Import and check JSDoc-annotated JavaScript | `{"allowJs": true, "checkJs": true, "strict": true}` | Excellent |
| Import JavaScript with type declarations | `{"allowJs": false, "strict": true}` | Excellent |
| Import TypeScript | `{"allowJs": false, "strict": true}` | Excellent |

In this chapter we covered various aspects of using JavaScript and TypeScript together, from the different kinds of type declarations and how to use them, to migrating your existing JavaScript project to TypeScript piece by piece, to using third-party JavaScript safely (and unsafely). Interoperating with JavaScript can be one of the trickiest aspects of TypeScript; with all the tools at your disposal, you're now equipped to do it in your own project.

1 Wildcard matching with `*` follows the same rules as regular glob pattern matching.

2 DefinitelyTyped is the open source repository for JavaScript type declarations. Read on to learn more.

3 For really big projects it can be slow to run every single file through TSC. For a way to improve performance for large projects, see "Project References".

4 Strictly speaking, this is true for module-mode, but not script-mode, files. Read more in "Module Mode Versus Script Mode".