TYPESCRIPT JUMPSTART

FAST-TRACK TO TYPESCRIPT PROFICIENCY, FOR EXPERIENCED DEVELOPERS

ANGULAR UNIVERSITY

Table Of Contents

- Do we really need type annotations to get type-safety?
- Why Type safety does not mean more ceremony when coding
- The biggest advantage of Typescript
- How to make the most of Typescript Type Definitions
- What is @types, when should I use it and why?
- What happened to the typings executable and DefinitivelyTyped?
- Not all type definitions leverage completely the Typescript type system
- What are compiler opt-in types, when should I use them and why?
- Why do I sometimes get this 'duplicate type definition' error?
- Handling the gap between libraries and the compiler
- Guidelines for Using the multiple Type Definitions available
- When should we use compiler opt-in types?
- When should we use @types?
- What if no type definitions are available?
- How to make sure our programs leverage type safety effectively?

Section 4 - Conclusions & Bonus Content

- Final Thoughts
- Bonus Content - Typescript - A Video List

The Typescript Jumpstart Book

Introduction and Book Goals

Welcome to the Typescript Jumpstart Book, thank you for joining! Without further ado, let's get started: like the title says, the goal of the book is to get you proficient quickly in the Typescript language!

There is one fundamental assumption that this book makes, which is that you already have some experience with another object-oriented programming language such as for example the most common **statically-typed** languages:

- Java
- C#
- Scala

You could also be familiar with one of the the most popular **dynamically-typed** languages, such as for example:

- Javascript / ES6
- Ruby
- Python

Most likely you are familiar with a combination of several. Typescript brings the best of these two worlds into one single language, that is gaining popularity very quickly.

Google has recently announced that it will start using Typescript internally alongside Java, so Full stack development in Typescript using Node as a runtime will likely become mainstream in the next few years and fulfill the old Java dream:

> *Write Once, Run Anywhere - this is possible today with Typescript!*

If you are familiar with any of the languages mentioned above, then essentially you already know most of Typescript: it will look and feel extremely familiar.

In fact, any Javascript program is also a valid Typescript program! But this familiarity is very deceptive because the Typescript type system is at the same time:

- very similar to currently statically typed type systems like Java, C# or Scala

- but at the same time fundamentally different than those type systems, and designed for maximum compatibility with existing dynamically typed Javascript codebases

Due to this large similarity to things that you already know for years, going through a huge catalog of language features would not be a good use of your time and attention: it would just be repeating the official documentation, right?

Instead, to become proficient with Typescript we need to focus on the key differentiating new factors that the language brings to the table.

We want to focus on the fundamental aspects of the type system, and on the key language feature that gives Typescript its name: the Type Definitions.

And that, in a nutshell, is what we will be doing in this book. We will provide an answer to the following key questions:

- How does the type system really work under the hood?
- How does it combine the best of both dynamic and statically typed languages?
- In which way is this type system better and why is it designed in a certain way?
- What are the different types of type definitions, when to use each and why?

This small set of key concepts, together with your background in a previous language is all you really need for comfortably start writing Typescript programs.

Let's then start exploring the key concepts of the Typescript language, we will start at the beginning: the Typescript Type System.

To make the most out of this book, I invite you to try as we go along the examples on this next section using the Official Online Typescript Playground.

On the second section, we will be using the command line Typescript compiler which is based on the Node runtime.

The Typescript Type System

A key thing about the Typescript Type System is that most of the times it just works, but sometimes we get some surprising error messages that give us an indication that there is something fundamental about it that we might not be aware yet.

The great thing about Typescript, is that we could go for months using it without knowing some important concepts about what is going on with the type system.

We will find unexpected compiler error messages, but not to the point where we can't use Typescript and be productive with the language, because in general it just works.

But if we add these concepts to our toolbox, this will make our experience of the language much more enjoyable and productive.

We are going to break this down step by step into 3 key concepts.

A Simple Example - Why Doesn't This Work?

Let me give you a quick example of what we mean when we say that the type system is actually quite different than other type systems. Let's try to guess if this simple code example would compile our not:

```
1
2  let user = {};
3
4  user.name = 'John';
```

```
5
```

If we are not familiar with how the Typescript Type system works, we might be surprised to realize that this actually does not compile. So why is that? Let's have a look at the error message:

```
Error:(54, 6) TS2339:Property 'name' does not exist on
type '{}'.
```

So what is going here? We have defined an empty object first and then tried to assign it the name property. This is just plain Javascript code, and Typescript should allow us to write that transparently. So why the error message?

This is related to the first key concept that we are going to cover: Type Inference.

Key Concept 1 - Type Inference is Always On

The key thing to start understanding this error is to realize that Type inference is active here. So the `user` variable was automatically assigned a type even if we didn't add any explicit type annotation.

If we hover over the user variable, we can see the inferred type. In Webstorm, if we click the variable and hit Ctrl+Shift+P, we get the following inferred type:

```
type: {}
```

You might think at this point, what is this type?

You might have heard of the type Any and the compiler property `noImplicitAny`.

We can see that the Any type is not related to this situation, because the inferred type is not Any. So what is that type that was just inferred?

We are going to understand that by providing another example, have a look at this and try to guess if it compiles, and if not what is the error message:

```
1
2  let course = {
3      name: 'Components'
4  };
5
6  course.name = 'Components and Directives';
7
8  course.lessonCount = 20;
9
```

Again it might be a bit surprising that this code does not compile. Here is the error message:

```
Error:(59, 8) TS2339:Property 'lessonCount' does not exist
on type '{ name: string; }'.
```

And if we check what is the inferred type of the variable course, we get this type:

```
type: {name:string}
```

Let's break this down, so what is going on in this scenario?

- We can see that the variable `course` is not of type Any, it got a different type assigned
- The type inferred looks like it's the one of an object that has only one property named `name'?
- We can set new values to this property called `name`
- but we cannot assign any other variable to a variable of this type

Let's test this to see if its true

Let's see if this could be the case, that indeed a type was inferred with only one property. Let's define such type explicitly:

```
1
2  let course : {name:string} = {
3      name: 'Components'
4  };
5
6  course.name = 'Components and Directives';
7
8  course.lessonCount = 20;
9
```

As we can see, we have defined the type inline using a Type annotation. The result is that we get the same error message as above: we can overwrite `name` but we cannot set a new property `lessonsCount`.

This seems to confirm that there was a type inferred with only one property. What if we define this type not inline, but create a custom type? For example like this:

```
1
2  interface Course {
3      name:string;
4  }
5
6
7  let course : Course = {
8      name: 'Components'
9  };
10
11 course.name = 'Components and Directives';
12
13 course.lessonCount = 20;
14
```

Notice the use of the `interface` keyword for defining a custom object type, the `Course` type. In Typescript the interface keyword is not just an object oriented concept, it has been generalized to include objects also:

> *In Typescript, a custom object type can also implement an Interface!*

And this generalization of the notion of interface will make even more sense in a moment. Back to the code example just above, we also get the same error message as before:

```
Error:(59, 8) TS2339:Property 'lessonCount' does not exist
on type '{ name: string; }'.
```

Which in this scenario would make much more sense because we are defining the type explicitly and not via type inference.

So what does this all mean? This leads us to the Key Concept number 2 in the Typescript Type System.

Key Concept 2 - Types are defined by the collection of their properties

In the current version of Typescript, the type system is said to be based on structural subtyping. What does this mean?

It means that what defines a type is not so much its name (like nominal type systems that are common in other languages). Instead, what defines a type is a collection of specific properties and their types.

For example what defines the type of the `Course` custom type is its list of properties, and not its name.

Also, if a variable has no type annotation associated to it, Typescript will look into its collection of properties and infer a type on the fly which contains that particular set of properties.

So how does this explain the compiler errors?

That is why the type inferred in course is `type: {name:string}`. Because the object only has one property with that particular name.

And this is also why we get a compiler error while assigning `lessonCount` to the course object.

This is also why we can't assign a name to the user property: because the inferred type is `type {}`, which means that `user` is an object with no properties, an empty object essentially.

So we could only assign it to another empty object. And this leads us to the last key concept: Type Compatibility.

Key Concept 3 - Type compatibility depends on the list of properties of a type

As we have seen what really defines a type in Typescript is its list of properties:

> *So that same list of properties and their types is also what defines if two types are compatible!*

Have a look at this example where we define two types and assign them to each other:

```
1
2   interface Course {
3       name:string;
4       lessonCount:number;
5   }
6
7   interface Named {
8       name:string;
9   }
10
11  let named : Named = {
12      name: 'Name goes here'
13  };
14
15  let course: Course = {
16      name: 'Components and Directives',
17      lessonCount: 20
18  };
19
20
21  named = course;
22
23  course = named;
24
```

There is still a compilation error here. This line `named = course` does compile correctly, because `Course` has all the mandatory properties needed by `Name`, so this type assignment is valid.

Note that the `Course` interface does not need to extend `Named`, like in other type systems (nominal type systems).

But the line `course = named` does not compile, and we get the following error:

```
Error:(73, 1) TS2322:Type 'Named' is not assignable to
type 'Course'. Property 'lessonCount' is missing in type
'Named'.
```

So as we can see in the error message, the two types are not compatible because of a missing property, and not because the two types are different.

So How to we fix the compilation error?

Let's go back to the initial example and make it compile, as it's a very common case:

```
1
2  let user:any = {};
3
4  user.name = 'John';
5
```

By assigning the type Any to the user variable, we can now assign it any property we need, because that is how the Any type works. Another

thing about the Any type is that we could take the variable user and also assign it to anything.

So annotating a variable with type Any is essentially telling the compiler to bypass the type system, and in general not check type compatibility for this variable.

How To Define Optional Variables

Another way of fixing this type of errors is to mark variables as optional, for example by annotating variables with a question mark:

```
1
2  interface Course {
3      name:string;
4      lessonCount?:number;
5  }
6
7  interface Named {
8      name:string;
9  }
10
11 let named : Named = {
12     name: 'Name goes here'
13 };
14
15 let course: Course = {
16     name: 'Components and Directives',
17     lessonCount: 20
18 };
19
20
21 named = course;
22
23 course = named;
24
```

In this example, we have marked the `lessonCount` variable as optional by adding a question mark to the member variable declaration in `Course`. So now the line `course = named` also compiles, because `named` has all the mandatory properties of the `Course` custom type.

So as we can see at this point, although the Typescript Type System looks very familiar to developers coming from other languages at first, its actually designed in a fundamentally different way.

Why was the Typescript Type System designed like this?

The type inference mechanism and the type compatibility features of Typescript are very powerful and generally just work. We could actually code for a long time in Typescript without realizing what is going on under the hood except for some occasional error messages.

We can see why the Type system is built like this: its to allow as much as possible a style of coding that is almost identical to plain Javascript.

Everything is based on type inference as much as possible, although there are places like function arguments where we need to add type annotation if setting `noImplicityAny` to true, because there is no way for the compiler to infer those types.

The type system is built in a way that most of the error messages we get are actually errors that we would want to fix.

What is the tradeoff involved ?

But there is a small tradeoff involved to get all these type safety features which include: catching errors at compile time, refactoring and find

usages.

We will on occasion get an error for something that would just work in plain Javascript like the first scenario we saw in this section.

This does not happen very often, and when it happens it can be fixed using the Any type. It's better to try to use Any the least possible, to keep all the benefits of the type system intact.

The Typescript language is continuously evolving, its even in the works the possibility of adding nominal typing, have a look at this Github issue.

Also, another key feature is that in Typescript the type annotations are optional, and if we want to work with Javascript libraries that where not written in Typescript (which is the vast majority of libraries available), we will need to bring our own Type Definitions.

But there are many types of definitions to choose from, so in the second section of this book we will learn which ones to use in which situation.

We will also be introducing the command line Typescript compiler and a bit of the ecosystem around it.

Typescript Type Definitions

Typescript has been evolving very quickly, and one of the things that have evolved more is its most differentiating feature: the Typescript type definitions.

If you have been using Typescript with Angular recently or without it, you might have run into a couple of the following questions or errors situations:

- Does Typescript type safety necessarily mean more ceremony when writing code?
- What are the multiple types of Typescript Type Definitions?
- How do I use libraries that don't have Type Definitions available?
- What is the relation between Type Definitions and Npm?
- When to install third party types?
- How can packages provide their own custom types?
- What is `@types`, when should I use it and why?
- What happened to the `typings` executable and DefinitivelyTyped?
- What are compiler opt-in types, when should I use them and why?
- Why do I sometimes get a 'duplicate type definition' error, and how to fix it?
- Why does it look like Promise type definitions sometimes don't work correctly?
- Recommendations on to use Typescript type definitions effectively

We are going to be covering all of this in this section, I invite you to code along (from an empty folder) to get the most out of it.

What are the multiple scenarios for Typescript Type Definitions?

In Typescript 2 and beyond, when using a Javascript library there are now essentially 4 scenarios in what concerns type definitions:

- No type definitions of any kind are available
- Type definitions are available and shipped together with the compiler itself
- A library does not ship with type definitions, but they can be installed separately
- A library ships with its own type definitions built-in

So what is the difference? Let's start at the beginning: what if there are no type definitions at all? Because that is a very common case and will be so for years to come (if not forever).

So let's start with that: we have no guarantee that Javascript modules in the future will be systematically shipped with their own types, or that someone will write those types, publish and maintain them.

The larger and most used modules will likely have good type definitions, but what about smaller modules?

How do I use libraries that don't have Type Definitions available?

Let's start with a simple example, let's setup a node project in an empty folder and install a simple module named uuid, that generates unique identifiers.

```
npm init
.... hit enter to all questions

# install Typescript locally, inside node_modules
npm install --save-dev typescript

# setup a Typescript compiler configuration file tsconfig.json
./node_modules/.bin/tsc --init

# install uuid
npm install --save uuid

```

Please notice that here we have created an initial `package.json` with npm init, and have installed a local version of Typescript. If you open this folder with an IDE like for example Webstorm, the Typescript version inside node_modules will be taken and used automatically.

So you don't have to install Typescript globally, and its probably better to avoid to install it globally to avoid version confusions between projects, command line and IDE.

So now that we have installed `uuid`, how do we use it?

Using Javascript Libraries with No Type Definitions Available

First, let's check which version of Typescript we are using by doing `tsc -v`. What happens if we try to import the `uuid` library? Let's give it a try:

```
import * as uuid from 'uuid';

console.log(uuid());

```

OK, so what is going on in that import statement? Let's break it down:

- we are using the ES6 import syntax to import something from an ES6 module named `uuid`
- we are saying that the module will have a default export because we are using `*`
- We are assigning whatever that single export is and assigning it to a constant named `uuid`, but what type will it have?

And then we are using the `uuid`, which has implicitly taken the type `any` and we are using it to call it as a function. Let's try to run this to see what happens.

A simple way to run Typescript files

But this is a Typescript file, so we can't call node on it and run it. Do we need a complex build system for that? No, we could simply create an npm script task that calls the `tsc` compiler and then runs the output using node.

But let's keep the file system clean of generated files, let's instead using an utility called ts-node:

```
### install ts-node
npm install --save-dev ts-node

```

With `ts-node` installed, let's add a npm script task to run our test program above, which will be in a file called `test.ts`:

```
{
```

```
  "name": "types-examples",
  "scripts": {
    "demo": "./node_modules/.bin/ts-node ./test.ts"
  },
  "devDependencies": {
    "ts-node": "^2.0.0",
    "typescript": "^2.1.0"
  },
  "dependencies": {
    "uuid": "~3.0.1"
  }
}
```

OK, so now we can run the test program using a simple npm command:

```

npm run demo

```

What will the results be of such a simple program?

What is going on here is that there is a `uuid` module present, but `uuid` is not shipped with its own type definitions.

Since Typescript 2.1 or above, if we have a CommonJs module available inside node_modules that has no type definitions available, we will still be able to import it and use it.

But how can we use it as a function, what type is `uuid` then?

What happens is that this module is being imported and implicitly assigned to the Any Type.

How does the Any Type work?

The Any type allows up to essentially bypass the type-safety of the Typescript type system:

- we can use Any as a function and call it using parentheses like we did with `uuid`
- a variable of type Any is assumed to potentially have any property, like a plain Javascript object
- we can also take a variable of Type Any and assign it to essentially anything else (without getting an error)

What does the use of the Type Any mean?

This means that although our program compiles we are essentially back to writing plain Javascript with that library: we won't have reliable auto-completion or refactoring.

But this also means that any module in npm is available for being seamlessly used in a Typescript program!

So this is a great start. If anything else fails we simply write Javascript and it just works. But how can we improve this and get type safety for the most commonly used npm libraries?

What is the relation between Type Definitions and Npm?

Let's now cover modules that ship with their own types. There are more and more modules each day that get shipped in npm with their own type-definitions already built-in.

This means that the types are shipped directly inside the node module itself, and don't have to be installed separately.

Let's start with a example, let's for example install the Axios isomorphic HTTP library. If you don't know Axios, it's a great library for doing both Ajax on the client and HTTP calls on the server, while using the same Promise-based API:

```
npm install --save axios
```

This command will install Axios, so we could start using this client to query a REST API using plain Javascript. The good news is: we can also do the same thing in a type-safe way as well!

This is because Axios comes with its own type definitions built-in. Let's have a look:

```
cd node_modules/axios

bash-3.2$ ls -1 *.d.ts

index.d.ts
```

As we can see, the Axios node module comes bundled with its own Type definitions, so we don't need to install them separately! So let's see this in action in our program:

```
import axios from 'axios';
import {AxiosPromise} from "axios";

const response: AxiosPromise = axios.get('/lessons', {
    ...
```

```
7  });
8
```

The Axios library has a default export that we have imported and gave the name `axios`. This import has implicitly the type `AxiosStatic` as declared on the Axios type definition file.

Do we really need type annotations to get type-safety?

This import named `axios` is not of type Any, and we have auto-completion available as well as refactoring and find usages, all working out of the box.

More than that, do you see the `AxiosPromise` type annotation? It's actually redundant, if we delete it the type of the response the constant would still be inferred as being of type `AxiosPromise`, and we would have auto-completion working for that variable as well.

More than that, do you see the configuration object after the url? That is automatically inferred to be of type `AxiosRequestConfig`, so we have auto-completion to fill in the request parameters as well.

Type safety does not mean more ceremony when coding

So why don't we get a compilation error at this stage because the object is empty?

That is because the `AxiosRequestConfig` type definition only has optional properties. With our IDE we can jump into the definition of `AxiosRequestConfig`:

```
export interface AxiosRequestConfig {
  url?: string;
  method?: string;
  baseURL?: string;
  ....
}
```

As we can see, all the properties are marked as optional using a question mark. So this is a good example of how using a library with built-in type definitions does not mean more verbosity in our program, or getting constant compiler errors.

The biggest advantage of Typescript

With Typescript, we can mostly have both the convenience of plain Javascript plus the enhanced tooling. If we use libraries that provide their own built-in types, we can have auto-completion, refactoring and find usages almost everywhere in our program, at the expense of using just a few type annotations at strategic places.

The biggest exception for this will be function parameters, where there is no way for the compiler to infer what is the type of a function parameter. But it's a great idea to mention the types of our function parameters for documentation purposes.

How to make the most of Typescript type definitions

If you want to leverage Typescript type inference to its maximum and have it auto-detect the type of the largest amount possible of variables, the best way is to go to the `tsconfig.json` and set the `noImplicitAny` property to true:

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": true,
        "sourceMap": false
    }
}
```

This way, if by some reason the compiler can't infer the type of a variable, it will not implicitly assign it the type any. This is probably one of the most important properties available to configure the compiler.

> *We could almost have named it* `useTypeInferenceAsMuchAsPossible` *instead of* `noImplicitAny`

Now let's take the Axios type definitions, and see what we can do with them in terms of helping us to build a type-safe program.

Leveraging the Promise API in our programs

We would like to be able to define a method that does an HTTP call and returns a Promise of a given type:

```

import axios from 'axios';
import {AxiosPromise} from "axios";


interface Lesson {
    id:number;
    description: string;
}


```

```
function getLesson(lessonId:number): AxiosPromise<Lesson> {
    return axios.get(`lessons/${lessonId}`);
}


const promise  = getLesson(1);


promise.then(response => {
   ...
});


```

So what have we done in this small program? Let's break it down:

- we have created a custom object type called `Lesson`, with two mandatory properties
- we have defined a function `getLesson` that does an HTTP call and returns a Promise
- We are trying to specify what is the data returned by the promise via a generic parameter in the `AxiosResponse` type

So out goal here is to have type safety in the `then` clause, by knowing there implicitly that the data returned is `Lesson`, without having to use a type annotation.

Trying to use generics in promise return types

So what is the result? We currently get two compiler errors:

```
Error:(13, 38) TS2315: Type 'AxiosPromise' is not generic.
Error:(21, 14) TS7006: Parameter 'response' implicitly has
```

```
an 'any' type.
```

So what do these errors mean? Let's break it down:

- we can't specify the data returned by an Axios promise, via a generic parameter
- this means that the data returned is implicitly of type Any
- the promise response is also implicitly of type any, which throws an error

At this point, we can see that the Promise type definitions shipped with Axios although useful, would need some extra type annotations to ensure type safety on our program. For example:

```
1  function getLesson(lessonId:number) {
2      return axios.get(`lessons/${lessonId}`);
3  }
4
5  const promise  = getLesson(1);
6
7  promise.then(response => {
8
9      const lesson: Lesson = response.data;
10
11     ...
12
13 });
14
```

And this would work great, have a look at line 9 we added an explicit type annotation.

But at this point we could think that Promise is a standard ES6 API, so why not use that instead? We would benefit both from a standard API

and type inference in our program.

So if this would be a node program, how could we do that?

Writing Node programs using the standard Promise API

What we need to do is to use another library that does not necessarily ship with its own promise type definitions, or in this case that returns Promise-like types that are compatible with ES6 promises.

Let's for example set up `request-promise`, which is a promise enabler for the popular `request` node HTTP client:

```
npm install --save request
npm install --save request-promise
```

So how can we use this client to write type-safe node programs, using the standard promise API?

Using Node require in Typescript programs

The way that `request-promise` works is that it has the same API as `request`, but it returns promises. So how can we use it? We could start by using it as a plain node module, by requiring it like this:

```
const rp = require('request-promise');
```

But at this point we would get an error:

```
Error:(3, 12) TS2304:Cannot find name 'require'.
```

What is happening here is that we have no type definition for this global function named `require`.

The Node runtime does not ship with its own type definitions, so we need to import those types separately. Where can we find them? They are also in npm but need to be installed separately.

We can install the node runtime type definitions in the following way:

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

So what did this do, what is this `@types` module?

What is `@types`, when should I use it and why?

This `@types` scoped package is where we can find a ton of useful type definitions, such as for example the type definitions of node that allow us to use `require` for example.

But if you have been following Typescript for a while, you might remember something called DefinitivelyTyped, and a typings executable, that we used to use before to install type definitions.

What happened to the `typings` executable and DefinitivelyTyped?

If we head over to the npm page of the `@types` scoped package, we can see what happened there - @types.

As we can see, all the content of DefintivelyTyped is now available under the `@types` scoped package, and we don't need anymore to use the typings executable to download type definitions.

We can now simply use npm and the Typescript compiler will implicitly take any type definitions installed inside the `node_modules/@types` folder and include it during compilation transparently.

When to use @types then ?

The `@types` scope package contains type definitions for a lot of libraries, like Express, Sequelize, JQuery, and many others. So definitively have a look there if you are missing some type definitions, but make sure of two things first:

- check if the package you are using already has types built-in, and if so prefer those
- check if type definitions are already shipped with the compiler, more on this later

The type definitions inside `@types` are super helpful, but some of those types might not be appropriate anymore in certain situations. Let's give an example with the Promises library.

Using Request Promise to build a type safe promise call

Let's start by installing the type definitions for `request-promise` available in `@types`, and if you have been coding along its a great time to uninstall axios to avoid library conflicts:

```
npm uninstall axios --save

npm install --save-dev @types/request-promise
```

Now that we have the type definitions for `request-promise`, this is the program that we would like to be able to write:

```
import * as rp from 'request-promise';


interface Lesson {
    id:number;
    description: string;
}


function getLesson(lessonId:number): Promise<Lesson> {
    return rp.get(`lessons/${lessonId}`);
}

const promise  = getLesson(1);

promise.then(lesson => {
    .... we want this lesson variable to be implicitly of type Lesson
});
```

But at this stage, we get an error:

```
Error:(11, 38) TS2304: Cannot find name 'Promise'.
```

Understanding the Cannot Find Promise common issue

So it looks like the node runtime type definitions that we added to our program don't include promises. So let's have a look at `@types` to see where we can find them.

Please do read the conclusions section on this, but right now let's say that we would bring some type definitions from `@types` for Promises:

So what will this do? IT will install type definitions for ES6 promises, which includes a generic parameter for the Promise type.

So now we can say that this function returns a Promise of Lesson, and have the type of the variable lesson to be correctly inferred by the compiler as being of type Lesson.

Which is exactly what we wanted, but there is a huge catch. And it's a good example of how we should not take all the types available in `@types` systematically, but pick and choose.

What is the catch with the use of es6-promise?

To understand what the problem is, let's try the following program, which should throw an error:

```
import * as rp from 'request-promise';

interface Lesson {
    id:number;
    description: string;
}

```

```
8
9  function getLesson(lessonId:number): Promise<Lesson> {
10     return rp.get(`lessons/${lessonId}`)
11         .then((lesson:any) =>  Promise.resolve(lesson.description));
12 }
```

Can you see the issue? The function is returning a Promise of string, because the lesson has been transformed into a string by the then clause.

But yet the program compiles without any error. So what is going on here?

Not all type definitions leverage completely the Typescript type system

As we have seen before, not all type definitions leverage the type system to its maximum extent. This is also because the Typescript compiler features keep evolving so fast that not all type definitions leverage all the latest features.

Which can be great because we might not want to use generics all the time in our programs, so returning Any in our API and assuming the caller will add a type annotation is a viable solution as well.

But in this case, we would really like to use Promise with a generic parameter, because it's a really great fit for that. It really makes sense to specify in our program what type of data is the Promise expected to return, and use that information to type-check the program.

So what can we do? It turns out that the Typescript compiler itself ships with a ton of Type definitions ready to use, and one of them is Promises.

What are compiler opt-in types, when should I use them and why?

Have a look at the compiler options, at the `--lib` flag here.

There are a ton of type definitions available that come bundled with the compiler, including for example all the type definitions of ES6 itself, which includes Promises.

So we can simply leverage these by using the `lib` compiler flag:

So with this, the compiler knows of a Promise type that is built-in with the compiler. So if we compile our program with that turned on, what do we get?

Initially we will get a few errors:

```
node_modules/@types/es6-promise/index.d.ts(42,19): error
TS2300: Duplicate identifier 'Promise'.
node_modules/typescript/lib/lib.es2015.iterable.d.ts(145,1
1): error TS2300:     Duplicate identifier 'Promise'.
...
```

Why do I sometimes get this 'duplicate type definition' error?

In this case, we have a type definition named Promise in two places:

- one via `@types/es6-promise`
- the other via the built-in compiler types that we have opted-in

So how do we solve this duplicate type definition issue ? The solution is to uninstall the types that we had installed via `@types` for promises:

Now we are only going to get this errors:

```
test.ts(11,12): error TS2322: Type 'Bluebird<any>' is not
assignable to type 'Promise<Lesson>'.
Property '[Symbol.toStringTag]' is missing in type
'Bluebird<any>'.
```

It turns out that `@types/request-promise` already comes with its own Promise type definitions after all: Because those type definitions use internally Bluebird type definitions (Bluebird is a great promise library, used by Sequelize for example - a Node ORM).

So what can we do at this point? Because it looks like the type definitions of `@types/request-promise` are incompatible with the types from the ES6 built-in type definitions at the moment.

This a temporary situation because Bluebird promises used to be compatible with ES6 promises for a long time, and actually this is likely to have already been solved by the time you read this.

What to learn from this example?

This is a good example of how using the latest type definitions might not always be viable or the best approach at any given time. It's an option but needs to be weighed against other things.

Sometimes using the simpler types that are available like `AxiosPromise` or the return of `request-promise` calls is a much more viable

alternative, and then simply add a few extra type annotations where necessary, in case those APIs return Any.

> *it's not because an API returns an explicit any that we need to use any also in our own program*

So as a general guideline, it's better to choose packages that come with built-in types and use those types as much as possible via type inference, only importing third party types if needed and picking and choosing the best fit at each moment.

What would the error look like that we were looking for?

Let's try to see if Typescript catches the error that we are trying it to throw with a simpler example (taken from this issue report). How about this:

This would throw as expected the following error:

```
test.ts(31,16): error TS2322: Type 'Promise<string>' is
not assignable to type 'Promise<Foo>'.
  Type 'string' is not assignable to type 'Foo'.
```

So the compiler is prepared to handle the detection of this error. It's just that the library type definitions available need to evolve over time to leverage this functionality, and there will likely always be some sort of gap.

Handling the gap between libraries and the compiler

The Typescript compiler will apply the latest type checks to any type definitions available in node modules, including `@types`.

To avoid this, and ensure that only our program is checked by the compiler we can use the flag skipLibCheck to true.

This prevents the more recent versions of the compiler to throw errors against ancient libraries and also has the added benefit of increasing the performance of your build in a noticeable way.

OK so now we have reached the end, it was a bit of a roller coaster but this is very close to the everyday issues that you will find while developing with Typescript.

Let's try to make sense of it all, and try to come with some general guidelines to use the type system effectively.

Guidelines for Using the multiple Type Definitions available

If we are using Typescript, we have multiple sources of Type Definitions, and knowing which ones to choose and why is essential to be able to have a good developer experience.

We need to be aware of one thing: the compiler is always adding more features, but the libraries available might not leverage them yet.

When should we use compiler opt-in types?

It's a great idea to use the compiler built-in types (the `lib` flag) as much as possible because those types are written to maximize the type safety of our programs and make sure we conform to standard APIs.

But depending on the current state of existing libraries, it might introduce other issues. When opting-in to compiler built-in types, do make sure that they don't conflict with types that we already imported before like `es6-promise`, and that they don't affect third party types if not needed by using `skipLibCheck`.

But don't feel obliged to use opt-in types, there is a good reason why right now they are turned off by default, it's because of backwards compatibility with part of the existing ecosystem.

When should we use `@types`?

It's better to instead of using `@types` systematically, to try to use the built-in types of each module as much as possible, and use `@types` strategically if necessary for example for modules like Express or Sequelize.

Plain Javascript modules like those two will likely make the bulk of your program and have great types available on `@types`. But for example newer modules like Firebase: they already come with types now.

So if you also install `@types/firebase` you will run into duplicate type issues.

On the other hand things like `@types/node` are essential to writing any node program.

So the suggestion here is: have a look at the built-in types to see if there is anything there similar to what you are looking for. Have a look at the node module itself to see if it has already types inside it: this will be more and more common.

If no types are found neither inside the module nor built-in to the compiler, then have a look at `@types` to see if there are some good types there. Fast forwarding to the future, the ideal would be that `@types` no longer exists and that most libraries ship their own type definitions.

But that won't happen anytime soon and its great to have those high-quality types there available.

What if no type definitions are available?

If no type definition files are available for a given library, that will not prevent us from using it. With Typescript 2.1, we can import the library directly and anything imported will be assigned the type Any.

So we can seamlessly integrate with any existing Javascript module without special integration needed.

How to make sure our programs leverage type safety effectively?

One of the best things we can do other than carefully choosing the types we add to our program is to turn `noImplicitAny` to true.

This will have the effect that the compiler will be able to infer almost all types in your program, at the expense of only a few type annotations, especially in function arguments which is a great thing to add anyway.

Conclusions

Let's summarize what we have learned about Typescript. In the next section, have a look also at the bonus content video list. There is a link

to a free Angular course right at the end of the book.

I hope that you enjoyed this book and that it helped you make sense of the key aspects of the Typescript Type System and Type Definitions in the most effective way.

Please let me know your thoughts on this book in the comments.

As we could see, these two key points are crucial for understanding Typescript, and without them, we would have a hard time making sense of the language:

- it's a structural subtyping system and not a nominal subtyping system, meaning that type compatibility is achieved by comparing lists of properties and not type names

- there are multiple types of Type Definitions (`@types`, built-in compiler types and package types), and it's essential to know when to use each and why

With these two key points in place, we are in a great point for starting Typescript development: now you will really be able to leverage all the familiarity to the features that you already know from other languages, and will be able to cope with the vast majority of the type-safety related compiler error messages that you might come accross.

Typescript will likely evolve in the next few years as a full stack single language development solution: the future looks bright for Typescript!

I hope that you enjoyed this book and I will talk to you soon.

Kind Regards,
Vasco
Angular University

Typescript - A Video List

In this section, we are going to present a series of videos that cover some very commonly used Typescript features.

Click Here to View The Typescript Video List

These are the videos available on this list:

- Video 1 - Top 4 Advantages of Typescript - Why Typescript?
- Video 2 - ES6 / Typescript let vs const vs var When To Use Each? Const and Immutability
- Video 3 - Learn ES6 Object Destructuring (in Typescript), Shorthand Object Creation and How They Are Related

- Video 4 - Debugging Typescript in the Browser and a Node Server - Step By Step Instructions
- Video 5 - Build Type Safe Programs Without Classes Using Typescript
- Video 6 - The Typescript Any Type - How Does It Really Work?
- Video 7 - Typescript @types - Installing Type Definitions For 3rd Party Libraries
- Video 8 - Typescript Non-Nullable Types - Avoiding null and undefined Bugs
- Video 9 - Typescript Union and Intersection Types- Interface vs Type Aliases
- Video 10 - Typescript Tuple Types and Arrays Strong Typing