TypeScript

Deep Dive

Basarat Ali Syed

Table of Contents

TypeScript Deep Dive

I've been looking at the issues that turn up commonly when people start using TypeScript. This is based on the lessons from StackOverflow / DefinitelyTyped and general engagement with the TypeScript community. You can follow for updates and don't forget to ★ on Github

Reviews

- Thanks for the wonderful book. Learned a lot from it. (link)
- Its probably the Best TypeScript book out there. Good Job (link)
- Love how precise and clear the examples and explanations are! (link)
- For the low, low price of free, you get pages of pure awesomeness. Chock full of source code examples and clear, concise explanations, TypeScript Deep Dive will help you learn TypeScript development. (link)
- Just a big thank you! **Best TypeScript 2 detailed explanation!** (link)
- This gitbook got my project going pronto. Fluent easy read 5 stars. (link)
- I recommend the online #typescript book by @basarat you'll love it.(link)
- I've always found this by @basarat really helpful. (link)
- We must highlight TypeScript Deep Dive, an open source book.(link)
- Great online resource for learning. (link)
- Thank you for putting this book together, and for all your hard work within the TypeScript community. (link)
- TypeScript Deep Dive is one of the best technical texts I've read in a while. (link)
- Thanks @basarat for the TypeScript Deep Dive Book. Help me a lot with my first TypeScript project. (link)
- Thanks to @basarat for this great #typescript learning resource. (link)
- Guyz excellent book on Typescript(@typescriptlang) by @basarat (link)
- Leaning on the legendary @basarat's "TypeScript Deep Dive" book heavily at the moment (link)
- numTimesPointedPeopleToBasaratsTypeScriptBook++; (link)
- A book not only for typescript, a good one for deeper javascript knowledge as well. link

Get Started

If you are here to read the book online get started.

Other Options

You can also download one of the following:

- EPUB for iPad,iPhone,Mac
- PDF for Windows and others
- MOBI for Kindle

Special Thanks

All the amazing contributors

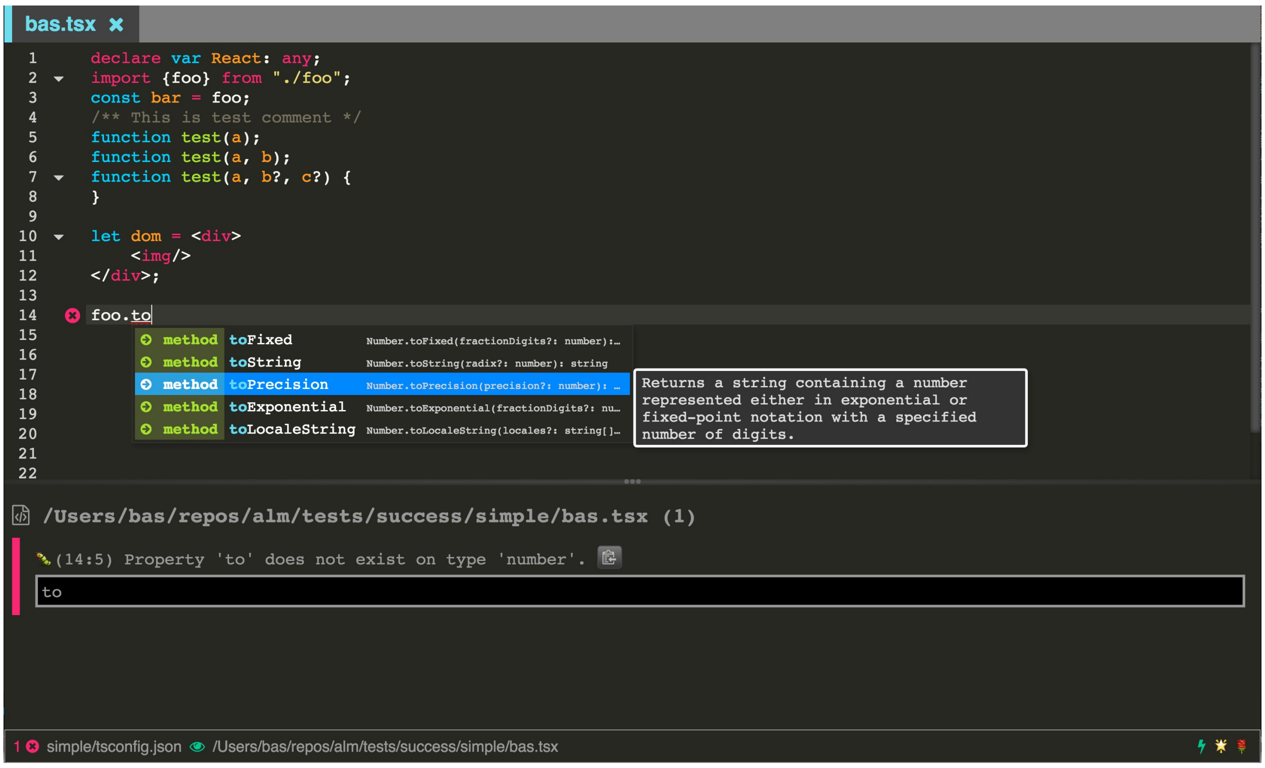
Share

Share URL: http://basarat.gitbooks.io/typescript/

Getting Started With TypeScript

TypeScript compiles into JavaScript. JavaScript is what you are actually going to execute (either in the browser or on the server). So you are going to need the following:

* TypeScript compiler (OSS available in source and on NPM)
* A TypeScript editor (you can use notepad if you want but I use alm . Also lots of other IDES support it as well)



bas.tsx

```
declare var React: any;
import {foo} from "./foo";
const bar = foo;
/** This is test comment */
function test(a);
function test(a, b);
function test(a, b?, c?) {
}

let dom = <div>
    <img/>
</div>;

foo.to
```

```
method  toFixed          Number.toFixed(fractionDigits?: number):…
method  toString         Number.toString(radix?: number): string
method  toPrecision      Number.toPrecision(precision?: number): …
method  toExponential    Number.toExponential(fractionDigits?: nu…
method  toLocaleString   Number.toLocaleString(locales?: string[]…
```

```
Returns a string containing a number
represented either in exponential or
fixed-point notation with a specified
number of digits.
```

```
/Users/bas/repos/alm/tests/success/simple/bas.tsx (1)
(14:5) Property 'to' does not exist on type 'number'.
to
```

1 simple/tsconfig.json /Users/bas/repos/alm/tests/success/simple/bas.tsx

TypeScript Version

Instead of using the *stable* TypeScript compiler we will be presenting a lot of new stuff in this book that may not be associated with a version number yet. I generally recommend people to use the nightly version because **the compiler test suite only catches more bugs over time**.

You can install it on the command line as

```
npm install -g typescript@next
```

And now the command line `tsc` will be the latest and greatest. Various IDEs support it too, e.g.

- `alm` always ships with the latest TypeScript version.
- You can ask vscode to use this version by creating `.vscode/settings.json` with the following contents:

```json
{
  "typescript.tsdk": "./node_modules/typescript/lib"
}
```

Getting the Source Code

The source for this book is available in the books github repository https://github.com/basarat/typescript-book/tree/master/code most of the code samples can be copied into alm and you can play with them as is. For code samples that need additional setup (e.g. npm modules), we will link you to the code sample before presenting the code. e.g.

`this/will/be/the/link/to/the/code.ts`

```ts
// This will be the code under discussion
```

With a dev setup out of the way let's jump into TypeScript syntax.

Why TypeScript

There are two main goals of TypeScript:

- Provide an *optional type system* for JavaScript.
- Provide planned features from future JavaScript editions to current JavaScript engines

The desire for these goals is motivated below.

The TypeScript type system

You might be wondering "**Why add types to JavaScript?**"

Types have proven ability to enhance code quality and understandability. Large teams (google,microsoft,facebook) have continually arrived at this conclusion. Specifically:

- Types increase your agility when doing refactoring. *It's better for the compiler to catch errors than to have things fail at runtime*.
- Types are one of the best forms of documentation you can have. *The function signature is a theorem and the function body is the proof*.

However types have a way of being unnecessarily ceremonious. TypeScript is very particular about keeping the barrier to entry as low as possible. Here's how:

Your JavaScript is TypeScript

TypeScript provides compile time type safety for your JavaScript code. This is no surprise given its name. The great thing is that the types are completely optional. Your JavaScript code `.js` file can be renamed to a `.ts` file and TypeScript will still give you back valid `.js` equivalent to the original JavaScript file. TypeScript is *intentionally* and strictly a superset of JavaScript with optional Type checking.

Types can be Implicit

TypeScript will try to infer as much of the type information as it can in order to give you type safety with minimal cost of productivity during code development. For example, in the following example TypeScript will know that foo is of type `number` below and will give an error on the second line as shown:

```
var foo = 123;
foo = '456'; // Error: cannot assign `string` to `number`

// Is foo a number or a string?
```

This type inference is well motivated. If you do stuff like shown in this example, then, in the rest of your code, you cannot be certain that `foo` is a `number` or a `string`. Such issues turn up often in large multi-file code bases. We will deep dive into the type inference rules later.

Types can be Explicit

As we've mentioned before, TypeScript will infer as much as it can safely, however you can use annotations to:

1. Help along the compiler, and more importantly document stuff for the next developer who has to read your code (that might be future you!).
2. Enforce that what the compiler sees, is what you thought it should see. That is your understanding of the code matches an algorithmic analysis of the code (done by the compiler).

TypeScript uses postfix type annotations popular in other *optionally* annotated languages (e.g. ActionScript and F#).

```
var foo: number = 123;
```

So if you do something wrong the compiler will error e.g.:

```
var foo: number = '123'; // Error: cannot assign a `string` to a `number`
```

We will discuss all the details of all the annotation syntax supported by TypeScript in a later chapter.

Types are structural

In some languages (specifically nominally typed ones) static typing results in unnecessary ceremony because even though *you know* that the code will work fine the language semantics force you to copy stuff around. This is why stuff like automapper for C# is *vital* for C#. In TypeScript because we really want it to be easy for JavaScript developers with a minimum cognitive overload, types are *structural*. This means that *duck typing* is a first class language construct. Consider the following example. The function `iTakePoint2D` will accept anything that contains all the things (`x` and `y`) it expects:

```
interface Point2D {
    x: number;
    y: number;
}
interface Point3D {
    x: number;
    y: number;
    z: number;
}
var point2D: Point2D = { x: 0, y: 10 }
var point3D: Point3D = { x: 0, y: 10, z: 20 }
function iTakePoint2D(point: Point2D) { /* do something */ }

iTakePoint2D(point2D); // exact match okay
iTakePoint2D(point3D); // extra information okay
iTakePoint2D({ x: 0 }); // Error: missing information `y`
```

Type errors do not prevent JavaScript emit

To make it easy for you to migrate your JavaScript code to TypeScript, even if there are compilation errors, by default TypeScript *will emit valid JavaScript* the best that it can. e.g.

```
var foo = 123;
foo = '456'; // Error: cannot assign a `string` to a `number`
```

will emit the following js:

```
var foo = 123;
foo = '456';
```

So you can incrementally upgrade your JavaScript code to TypeScript. This is very different from how many other language compilers work and yet another reason to move to TypeScript.

Types can be ambient

A major design goal of TypeScript was to make it possible for you to safely and easily use existing JavaScript libraries in TypeScript. TypeScript does this by means of *declaration*. TypeScript provides you with a sliding scale of how much or how little effort you want to put in your declarations, the more effort you put the more type safety + code intelligence you get. Note that definitions for most of the popular JavaScript libraries have already been written for you by the DefinitelyTyped community so for most purposes either:

1. The definition file already exists.
2. Or at the very least, you have a vast list of well reviewed TypeScript declaration templates already available

As a quick example of how you would author your own declaration file, consider a trivial example of jquery. By default (as is to be expected of good JS code) TypeScript expects you to declare (i.e. use `var` somewhere) before you use a variable

```
$('.awesome').show(); // Error: cannot find name `$`
```

As a quick fix *you can tell TypeScript* that there is indeed something called `$` :

```
declare var $:any;
$('.awesome').show(); // Okay!
```

If you want you can build on this basic definition and provide more information to help protect you from errors:

```
declare var $:{
    (selector:string): any;
};
$('.awesome').show(); // Okay!
$(123).show(); // Error: selector needs to be a string
```

We will discuss the details of creating TypeScript definitions for existing JavaScript in detail later once you know more about TypeScript (e.g. stuff like `interface` and the `any`).

Future JavaScript => Now

TypeScript provides a number of features that are planned in ES6 for current JavaScript engines (that only support ES5 etc). The typescript team is actively adding these features and this list is only going to get bigger over time and we will cover this in its own section. But just as a specimen here is an example of a class:

```
class Point {
    constructor(public x: number, public y: number) {
    }
    add(point: Point) {
        return new Point(this.x + point.x, this.y + point.y);
    }
}

var p1 = new Point(0, 10);
var p2 = new Point(10, 20);
var p3 = p1.add(p2); // {x:10,y:30}
```

and the lovely fat arrow function:

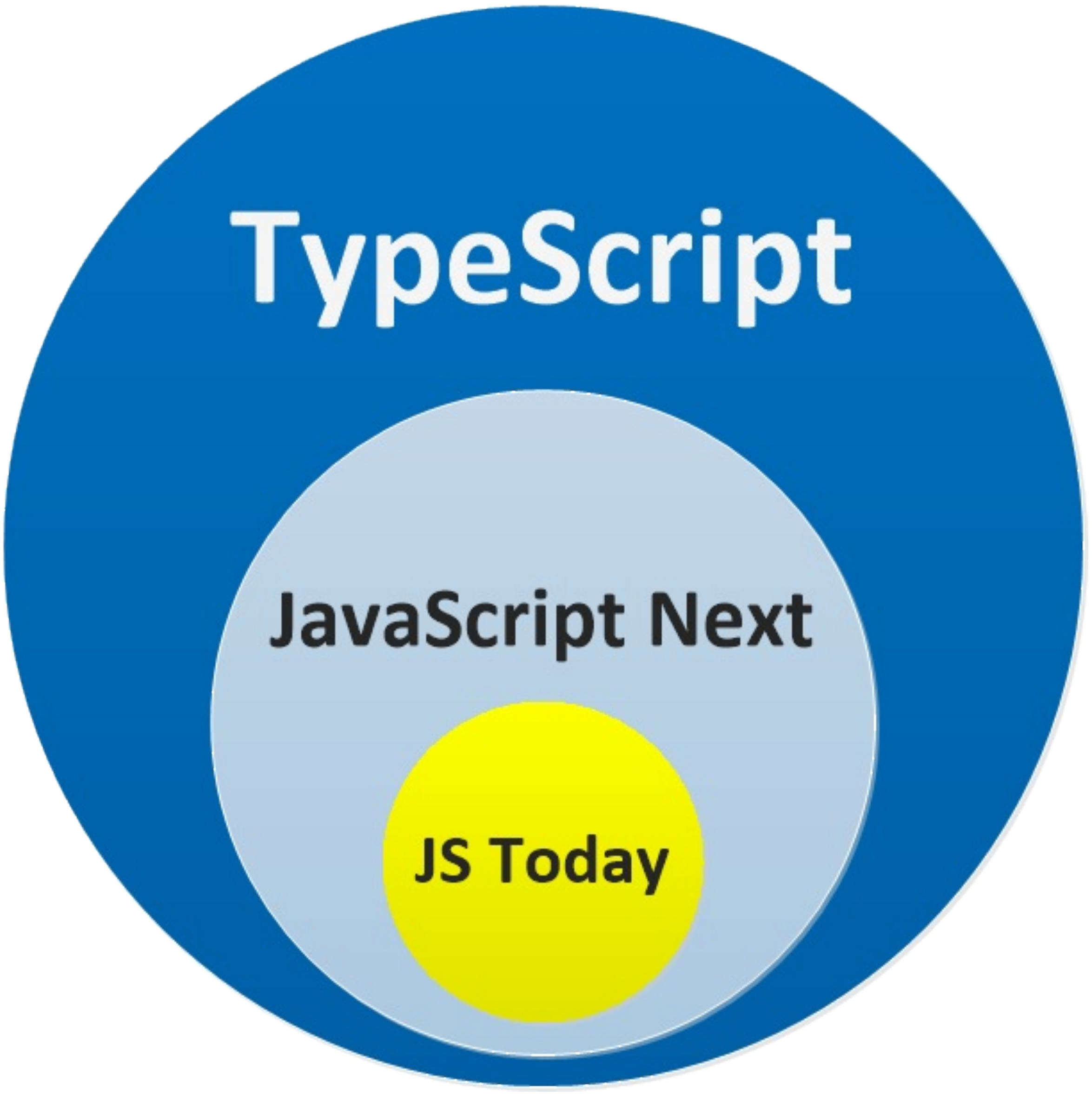
```
var inc = (x)=>x+1;
```

Summary

In this section we have provided you with the motivation and design goals of TypeScript. With this out of the way we can dig into the nitty gritty details of TypeScript.

Your JavaScript is TypeScript

There were (and will continue to be) a lot of competitors in *Some syntax* to *JavaScript* compilers. TypeScript is different from them in that *Your JavaScript is TypeScript*. Here's a diagram:



However it does mean that *you need to learn JavaScript* (the good news is *you **only** need to learn JavaScript*). TypeScript is just standardizing all the ways you provide *good documentation* on JavaScript.

- Just giving you a new syntax doesn't help fix bugs (looking at you CoffeeScript).
- Creating a new language abstracts you too far from your runtimes, communities (looking at you Dart).

TypeScript is just JavaScript with docs.

Making JavaScript Better

TypeScript will try to protect you from portions of JavaScript that never worked (so you don't need to remember this stuff):

```
[] + []; // JavaScript will give you "" (which makes little sense), TypeScript will er
ror

//
// other things that are nonsensical in JavaScript
// - don't give a runtime error (making debugging hard)
// - but TypeScript will give a compile time error (making debugging unnecessary)
//
{} + []; // JS : 0, TS Error
[] + {}; // JS : "[object Object]", TS Error
{} + {}; // JS : NaN, TS Error
"hello" - 1; // JS : NaN, TS Error

function add(a,b) {
  return
    a + b; // JS : undefined, TS Error 'unreachable code detected'
}
```

Essentially TypeScript is linting JavaScript. Just doing a better job at it than other linters that don't have *type information*.

You still need to learn JavaScript

That said TypeScript is very pragmatic about the fact that *you do write JavaScript* so there are some things about JavaScript that you still need to know in order to not be caught off-guard. Let's discuss them next.

JavaScript the awful parts

Here are some awful (misunderstood) parts of JavaScript that you must know.

> Note: TypeScript is a superset of JavaScript. Just with documentation that can actually be used by compilers / IDEs ;)

Null and Undefined

Fact is you will need to deal with both. Just check for either with `==` check.

```
/// Imagine you are doing `foo.bar == undefined` where bar can be one of:
console.log(undefined == undefined); // true
console.log(null == undefined); // true
console.log(0 == undefined); // false
console.log('' == undefined); // false
console.log(false == undefined); // false
```

Recommend `== null` to check for both `undefined` or `null`. You generally don't want to make a distinction between the two.

undefined

Remember how I said you should use `== null`. Of course you do (cause I just said it ^). Don't use it for root level things. In strict mode if you use `foo` and `foo` is undefined you get a `ReferenceError` **exception** and the whole call stack unwinds.

> You should use strict mode ... and in fact the TS compiler will insert it for you if you use modules ... more on those later in the book so you don't have to be explicit about it :)

So to check if a variable is defined or not at a *global* level you normally use `typeof`:

```
if (typeof someglobal !== 'undefined') {
  // someglobal is now safe to use
  console.log(someglobal);
}
```

this

Any access to `this` keyword within a function is actually controlled by how the function is actually called. It is commonly referred to as the `calling context`.

Here is an example:

```
function foo() {
  console.log(this);
}

foo(); // logs out the global e.g. `window` in browsers
let bar = {
  foo
}
bar.foo(); // Logs out `bar` as `foo` was called on `bar`
```

So be mindful of your usage of `this`. If you want to disconnect `this` in a class from the calling context use an arrow function, more on that later.

Next

That's it. Those are the simple *misunderstood* portions of JavaScript that still result in various bugs for developers that are new to the language .

Closure

The best thing that JavaScript ever got was closures. A function in JavaScript has access to any variables defined in the outer scope. Closures are best explained with examples:

```
function outerFunction(arg) {
    var variableInOuterFunction = arg;

    function bar() {
        console.log(variableInOuterFunction); // Access a variable from the outer scope

    }

    // Call the local function to demonstrate that it has access to arg
    bar();
}

outerFunction("hello closure"); // logs hello closure!
```

You can see that the inner function has access to a variable (variableInOuterFunction) from the outer scope. The variables in the outer function have been closed by (or bound in) the inner function. Hence the term **closure**. The concept in itself is simple enough and pretty intuitive.

Now the awesome part: The inner function can access the variables from the outer scope *even after the outer function has returned*. This is because the variables are still bound in the inner function and not dependent on the outer function. Again let's look at an example:

```
function outerFunction(arg) {
    var variableInOuterFunction = arg;
    return function() {
        console.log(variableInOuterFunction);
    }
}

var innerFunction = outerFunction("hello closure!");

// Note the outerFunction has returned
innerFunction(); // logs hello closure!
```

Reason why it's awesome

It allows you to compose objects easily e.g. the revealing module pattern:

```
function createCounter() {
    let val = 0;
    return {
        increment() { val++ },
        getVal() { return val }
    }
}

let counter = createCounter();
counter.increment();
console.log(counter.getVal()); // 1
counter.increment();
console.log(counter.getVal()); // 2
```

At a high level it is also what makes something like nodejs possible (don't worry if it doesn't click in your brain right now. It will eventually):

```
// Pseudo code to explain the concept
server.on(function handler(req, res) {
    loadData(req.id).then(function(data) {
        // the `res` has been closed over and is available
        res.send(data);
    })
});
```