Prerequisites

You need to have Node.js and NPM installed. Check out the following posts for help if you haven’t done so already

* Installing Node.js and NPM on Windows
* How to Install Node.js and NPM on Linux
* How to Install Node.js and NPM on a Mac

Installation

TypeScript can be installed through the NPM package manager.

npm install -g typescript

The -g means it’s installed on your system globally so that the TypeScript compiler can be used in any of your projects.

Test that the TypeScript is installed correctly by typing tsc -v in to your terminal or command prompt. You should see the TypeScript version print out to the screen.

For help on possible arguments you can type tsc -h or just tsc.

Command Line Usage

You can use the tsc command in several different ways. Here’s a couple of helpful common usages.

Run and Compile

The following command will compile a single .ts file in to a .js file:

tsc app.ts

This will result in an app.js file being created.

To compile multiple .ts files:

tsc app.ts another.ts someMore.ts

This will result in 3 files, app.js, another.js and someMore.js.

You can also use wildcards too. The following command will compile all TypeScript files in the current folder.

tsc \*.ts

All TypeScript files will compile to their corresponding JavaScript files.

Joining Files

You can also compile all your TypeScript files down to a single JavaScript file. This can reduce the number of HTTP requests a browser has to make and improve performance on HTTP 1.x sites. To do this use the --out option like so:

tsc \*.ts --out app.js

Watcher

Instead of running the tsc command all the time you can use the option --watch.

tsc \*.ts --out app.js --watch

Every time there’s an update to a TypeScript file it’ll recompile the source files to JavaScript.

If you’re using a wildcard like this, any new files created since running the tsc command won’t get compiled, you need to stop the watcher and start again.

Syntax

Now on to the good stuff – the stuff that makes you less error prone and more productive – it’s syntax.

Optional Typing

When there’s a variable or an argument in a function or method call, you can annotate your code with types. To annotate, follow the variable or argument with a colon and followed by it’s type.

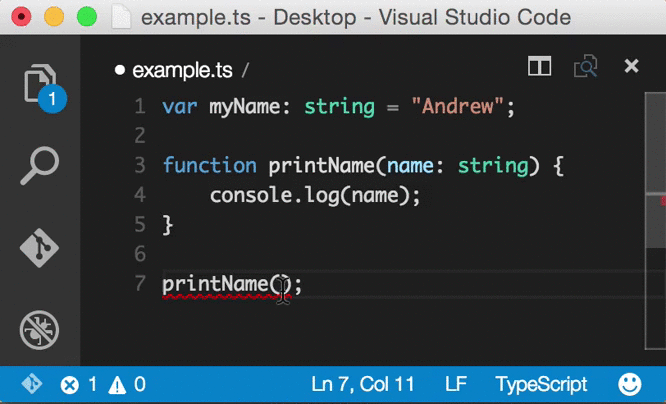
var myName: string = "Andrew";

function printName(name: string) {

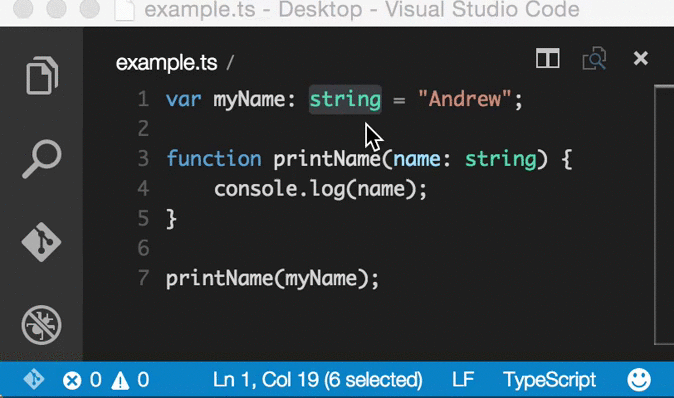
console.log(name);

}

When you try and call the function with the missing or wrong types of parameter it warns us. Also it helps with autocompletion since every variable’s type is known and it suggests an appropriate variable to pass into the function call.



In fact, TypeScript is clever enough to know the type of the myName variable.



Interfaces

Interfaces are great ways to set up agreements on the shape of object literals. Sometimes you just need to know the structure of a thing as a data store.

In this code we want every object pushed in to the addressBook array to conform to the Contact interface. When this is compiled to JavaScript this disappears, but this is helpful in development.

interface Contact {

name: string,

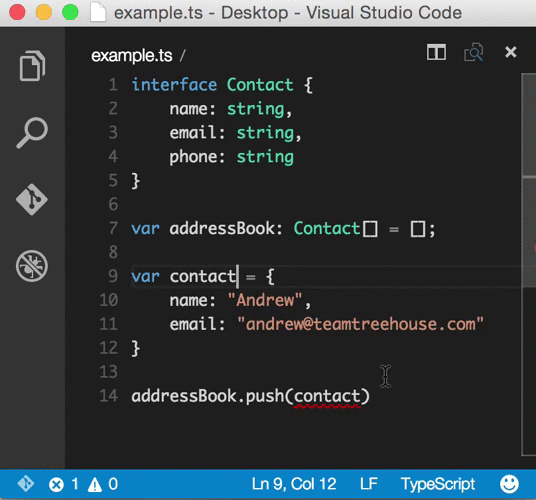
email: string,

phone: string

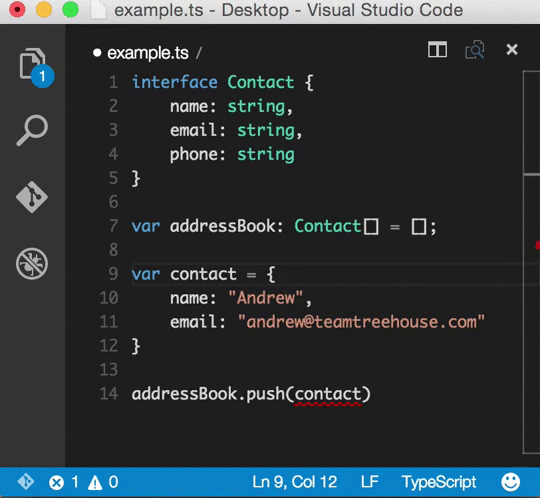
}

var addressBook: Contact[] = [];

It can warn me in the method call of push if I don’t define the type on line 9.



Or if I specify the type on the variable declaration, it could warn me on line 9 that I’m missing something in the object literal that I’m assigning to the variable. The added benefit of this is that it gives me autocompletion for the properties missing. This can be particularly helpful for when you’re dealing with a lot of object literals.



Classes

TypeScript uses ECMAScript 6 classes and adds a little TypeScript goodness.

Take this ECMAScript class.

class Monster {

constructor(name, initialPosition) {

this.name = name;

this.initialPosition = initialPosition;

}

}

This would be totally fine in an ECMAScript 6 environment but TypeScript would like you to define the properties on your object.

interface Point {

x: number,

y: number

}

class Monster {

name: string;

initialPosition: Point

constructor(name, initialPosition) {

this.name = name;

this.initialPosition = initialPosition;

}

}

var scary = new Monster("Alien", {x: 0, y: 0});

However there’s a shorthand, the following code does the same thing.

interface Point {

x: number,

y: number

}

class Monster {

constructor(public name: string, public initialPosition: Point) {

}

}

var scary = new Monster("Alien", {x: 0, y: 0});

Note how the constructer has the public keyword before each variable? That means you want that parameter to be a public property on the object.

So scary.name will return the string "Alien" and scary.initalPosition will be {x: 0, y: 0}.