**Contents**

### [About](#_bookmark0)

................................................................................................................................................................................... [1](#_bookmark0)

### [Chapter 1: Getting started with TypeScript](#_bookmark1)

.................................................................................................... [2](#_bookmark1)

[Section 1.1: Installation and setup](#_bookmark2) [2](#_bookmark2)

[Section 1.2: Basic syntax](#_bookmark3) [Section 1.3: Hello World](#_bookmark4)

............................................................................................................................................... [4](#_bookmark3)

................................................................................................................................................. [5](#_bookmark4)

[Section 1.4: Running TypeScript using ts-node](#_bookmark5) [6](#_bookmark5)

[Section 1.5: TypeScript REPL in Node.js](#_bookmark6) [6](#_bookmark6)

### [Chapter 2: Why and when to use TypeScript](#_bookmark7)

................................................................................................. [8](#_bookmark7)

[Section 2.1: Safety](#_bookmark8) [8](#_bookmark8)

[Section 2.2: Readability](#_bookmark9) [8](#_bookmark9)

[Section 2.3: Tooling](#_bookmark10) [8](#_bookmark10)

### [Chapter 3: TypeScript Core Types](#_bookmark11)

....................................................................................................................... [9](#_bookmark11)

[Section 3.1: String Literal Types](#_bookmark12) [9](#_bookmark12)

[Section 3.2: Tuple](#_bookmark13) [12](#_bookmark13)

[Section 3.3: Boolean](#_bookmark14) [12](#_bookmark14)

[Section 3.4: Intersection Types](#_bookmark15) [13](#_bookmark15)

[Section 3.5: Types in function arguments and return value. Number](#_bookmark16) [13](#_bookmark16)

[Section 3.6: Types in function arguments and return value. String](#_bookmark17) [14](#_bookmark17)

[Section 3.7: const Enum](#_bookmark18) [14](#_bookmark18)

[Section 3.8: Number](#_bookmark19) [15](#_bookmark19)

[Section 3.9: String](#_bookmark20) [Section 3.10: Array](#_bookmark21) [Section 3.11: Enum](#_bookmark22)

........................................................................................................................................................ [15](#_bookmark20)

....................................................................................................................................................... [16](#_bookmark21)

........................................................................................................................................................ [16](#_bookmark22)

[Section 3.12: Any](#_bookmark23) [Section 3.13: Void](#_bookmark24)

.......................................................................................................................................................... [16](#_bookmark23)

......................................................................................................................................................... [16](#_bookmark24)

### [Chapter 4: Arrays](#_bookmark25)

....................................................................................................................................................... [17](#_bookmark25)

[Section 4.1: Finding Object in Array](#_bookmark26) [17](#_bookmark26)

### [Chapter 5: Enums](#_bookmark27)

........................................................................................................................................................ [18](#_bookmark27)

[Section 5.1: Enums with explicit values](#_bookmark28) [18](#_bookmark28)

[Section 5.2: How to get all enum values](#_bookmark29) [19](#_bookmark29)

[Section 5.3: Extending enums without custom enum implementation](#_bookmark30) [19](#_bookmark30)

[Section 5.4: Custom enum implementation: extends for enums](#_bookmark31) [19](#_bookmark31)

### [Chapter 6: Functions](#_bookmark32)

................................................................................................................................................. [21](#_bookmark32)

[Section 6.1: Optional and Default Parameters](#_bookmark33) [21](#_bookmark33)

[Section 6.2: Function as a parameter](#_bookmark34) [21](#_bookmark34)

[Section 6.3: Functions with Union Types](#_bookmark35) [23](#_bookmark35)

[Section 6.4: Types of Functions](#_bookmark36) [23](#_bookmark36)

### [Chapter 7: Classes](#_bookmark37)

...................................................................................................................................................... [24](#_bookmark37)

[Section 7.1: Abstract Classes](#_bookmark38) [24](#_bookmark38)

[Section 7.2: Simple class](#_bookmark39) [24](#_bookmark39)

[Section 7.3: Basic Inheritance](#_bookmark40) [25](#_bookmark40)

[Section 7.4: Constructors](#_bookmark41) [25](#_bookmark41)

[Section 7.5: Accessors](#_bookmark42) [26](#_bookmark42)

[Section 7.6: Transpilation](#_bookmark43) [27](#_bookmark43)

[Section 7.7: Monkey patch a function into an existing class](#_bookmark44) [28](#_bookmark44)

### [Chapter 8: Class Decorator](#_bookmark45)

................................................................................................................................... [29](#_bookmark45)

[Section 8.1: Generating metadata using a class decorator](#_bookmark46) [29](#_bookmark46)

[Section 8.2: Passing arguments to a class decorator](#_bookmark47) [29](#_bookmark47)

[Section 8.3: Basic class decorator](#_bookmark48) [30](#_bookmark48)

### [Chapter 9: Interfaces](#_bookmark49)

................................................................................................................................................ [32](#_bookmark49)

[Section 9.1: Extending Interface](#_bookmark50) [32](#_bookmark50)

[Section 9.2: Class Interface](#_bookmark51) [32](#_bookmark51)

[Section 9.3: Using Interfaces for Polymorphism](#_bookmark52) [33](#_bookmark52)

[Section 9.4: Generic Interfaces](#_bookmark53) [34](#_bookmark53)

[Section 9.5: Add functions or properties to an existing interface](#_bookmark54) [35](#_bookmark54)

[Section 9.6: Implicit Implementation And Object Shape](#_bookmark55) [35](#_bookmark55)

[Section 9.7: Using Interfaces to Enforce Types](#_bookmark56) [36](#_bookmark56)

### [Chapter 10: Generics](#_bookmark57)

................................................................................................................................................. [37](#_bookmark57)

[Section 10.1: Generic Interfaces](#_bookmark58) [37](#_bookmark58)

[Section 10.2: Generic Class](#_bookmark59) [37](#_bookmark59)

[Section 10.3: Type parameters as constraints](#_bookmark60) [38](#_bookmark60)

[Section 10.4: Generics Constraints](#_bookmark61) [38](#_bookmark61)

[Section 10.5: Generic Functions](#_bookmark62) [39](#_bookmark62)

[Section 10.6: Using generic Classes and Functions:](#_bookmark63) [39](#_bookmark63)

### [Chapter 11: Strict null checks](#_bookmark64)

................................................................................................................................ [40](#_bookmark64)

[Section 11.1: Strict null checks in action](#_bookmark65) [40](#_bookmark65)

[Section 11.2: Non-null assertions](#_bookmark66) [40](#_bookmark66)

### [Chapter 12: User-deﬁned Type Guards](#_bookmark67)

........................................................................................................... [42](#_bookmark67)

[Section 12.1: Type guarding functions](#_bookmark68) [42](#_bookmark68)

[Section 12.2: Using instanceof](#_bookmark69) [43](#_bookmark69)

[Section 12.3: Using typeof](#_bookmark70) [43](#_bookmark70)

### [Chapter 13: TypeScript basic examples](#_bookmark71)

.......................................................................................................... [45](#_bookmark71)

[Section 13.1: 1 basic class inheritance example using extends and super keyword](#_bookmark72) [45](#_bookmark72)

[Section 13.2: 2 static class variable example - count how many time method is being invoked](#_bookmark73) [45](#_bookmark73)

### [Chapter 14: Importing external libraries](#_bookmark74)

........................................................................................................ [46](#_bookmark74)

[Section 14.1: Finding deﬁnition ﬁles](#_bookmark75) [46](#_bookmark75)

[Section 14.2: Importing a module from npm](#_bookmark76) [47](#_bookmark76)

[Section 14.3: Using global external libraries without typings](#_bookmark77) [47](#_bookmark77)

[Section 14.4: Finding deﬁnition ﬁles with TypeScript 2.x](#_bookmark78)

### [Chapter 15: Modules - exporting and importing](#_bookmark79)

.......................................................................................... [47](#_bookmark78)

......................................................................................... [49](#_bookmark79)

[Section 15.1: Hello world module](#_bookmark80) [49](#_bookmark80)

[Section 15.2: Re-export](#_bookmark81) [49](#_bookmark81)

[Section 15.3: Exporting/Importing declarations](#_bookmark82) [51](#_bookmark82)

### [Chapter 16: Publish TypeScript deﬁnition ﬁles](#_bookmark83)

[Section 16.1: Include deﬁnition ﬁle with library on npm](#_bookmark84)

............................................................................................ [52](#_bookmark83)

........................................................................................... [52](#_bookmark84)

### [Chapter 17: Using TypeScript with webpack](#_bookmark85)

................................................................................................ [53](#_bookmark85)

[Section 17.1: webpack.conﬁg.js](#_bookmark86) [53](#_bookmark86)

### [Chapter 18: Mixins](#_bookmark87)

....................................................................................................................................................... [54](#_bookmark87)

[Section 18.1: Example of Mixins](#_bookmark88) [54](#_bookmark88)

### [Chapter 19: How to use a JavaScript library without a type deﬁnition ﬁle](#_bookmark89)

................................. [55](#_bookmark89)

[Section 19.1: Make a module that exports a default any](#_bookmark90) [55](#_bookmark90)

[Section 19.2: Declare an any global](#_bookmark91) [55](#_bookmark91)

[Section 19.3: Use an ambient module](#_bookmark92) [56](#_bookmark92)

### [Chapter 20: TypeScript installing typescript and running the typescript compiler tsc](#_bookmark93)

........ [57](#_bookmark93)

[Section 20.1: Steps](#_bookmark94) [57](#_bookmark94)

### [Chapter 21: Conﬁgure typescript project to compile all ﬁles in typescript.](#_bookmark95)

................................ [59](#_bookmark95)

[Section 21.1: TypeScript Conﬁguration ﬁle setup](#_bookmark96) [59](#_bookmark96)

### [Chapter 22: Integrating with Build Tools](#_bookmark97)

........................................................................................................ [61](#_bookmark97)

[Section 22.1: Browserify](#_bookmark98) [Section 22.2: Webpack](#_bookmark99)

.............................................................................................................................................. [61](#_bookmark98)

............................................................................................................................................... [61](#_bookmark99)

[Section 22.3: Grunt](#_bookmark100) [Section 22.4: Gulp](#_bookmark101)

...................................................................................................................................................... [62](#_bookmark100)

........................................................................................................................................................ [62](#_bookmark101)

[Section 22.5: MSBuild](#_bookmark102) [63](#_bookmark102)

[Section 22.6: NuGet](#_bookmark103) [63](#_bookmark103)

[Section 22.7: Install and conﬁgure webpack + loaders](#_bookmark104) [64](#_bookmark104)

### [Chapter 23: Using TypeScript with RequireJS](#_bookmark105)

............................................................................................. [65](#_bookmark105)

[Section 23.1: HTML example using RequireJS CDN to include an already compiled TypeScript ﬁle](#_bookmark106) [65](#_bookmark106)

[Section 23.2: tsconﬁg.json example to compile to view folder using RequireJS import style](#_bookmark107) [65](#_bookmark107)

### [Chapter 24: TypeScript with AngularJS](#_bookmark108)

......................................................................................................... [66](#_bookmark108)

[Section 24.1: Directive](#_bookmark109) [66](#_bookmark109)

[Section 24.2: Simple example](#_bookmark110) [67](#_bookmark110)

[Section 24.3: Component](#_bookmark111) [67](#_bookmark111)

### [Chapter 25: TypeScript with SystemJS](#_bookmark112)

........................................................................................................... [69](#_bookmark112)

[Section 25.1: Hello World in the browser with SystemJS](#_bookmark113) [69](#_bookmark113)

### [Chapter 26: Using TypeScript with React (JS & native)](#_bookmark114)

......................................................................... [72](#_bookmark114)

[Section 26.1: ReactJS component written in TypeScript](#_bookmark115) [72](#_bookmark115)

[Section 26.2: TypeScript & react & webpack](#_bookmark116) [73](#_bookmark116)

### [Chapter 27: TSLint - assuring code quality and consistency](#_bookmark117)

............................................................... [75](#_bookmark117)

[Section 27.1: Conﬁguration for fewer programming errors](#_bookmark118) [75](#_bookmark118)

[Section 27.2: Installation and setup](#_bookmark119) [75](#_bookmark119)

[Section 27.3: Sets of TSLint Rules](#_bookmark120) [76](#_bookmark120)

[Section 27.4: Basic tslint.json setup](#_bookmark121) [76](#_bookmark121)

[Section 27.5: Using a predeﬁned ruleset as default](#_bookmark122) [76](#_bookmark122)

### [Chapter 28: tsconﬁg.json](#_bookmark123)

........................................................................................................................................ [78](#_bookmark123)

[Section 28.1: Create TypeScript project with tsconﬁg.json](#_bookmark124) [78](#_bookmark124)

[Section 28.2: Conﬁguration for fewer programming errors](#_bookmark125) [79](#_bookmark125)

[Section 28.3: compileOnSave](#_bookmark126) [80](#_bookmark126)

[Section 28.4: Comments](#_bookmark127) [80](#_bookmark127)

[Section 28.5: preserveConstEnums](#_bookmark128) [81](#_bookmark128)

### [Chapter 29: Debugging](#_bookmark129)

............................................................................................................................................ [82](#_bookmark129)

[Section 29.1: TypeScript with ts-node in WebStorm](#_bookmark130) [82](#_bookmark130)

[Section 29.2: TypeScript with ts-node in Visual Studio Code](#_bookmark131) [83](#_bookmark131)

[Section 29.3: JavaScript with SourceMaps in Visual Studio Code](#_bookmark132) [84](#_bookmark132)

[Section 29.4: JavaScript with SourceMaps in WebStorm](#_bookmark133) [84](#_bookmark133)

### [Chapter 30: Unit Testing](#_bookmark134)

......................................................................................................................................... [86](#_bookmark134)

[Section 30.1: tape](#_bookmark135) [86](#_bookmark135)

[Section 30.2: jest (ts-jest)](#_bookmark136) [87](#_bookmark136)

[Section 30.3: Alsatian](#_bookmark137) [89](#_bookmark137)

[Section 30.4: chai-immutable plugin](#_bookmark138) [89](#_bookmark138)

### [Credits](#_bookmark139)

.............................................................................................................................................................................. [91](#_bookmark139)

### [You may also like](#_bookmark140)

........................................................................................................................................................ [93](#_bookmark140)

**Chapter 1: Getting started with TypeScript**

|  |  |
| --- | --- |
| **Version** | **Release Date** |
| [2.8.3](https://github.com/Microsoft/TypeScript/releases/tag/v2.8.3) | 2018-04-20 |
| [2.8](https://github.com/Microsoft/TypeScript/releases/tag/v2.8.1) | 2018-03-28 |
| [2.8 RC](https://github.com/Microsoft/TypeScript/releases/tag/v2.8-rc) | 2018-03-16 |
| [2.7.2](https://github.com/Microsoft/TypeScript/releases/tag/v2.7.2) | 2018-02-16 |
| [2.7.1](https://github.com/Microsoft/TypeScript/releases/tag/v2.7.1) | 2018-02-01 |
| [2.7 beta](https://github.com/Microsoft/TypeScript/releases/tag/v2.7-rc) | 2018-01-18 |
| [2.6.1](https://github.com/Microsoft/TypeScript/releases/tag/v2.6.1) | 2017-11-01 |
| [2.5.2](https://github.com/Microsoft/TypeScript/releases/tag/v2.5.2) | 2017-09-01 |
| [2.4.1](https://github.com/Microsoft/TypeScript/releases/tag/v2.4.1) | 2017-06-28 |
| [2.3.2](https://github.com/Microsoft/TypeScript/releases/tag/v2.3.2) | 2017-04-28 |
| [2.3.1](https://github.com/Microsoft/TypeScript/releases/tag/v2.3.1) | 2017-04-25 |
| [2.3.0 beta](https://github.com/Microsoft/TypeScript/releases/tag/v2.3.0) 2017-04-04 | |
| [2.2.2](https://github.com/Microsoft/TypeScript/releases/tag/v2.2.2) | 2017-03-13 |
| [2.2](https://github.com/Microsoft/TypeScript/releases/tag/v2.2.1) | 2017-02-17 |
| [2.1.6](https://github.com/Microsoft/TypeScript/releases/tag/v2.1.6) | 2017-02-07 |
| [2.2 beta](https://github.com/Microsoft/TypeScript/releases/tag/v2.1.6) | 2017-02-02 |
| [2.1.5](https://github.com/Microsoft/TypeScript/releases/tag/v2.1.5) | 2017-01-05 |
| [2.1.4](https://github.com/Microsoft/TypeScript/releases/tag/v2.1.4) | 2016-12-05 |
| [2.0.8](https://github.com/Microsoft/TypeScript/releases/tag/v2.0.8) | 2016-11-08 |
| [2.0.7](https://github.com/Microsoft/TypeScript/releases/tag/v2.0.7) | 2016-11-03 |
| [2.0.6](https://github.com/Microsoft/TypeScript/releases/tag/v2.0.6) | 2016-10-23 |
| [2.0.5](https://github.com/Microsoft/TypeScript/releases/tag/v2.0.5) | 2016-09-22 |
| [2.0 Beta](https://github.com/Microsoft/TypeScript/releases/tag/v2.0.0-beta) | 2016-07-08 |
| [1.8.10](https://github.com/Microsoft/TypeScript/releases/tag/v1.8.10) | 2016-04-09 |
| [1.8.9](https://github.com/Microsoft/TypeScript/releases/tag/v1.8.9) | 2016-03-16 |
| [1.8.5](https://github.com/Microsoft/TypeScript/releases/tag/v1.8.5) | 2016-03-02 |
| [1.8.2](https://github.com/Microsoft/TypeScript/releases/tag/v1.8.2) | 2016-02-17 |
| [1.7.5](https://github.com/Microsoft/TypeScript/releases/tag/v1.7.5) | 2015-12-14 |
| [1.7](https://github.com/Microsoft/TypeScript/releases/tag/v1.7.3) | 2015-11-20 |
| [1.6](https://github.com/Microsoft/TypeScript/releases/tag/v1.6.2) | 2015-09-11 |
| [1.5.4](https://github.com/Microsoft/TypeScript/releases/tag/v1.5.4) | 2015-07-15 |
| [1.5](https://github.com/Microsoft/TypeScript/releases/tag/v1.5.3) | 2015-07-15 |
| [1.4](https://github.com/Microsoft/TypeScript/releases/tag/v1.4) | 2015-01-13 |
| [1.3](https://github.com/Microsoft/TypeScript/releases/tag/v1.3) | 2014-10-28 |
| [1.1.0.1](https://github.com/Microsoft/TypeScript/releases/tag/v1.1.0.1) | 2014-09-23 |

**Section 1.1: Installation and setup**

##### Background

TypeScript is a typed superset of JavaScript that compiles directly to JavaScript code. TypeScript ﬁles commonly use the .ts extension. Many IDEs support TypeScript without any other setup required, but TypeScript can also be compiled with the TypeScript Node.JS package from the command line.

##### IDEs

##### Visual Studio

Visual Studio 2015 includes TypeScript.

or later includes TypeScript, or you can [download TypeScript for earlier](https://www.microsoft.com/en-us/download/details.aspx?id=48739)

Visual Studio 2013 Update 2

[versions](https://www.microsoft.com/en-us/download/details.aspx?id=48739).

##### Visual Studio Code

[Visual Studio Code](https://code.visualstudio.com/) (vscode) provides contextual autocomplete as well as refactoring and debugging tools for TypeScript. vscode is itself implemented in TypeScript. Available for Mac OS X, Windows and Linux.

##### WebStorm

[WebStorm 2016.2](https://www.jetbrains.com/webstorm/) comes with TypeScript and a built-in compiler. [WebStorm is not free]

##### IntelliJ IDEA

[IntelliJ IDEA 2016.2](https://www.jetbrains.com/idea/) has support for TypeScript and a compiler via a [plugin](https://www.jetbrains.com/help/idea/2016.2/typescript-support.html) maintained by the JetBrains team. [IntelliJ is not free]

##### Atom & atom-typescript

[Atom](https://atom.io/) supports TypeScript with the [atom-typescript](https://atom.io/packages/atom-typescript) package.

##### Sublime Text

[Sublime Text](https://www.sublimetext.com/) supports TypeScript with the [TypeScript](https://github.com/Microsoft/TypeScript-Sublime-Plugin) package.

##### Installing the command line interface Install [Node.js](https://nodejs.org/)

##### Install the npm package globally

You can install TypeScript globally to have access to it from any directory.

npm **install** -g typescript

*or*

##### Install the npm package locally

You can install TypeScript locally and save to package.json to restrict to a directory.

npm **install** typescript --save-dev **Installation channels**

You can install from:

Stable channel: npm **install** typescript Beta channel: npm **install** typescript**@**beta Dev channel: npm **install** typescript**@**next

##### Compiling TypeScript code

The tsc compilation command comes with typescript, which can be used to compile code.

tsc my-code.ts

This creates a my-code.js ﬁle.

##### Compile using tsconﬁg.json

You can also provide compilation options that travel with your code via a [tsconfig.json](http://www.typescriptlang.org/docs/handbook/tsconfig-json.html) ﬁle. To start a new TypeScript project, cd into your project's root directory in a terminal window and run tsc --init. This command will generate a tsconfig.json ﬁle with minimal conﬁguration options, similar to below.

{

"compilerOptions": { "module": "commonjs", "target": "es5", "noImplicitAny": **false**, "sourceMap": **false**, "pretty": **true**

},

"exclude": [

"node\_modules"

]

}

With a tsconfig.json ﬁle placed at the root of your TypeScript project, you can use the tsc command to run the compilation.

**Section 1.2: Basic syntax**

TypeScript is a typed superset of JavaScript, which means that all JavaScript code is valid TypeScript code. TypeScript adds a lot of new features on top of that.

TypeScript makes JavaScript more like a strongly-typed, object-oriented language akin to C# and Java. This means that TypeScript code tends to be easier to use for large projects and that code tends to be easier to understand and maintain. The strong typing also means that the language can (and is) precompiled and that variables cannot be assigned values that are out of their declared range. For instance, when a TypeScript variable is declared as a number, you cannot assign a text value to it.

This strong typing and object orientation makes TypeScript easier to debug and maintain, and those were two of the weakest points of standard JavaScript.

##### Type declarations

You can add type declarations to variables, function parameters and function return types. The type is written after a colon following the variable name, like this: **var** num: number = 5; The compiler will then check the types (where possible) during compilation and report type errors.

**var** num: number = 5;

num = "this is a string"; *// error: Type 'string' is not assignable to type 'number'.*

The basic types are :

number (both integers and ﬂoating point numbers)

string

boolean

Array. You can specify the types of an array's elements. There are two equivalent ways to deﬁne array types:

Array<T> and T[]. For example: number[] - array of numbers Array<string> - array of strings

Tuples. Tuples have a ﬁxed number of elements with speciﬁc types.

[boolean, string] - tuple where the ﬁrst element is a boolean and the second is a string.

[number, number, number] - tuple of three numbers.

{} - object, you can deﬁne its properties or indexer

{name: string, age: number} - object with name and age attributes

- a dictionary of numbers indexed by string

{[key: string]: number}

enum - 0 - enumeration mapped to numbers

, Blue, Green }

{ Red =

Function. You specify types for the parameters and return value:

(param: number) => string - function taking one number parameter returning string

- function with no parameters returning an number.

() => number

- function taking a string and optionally a boolean with no return

(a: string, b?: boolean) => **void**

value.

any - Permits any type. Expressions involving any are not type checked.

**void** - represents "nothing", can be used as a function return value. Only **null** and **undefined** are part of the type.

**void**

never

**let** foo: never; -As the type of variables under type guards that are never true.

**function** error(message: string): never { **throw new** Error(message); } - As the return type of functions that never return.

**null** - type for the value **null**. **null** is implicitly part of every type, unless strict null checks are enabled.

##### Casting

You can perform explicit casting through angle brackets, for instance:

**var** derived: MyInterface; (<ImplementingClass>derived).someSpecificMethod();

This example shows a derived class which is treated by the compiler as a MyInterface. Without the casting on the second line the compiler would throw an exception as it does not understand someSpecificMethod(), but casting through **<ImplementingClass>**derived suggests the compiler what to do.

Another way of casting in TypeScript is using the as keyword:

**var** derived: MyInterface;

(derived as ImplementingClass).someSpecificMethod();

Since TypeScript 1.6, the default is using the as keyword, because using <**>** is ambiguous in **.jsx** ﬁles. This is mentioned in [TypeScript oﬃcial documentation](https://github.com/Microsoft/TypeScript/wiki/What%27s-new-in-TypeScript#new-tsx-file-extension-and-as-operator).

##### Classes

Classes can be deﬁned and used in TypeScript code. To learn more about classes, see the Classes documentation page.

**Section 1.3: Hello World**

class Greeter { greeting: string;

constructor(message: string) {

**this**.greeting = message;

}

greet(): string {

**return this**.greeting;

}

};

**let** greeter = **new** Greeter("Hello, world!"); console.log(greeter.greet());

Here we have a class, Greeter, that has a constructor and a greet method. We can construct an instance of the class using the **new** keyword and pass in a string we want the greet method to output to the console. The instance of our Greeter class is stored in the greeter variable which we then us to call the greet method.

**Section 1.4: Running TypeScript using ts-node**

[ts-node](https://www.npmjs.com/package/ts-node) is an npm package which allows the user to run typescript ﬁles directly, without the need for precompilation using tsc. It also provides [REPL](https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop).

Install ts-node globally using

npm **install** -g ts-node

ts-node does not bundle typescript compiler, so you might need to install it.

npm **install** -g typescript

##### Executing script

To execute a script named *main.ts*, run

ts-node main.ts

*// main.ts*

console.log("Hello world");

Example usage

$ ts-node main.ts Hello world

##### Running REPL

To run REPL run command ts-node

Example usage

$ ts-node

* **const** sum = (a, b): number => a + b;

**undefined**

> sum(2, 2)

4

* .exit

To exit REPL use command .exit or press CTRL+C twice.

**Section 1.5: TypeScript REPL in Node.js**

For use TypeScript REPL in Node.js you can use [tsun package](https://www.npmjs.com/package/tsun) Install it globally with

npm **install** -g tsun

and run in your terminal or command prompt with tsun command Usage example:

$ tsun

TSUN : TypeScript Upgraded Node

type **in** TypeScript expression to evaluate type :help **for** commands **in** repl

$ **function** multiply(x, y) {

..**return** x \* y;

..}

**undefined**

$ multiply(3, 4)

12

**Chapter 2: Why and when to use TypeScript**

If you ﬁnd the arguments for type systems persuasive in general, then you'll be happy with TypeScript.

It brings many of the advantages of type system (safety, readability, improved tooling) to the JavaScript ecosystem. It also suﬀers from some of the drawbacks of type systems (added complexity and incompleteness).

**Section 2.1: Safety**

TypeScript catches type errors early through static analysis:

**function** double(x: number): number {

**return** 2 \* x;

}

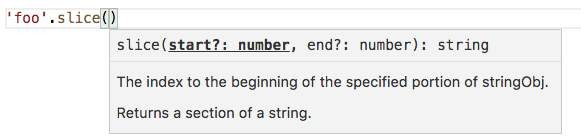
double('2');

*//*

*~~~ Argument of type '"2"' is not assignable to parameter of type 'number'.*

**Section 2.2: Readability**

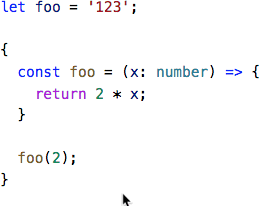
TypeScript enables editors to provide contextual documentation:



You'll never forget whether String.**prototype**.slice takes (start, stop) or (start, length) again!

**Section 2.3: Tooling**

TypeScript allows editors to perform automated refactors which are aware of the rules of the languages.



Here, for instance, Visual Studio Code is able to rename references to the inner foo without altering the outer foo. This would be diﬃcult to do with a simple ﬁnd/replace.

**Chapter 3: TypeScript Core Types**

**Section 3.1: String Literal Types**

String literal types allow you to specify the exact value a string can have.

**let** myFavoritePet: "dog"; myFavoritePet = "dog";

Any other string will give an error.

*// Error: Type '"rock"' is not assignable to type '"dog"'.*

*// myFavoritePet = "rock";*

Together with Type Aliases and Union Types you get a enum-like behavior.

type Species = "cat" | "dog" | "bird";

**function** buyPet(pet: Species, name: string) : Pet { */\*...\*/* } buyPet(myFavoritePet */\* "dog" as defined above \*/*, "Rocky");

*// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" | "bird". Type '"rock"' is not assignable to type '"bird"'.*

*// buyPet("rock", "Rocky");*

String Literal Types can be used to distinguish overloads.

**function** buyPet(pet: Species, name: string) : Pet; **function** buyPet(pet: "cat", name: string): Cat; **function** buyPet(pet: "dog", name: string): Dog; **function** buyPet(pet: "bird", name: string): Bird;

**function** buyPet(pet: Species, name: string) : Pet { */\*...\*/* }

**let** dog = buyPet(myFavoritePet */\* "dog" as defined above \*/*, "Rocky");

*// dog is from type Dog (dog: Dog)*

They works well for User-Deﬁned Type Guards.

interface Pet { species: Species; eat();

sleep();

}

interface Cat extends Pet { species: "cat";

}

interface Bird extends Pet { species: "bird";

sing();

}

**function** petIsCat(pet: Pet): pet is Cat {

**return** pet.species === "cat";

}

**function** petIsBird(pet: Pet): pet is Bird {

**return** pet.species === "bird";

}

**function** playWithPet(pet: Pet){

**if**(petIsCat(pet)) {

*// pet is now from type Cat (pet: Cat)*

pet.eat();

pet.sleep();

} **else if**(petIsBird(pet)) {

*// pet is now from type Bird (pet: Bird)*

pet.eat();

pet.sing();

pet.sleep();

}

}

Full example code

**let** myFavoritePet: "dog"; myFavoritePet = "dog";

*// Error: Type '"rock"' is not assignable to type '"dog"'.*

*// myFavoritePet = "rock";*

type Species = "cat" | "dog" | "bird"; interface Pet {

species: Species; name: string; eat();

walk();

sleep();

}

interface Cat extends Pet { species: "cat";

}

interface Dog extends Pet { species: "dog";

}

interface Bird extends Pet { species: "bird";

sing();

}

*// Error: Interface 'Rock' incorrectly extends interface 'Pet'. Types of property 'species' are incompatible. Type '"rock"' is not assignable to type '"cat" | "dog" | "bird"'. Type '"rock"' is not assignable to type '"bird"'.*

*// interface Rock extends Pet {*

*// type: "rock";*

*// }*

**function** buyPet(pet: Species, name: string) : Pet; **function** buyPet(pet: "cat", name: string): Cat; **function** buyPet(pet: "dog", name: string): Dog; **function** buyPet(pet: "bird", name: string): Bird; **function** buyPet(pet: Species, name: string) : Pet {

**if**(pet === "cat") {

**return** {

species: "cat", name: name,

eat: **function** () { console.log(`${**this**.name} eats.`);

}, walk: **function** () { console.log(`${**this**.name} walks.`);

}, sleep: **function** () { console.log(`${**this**.name} sleeps.`);

}

} as Cat;

} **else if**(pet === "dog") {

**return** {

species: "dog", name: name,

eat: **function** () { console.log(`${**this**.name} eats.`);

}, walk: **function** () { console.log(`${**this**.name} walks.`);

}, sleep: **function** () { console.log(`${**this**.name} sleeps.`);

}

} as Dog;

} **else if**(pet === "bird") {

**return** {

species: "bird", name: name,

eat: **function** () { console.log(`${**this**.name} eats.`);

}, walk: **function** () { console.log(`${**this**.name} walks.`);

}, sleep: **function** () { console.log(`${**this**.name} sleeps.`);

}, sing: **function** () { console.log(`${**this**.name} sings.`);

}

} as Bird;

} **else** {

**throw** `Sorry we **do** not have a ${pet}. Would you like to buy a dog?`;

}

}

**function** petIsCat(pet: Pet): pet is Cat {

**return** pet.species === "cat";

}

**function** petIsDog(pet: Pet): pet is Dog {

**return** pet.species === "dog";

}

**function** petIsBird(pet: Pet): pet is Bird {

**return** pet.species === "bird";

}

**function** playWithPet(pet: Pet) { console.log(`Hey ${pet.name}, lets play.`);

**if**(petIsCat(pet)) {

*// pet is now from type Cat (pet: Cat)*

pet.eat();

pet.sleep();

*// Error: Type '"bird"' is not assignable to type '"cat"'.*

*// pet.type = "bird";*

*// Error: Property 'sing' does not exist on type 'Cat'.*

*// pet.sing();*

} **else if**(petIsDog(pet)) {

*// pet is now from type Dog (pet: Dog)*

pet.eat();

pet.walk();

pet.sleep();

} **else if**(petIsBird(pet)) {

*// pet is now from type Bird (pet: Bird)*

pet.eat();

pet.sing();

pet.sleep();

} **else** {

**throw** "An unknown pet. Did you buy a rock?";

}

}

**let** dog = buyPet(myFavoritePet */\* "dog" as defined above \*/*, "Rocky");

*// dog is from type Dog (dog: Dog)*

*// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" | "bird". Type '"rock"' is not assignable to type '"bird"'.*

*// buyPet("rock", "Rocky");*

playWithPet(dog);

*// Output: Hey Rocky, lets play.*

*// Rocky eats.*

*// Rocky walks.*

*// Rocky sleeps.*

**Section 3.2: Tuple**

Array type with known and possibly diﬀerent types:

**let** day: [number, string];

day = [0, 'Monday']; *// valid*

day = ['zero', 'Monday']; *// invalid: 'zero' is not numeric*

console.log(day[0]); *// 0*

console.log(day[1]); *// Monday*

day[2] = 'Saturday'; *// valid: [0, 'Saturday']*

day[3] = **false**; *// invalid: must be union type of 'number | string'*

**Section 3.3: Boolean**

A boolean represents the most basic datatype in TypeScript, with the purpose of assigning true/false values.

*// set with initial value (either true or false)*

**let** isTrue: boolean = **true**;

*// defaults to 'undefined', when not explicitly set*

**let** unsetBool: boolean;

*// can also be set to 'null' as well*

**let** nullableBool: boolean = **null**;

**Section 3.4: Intersection Types**

A Intersection Type combines the member of two or more types.

interface Knife { cut();

}

interface BottleOpener{ openBottle();

}

interface Screwdriver{ turnScrew();

}

type SwissArmyKnife = Knife & BottleOpener & Screwdriver;

**function** use(tool: SwissArmyKnife){ console.log("I can do anything!");

tool.cut(); tool.openBottle(); tool.turnScrew();

}

**Section 3.5: Types in function arguments and return value. Number**

When you create a function in TypeScript you can specify the data type of the function's arguments and the data type for the return value

Example:

**function** sum(x: number, y: number): number {

**return** x + y;

}

Here the syntax x: number, y: number means that the function can accept two argumentsx and y and they can only be numbers and (...): number { means that the return value can only be a number

Usage:

sum(84 + 76) *// will be return 160*

Note:

You can not do so

**function** sum(x: string, y: string): number {

**return** x + y;

}

or

**function** sum(x: number, y: number): string {

**return** x + y;

}

it will receive the following errors:

error TS2322: Type 'string' is not assignable to type 'number' and error TS2322: Type 'number' is not assignable to type 'string' respectively

**Section 3.6: Types in function arguments and return value. String**

Example:

**function** hello(name: string): string {

**return** `Hello ${name}!`;

}

Here the syntax name: string means that the function can accept one name argument and this argument can only be string and (...): string { means that the return value can only be a string

Usage:

hello('StackOverflow Documentation') *// will be return Hello StackOverflow Documentation!*

**Section 3.7: const Enum**

A const Enum is the same as a normal Enum. Except that no Object is generated at compile time. Instead, the literal values are substituted where the const Enum is used.

*// TypeScript: A const Enum can be defined like a normal Enum (with start value, specific values, etc.)*

**const** enum NinjaActivity { Espionage,

Sabotage, Assassination

}

*// JavaScript: But nothing is generated*

*// TypeScript: Except if you use it*

**let** myFavoriteNinjaActivity = NinjaActivity.Espionage; console.log(myFavoritePirateActivity); *// 0*

*// JavaScript: Then only the number of the value is compiled into the code*

*// var myFavoriteNinjaActivity = 0 /\* Espionage \*/;*

*// console.log(myFavoritePirateActivity); // 0*

*// TypeScript: The same for the other constant example*

console.log(NinjaActivity["Sabotage"]); *// 1*

*// JavaScript: Just the number and in a comment the name of the value*

*// console.log(1 /\* "Sabotage" \*/); // 1*

*// TypeScript: But without the object none runtime access is possible*

*// Error: A const enum member can only be accessed using a string literal.*

*// console.log(NinjaActivity[myFavoriteNinjaActivity]);*

For comparison, a normal Enum

*// TypeScript: A normal Enum*

enum PirateActivity { Boarding, Drinking,

Fencing

}

*// JavaScript: The Enum after the compiling*

*// var PirateActivity;*

*// (function (PirateActivity) {*

*// PirateActivity[PirateActivity["Boarding"] = 0] = "Boarding";*

*// PirateActivity[PirateActivity["Drinking"] = 1] = "Drinking";*

*// PirateActivity[PirateActivity["Fencing"] = 2] = "Fencing";*

*// })(PirateActivity || (PirateActivity = {}));*

*// TypeScript: A normal use of this Enum*

**let** myFavoritePirateActivity = PirateActivity.Boarding; console.log(myFavoritePirateActivity); *// 0*

*// JavaScript: Looks quite similar in JavaScript*

*// var myFavoritePirateActivity = PirateActivity.Boarding;*

*// console.log(myFavoritePirateActivity); // 0*

*// TypeScript: And some other normal use*

console.log(PirateActivity["Drinking"]); *// 1*

*// JavaScript: Looks quite similar in JavaScript*

*// console.log(PirateActivity["Drinking"]); // 1*

*// TypeScript: At runtime, you can access an normal enum*

console.log(PirateActivity[myFavoritePirateActivity]); *// "Boarding"*

*// JavaScript: And it will be resolved at runtime*

*// console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"*

**Section 3.8: Number**

Like JavaScript, numbers are ﬂoating point values.

**let** pi: number = 3.14;

*// base 10 decimal by default*

**let** hexadecimal: number = 0xFF; *// 255 in decimal*

ECMAScript 2015 allows binary and octal.

**let** binary: number = 0b10; *// 2 in decimal*

**let** octal: number = 0o755; *// 493 in decimal*

**Section 3.9: String**

Textual data type:

**let** singleQuotes: string = 'single';

**let** doubleQuotes: string = "double";

**let** templateString: string = `I am ${ singleQuotes }`; *// I am single*

**Section 3.10: Array**

An array of values:

**let** threePigs: number[] = [1, 2, 3];

**let** genericStringArray: Array<string> = ['first', '2nd', '3rd'];

**Section 3.11: Enum**

A type to name a set of numeric values: Number values default to 0:

enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };

**let** bestDay: Day = Day.Saturday;

Set a default starting number:

enum TenPlus { Ten = 10, Eleven, Twelve }

or assign values:

enum MyOddSet { Three = 3, Five = 5, Seven = 7, Nine = 9 }

**Section 3.12: Any**

When unsure of a type, any is available:

**let** anything: any = 'I am a string'; anything = 5; *// but now I am the number 5*

**Section 3.13: Void**

If you have no type at all, commonly used for functions that do not return anything:

**function** log(): **void** { console.log('I return nothing');

}

**void** types Can only be assigned **null** or **undefined**.

**Chapter 4: Arrays**

# Section 4.1: Finding Object in Array

##### Using ﬁnd()

**const** inventory = [

{name: 'apples', quantity: 2},

{name: 'bananas', quantity: 0},

{name: 'cherries', quantity: 5}

];

**function** findCherries(fruit) {

**return** fruit.name === 'cherries';

}

inventory.find(findCherries); *// { name: 'cherries', quantity: 5 }*

*/\* OR \*/*

inventory.find(e => e.name === 'apples'); *// { name: 'apples', quantity: 2 }*

**Chapter 5: Enums**

# Section 5.1: Enums with explicit values

By default all enum values are resolved to numbers. Let's say if you have something like

enum MimeType { JPEG,

PNG, PDF

}

the real value behind e.g. MimeType.PDF will be 2.

But some of the time it is important to have the enum resolve to a diﬀerent type. E.g. you receive the value from backend / frontend / another system which is deﬁnitely a string. This could be a pain, but luckily there is this method:

enum MimeType {

JPEG = <any>'image/jpeg', PNG = <any>'image/png',

PDF = <any>'application/pdf'

}

This resolves the MimeType.PDF to application/pdf. Since TypeScript 2.4 it's possible to declare [string enums](https://blogs.msdn.microsoft.com/typescript/2017/06/27/announcing-typescript-2-4/#string-enums):

enum MimeType {

JPEG = 'image/jpeg', PNG = 'image/png',

PDF = 'application/pdf',

}

You can explicitly provide numeric values using the same method

enum MyType { Value = 3,

ValueEx = 30,

ValueEx2 = 300

}

Fancier types also work, since non-const enums are real objects at runtime, for example

enum FancyType { OneArr = <any>[1],

TwoArr = <any>[2, 2], ThreeArr = <any>[3, 3, 3]

}

becomes

**var** FancyType; (**function** (FancyType) {

FancyType[FancyType["OneArr"] = [1]] = "OneArr"; FancyType[FancyType["TwoArr"] = [2, 2]] = "TwoArr";

FancyType[FancyType["ThreeArr"] = [3, 3, 3]] = "ThreeArr";

})(FancyType || (FancyType = {}));

# Section 5.2: How to get all enum values

enum SomeEnum { A, B }

**let** enumValues:Array<string>= [];

**for**(**let** value **in** SomeEnum) {

**if**(**typeof** SomeEnum[value] === 'number') { enumValues.push(value);

}

}

enumValues.forEach(v=> console.log(v))

*//A*

*//B*

# Section 5.3: Extending enums without custom enum implementation

enum SourceEnum {

value1 = <any>'value1', value2 = <any>'value2'

}

enum AdditionToSourceEnum { value3 = <any>'value3', value4 = <any>'value4'

}

*// we need this type for TypeScript to resolve the types correctly*

type TestEnumType = SourceEnum | AdditionToSourceEnum;

*// and we need this value "instance" to use values*

**let** TestEnum = Object.assign({}, SourceEnum, AdditionToSourceEnum);

*// also works fine the TypeScript 2 feature*

*// let TestEnum = { ...SourceEnum, ...AdditionToSourceEnum };*

**function** check(test: TestEnumType) {

**return** test === TestEnum.value2;

}

console.log(TestEnum.value1); console.log(TestEnum.value2 === <any>'value2'); console.log(check(TestEnum.value2)); console.log(check(TestEnum.value3));

# Section 5.4: Custom enum implementation: extends for enums

Sometimes it is required to implement Enum on your own. E.g. there is no clear way to extend other enums. Custom implementation allows this:

class Enum {

constructor(protected value: string) {}

public toString() {

**return** String(**this**.value);

}

public is(value: Enum | string) {

**return this**.value = value.toString();

}

}

class SourceEnum extends Enum {

public **static** value1 = **new** SourceEnum('value1'); public **static** value2 = **new** SourceEnum('value2');

}

class TestEnum extends SourceEnum {

public **static** value3 = **new** TestEnum('value3'); public **static** value4 = **new** TestEnum('value4');

}

**function** check(test: TestEnum) {

**return** test === TestEnum.value2;

}

**let** value1 = TestEnum.value1;

console.log(value1 + 'hello'); console.log(value1.toString() === 'value1'); console.log(value1.is('value1')); console.log(!TestEnum.value3.is(TestEnum.value3)); console.log(check(TestEnum.value2));

*// this works but perhaps your TSLint would complain*

*// attention! does not work with ===*

*// use .is() instead*

console.log(TestEnum.value1 == <any>'value1');

**Chapter 6: Functions**

**Section 6.1: Optional and Default Parameters**

##### Optional Parameters

In TypeScript, every parameter is assumed to be required by the function. You can add a **?** at the end of a parameter name to set it as optional.

For example, the lastName parameter of this function is optional:

**function** buildName(firstName: string, lastName?: string) {

*// ...*

}

Optional parameters must come after all non-optional parameters:

**function** buildName(firstName?: string, lastName: string) *// Invalid*

##### Default Parameters

If the user passes **undefined** or doesn't specify an argument, the default value will be assigned. These are called

*default-initialized* parameters.

For example, "Smith" is the default value for the lastName parameter.

**function** buildName(firstName: string, lastName = "Smith") {

*// ...*

}

buildName('foo', 'bar'); *// firstName == 'foo', lastName == 'bar'* buildName('foo'); *// firstName == 'foo', lastName == 'Smith'* buildName('foo', **undefined**); *// firstName == 'foo', lastName == 'Smith'*

**Section 6.2: Function as a parameter**

Suppose we want to receive a function as a parameter, we can do it like this:

**function** foo(otherFunc: Function): **void** {

...

}

If we want to receive a constructor as a parameter:

**function** foo(constructorFunc: { **new**() }) {

**new** constructorFunc();

}

**function** foo(constructorWithParamsFunc: { **new**(num: number) }) {

**new** constructorWithParamsFunc(1);

}

Or to make it easier to read we can deﬁne an interface describing the constructor:

interface IConstructor {

**new**();

}

**function** foo(contructorFunc: IConstructor) {

**new** constructorFunc();

}

Or with parameters:

interface INumberConstructor {

**new**(num: number);

}

**function** foo(contructorFunc: INumberConstructor) {

**new** contructorFunc(1);

}

Even with generics:

interface ITConstructor<T, U> {

**new**(item: T): U;

}

**function** foo<T, U>(contructorFunc: ITConstructor<T, U>, item: T): U {

**return new** contructorFunc(item);

}

If we want to receive a simple function and not a constructor it's almost the same:

**function** foo(func: { (): **void** }) { func();

}

**function** foo(constructorWithParamsFunc: { (num: number): **void** }) {

**new** constructorWithParamsFunc(1);

}

Or to make it easier to read we can deﬁne an interface describing the function:

interface IFunction { (): **void**;

}

**function** foo(func: IFunction ) { func();

}

Or with parameters:

interface INumberFunction { (num: number): string;

}

**function** foo(func: INumberFunction ) { func(1);

}

Even with generics:

interface ITFunc<T, U> { (item: T): U;

}

**function** foo<T, U>(contructorFunc: ITFunc<T, U>, item: T): U {

**return** func(item);

}

**Section 6.3: Functions with Union Types**

A TypeScript function can take in parameters of multiple, predeﬁned types using union types.

**function** whatTime(hour:number|string, minute:number|string):string{

**return** hour+':'+minute;

}

whatTime(1,30) whatTime('1',30)

whatTime(1,'30')

whatTime('1','30')

*//'1:30'*

*//'1:30'*

*//'1:30'*

*//'1:30'*

TypeScript treats these parameters as a single type that is a union of the other types, so your function must be able to handle parameters of any type that is in the union.

**function** addTen(start:number|string):number{

**if**(**typeof** number === 'string'){

**return** parseInt(number)+10;

}**else**{

**else return** number+10;

}

}

**Section 6.4: Types of Functions**

##### Named functions

**function** multiply(a, b) {

**return** a \* b;

}

##### Anonymous functions

**let** multiply = **function**(a, b) { **return** a \* b; };

##### Lambda / arrow functions

**let** multiply = (a, b) => { **return** a \* b; };

**Chapter 7: Classes**

TypeScript, like ECMAScript 6, support object-oriented programming using classes. This contrasts with older JavaScript versions, which only supported prototype-based inheritance chain.

The class support in TypeScript is similar to that of languages like Java and C#, in that classes may inherit from other classes, while objects are instantiated as class instances.

Also similar to those languages, TypeScript classes may implement interfaces or make use of generics.

**Section 7.1: Abstract Classes**

abstract class Machine {

constructor(public manufacturer: string) {

}

*// An abstract class can define methods of its own, or...*

summary(): string {

**return** `${**this**.manufacturer} makes **this** machine.`;

}

*// Require inheriting classes to implement methods*

abstract moreInfo(): string;

}

class Car extends Machine {

constructor(manufacturer: string, public position: number, protected speed: number) { super(manufacturer);

}

move() {

**this**.position += **this**.speed;

}

moreInfo() {

**return** `This is a car located at ${**this**.position} and going ${**this**.speed}mph!`;

}

}

**let** myCar = **new** Car("Konda", 10, 70); myCar.move(); *// position is now 80*

console.log(myCar.summary()); *// prints "Konda makes this machine."*

console.log(myCar.moreInfo()); *// prints "This is a car located at 80 and going 70mph!"*

Abstract classes are base classes from which other classes can extend. They cannot be instantiated themselves (i.e. you **cannot** do **new** Machine("Konda")).

The two key characteristics of an abstract class in TypeScript are:

1. They can implement methods of their own.
2. They can deﬁne methods that inheriting classes **must** implement.

For this reason, abstract classes can conceptually be considered a **combination of an interface and a class**.

**Section 7.2: Simple class**

class Car {

public position: number = 0; private speed: number = 42;

move() {

**this**.position += **this**.speed;

}

}

In this example, we declare a simple class Car. The class has three members: a private property speed, a public property position and a public method move. Note that each member is public by default. That's why move() is public, even if we didn't use the public keyword.

**var** car = **new** Car(); *// create an instance of Car* car.move(); *// call a method* console.log(car.position); *// access a public property*

**Section 7.3: Basic Inheritance**

class Car {

public position: number = 0; protected speed: number = 42;

move() {

**this**.position += **this**.speed;

}

}

class SelfDrivingCar extends Car { move() {

*// start moving around :-)*

super.move();

super.move();

}

}

This examples shows how to create a very simple subclass of the Car class using the extends keyword. The

SelfDrivingCar class overrides the move() method and uses the base class implementation using super.

**Section 7.4: Constructors**

In this example we use the constructor to declare a public property position and a protected property speed in the base class. These properties are called *Parameter properties*. They let us declare a constructor parameter and a member in one place.

One of the best things in TypeScript, is automatic assignment of constructor parameters to the relevant property.

class Car {

public position: number; protected speed: number;

constructor(position: number, speed: number) {

**this**.position = position;

**this**.speed = speed;

}

move() {

**this**.position += **this**.speed;

}

}

All this code can be resumed in one single constructor:

class Car {

constructor(public position: number, protected speed: number) {}

move() {

**this**.position += **this**.speed;

}

}

And both of them will be transpiled from TypeScript (design time and compile time) to JavaScript with same result, but writing signiﬁcantly less code:

**var** Car = (**function** () {

**function** Car(position, speed) { **this**.position = position; **this**.speed = speed;

}

Car.**prototype**.move = **function** () {

**this**.position += **this**.speed;

};

**return** Car;

}());

Constructors of derived classes have to call the base class constructor with super().

class SelfDrivingCar extends Car { constructor(startAutoPilot: boolean) {

super(0, 42);

**if** (startAutoPilot) {

**this**.move();

}

}

}

**let** car = **new** SelfDrivingCar(**true**);

console.log(car.position); *// access the public property position*

**Section 7.5: Accessors**

In this example, we modify the "Simple class" example to allow access to the speed property. TypeScript accessors allow us to add additional code in getters or setters.

class Car {

public position: number = 0; private \_speed: number = 42; private \_MAX\_SPEED = 100

move() {

**this**.position += **this**.\_speed;

}

**get** speed(): number {

**return this**.\_speed;

}

**set** speed(value: number) {

**this**.\_speed = Math.min(value, **this**.\_MAX\_SPEED);

}

}

**let** car = **new** Car(); car.speed = 120;

console.log(car.speed); *// 100*

**Section 7.6: Transpilation**

Given a class SomeClass, let's see how the TypeScript is transpiled into JavaScript.

##### TypeScript source

class SomeClass {

public **static** SomeStaticValue: string = "hello"; public someMemberValue: number = 15;

private somePrivateValue: boolean = **false**;

constructor () {

SomeClass.SomeStaticValue = SomeClass.getGoodbye(); **this**.someMemberValue = **this**.getFortyTwo(); **this**.somePrivateValue = **this**.getTrue();

}

public **static** getGoodbye(): string {

**return** "goodbye!";

}

public getFortyTwo(): number {

**return** 42;

}

private getTrue(): boolean {

**return true**;

}

}

##### JavaScript source

When transpiled using TypeScript v2.2.2, the output is like so:

**var** SomeClass = (**function** () {

**function** SomeClass() { **this**.someMemberValue = 15; **this**.somePrivateValue = **false**;

SomeClass.SomeStaticValue = SomeClass.getGoodbye(); **this**.someMemberValue = **this**.getFortyTwo(); **this**.somePrivateValue = **this**.getTrue();

}

SomeClass.getGoodbye = **function** () {

**return** "goodbye!";

};

SomeClass.**prototype**.getFortyTwo = **function** () {

**return** 42;

};

SomeClass.**prototype**.getTrue = **function** () {

**return true**;

};

**return** SomeClass;

}());

SomeClass.SomeStaticValue = "hello";

##### Observations

The modiﬁcation of the class' prototype is wrapped inside an [IIFE](https://www.wikiwand.com/en/Immediately-invoked_function_expression). Member variables are deﬁned inside the main class **function**.

Static properties are added directly to the class object, whereas instance properties are added to the prototype.

**Section 7.7: Monkey patch a function into an existing class**

Sometimes it's useful to be able to extend a class with new functions. For example let's suppose that a string should be converted to a camel case string. So we need to tell TypeScript, that String contains a function called toCamelCase, which returns a string.

interface String { toCamelCase(): string;

}

Now we can patch this function into the String implementation.

String.**prototype**.toCamelCase = **function**() : string {

**return this**.replace(*/[^a-z ]/ig*, '')

.replace(*/(?:^\w|[A-Z]|\b\w|\s+)/g*, (match: any, index: number) => {

**return** +match === 0 ? "" : match[index === 0 ? 'toLowerCase' : 'toUpperCase']();

});

}

If this extension of String is loaded, it's usable like this:

"This is an example".toCamelCase();

*// => "thisIsAnExample"*

**Chapter 8: Class Decorator**

##### Parameter Details

target The class being decorated

**Section 8.1: Generating metadata using a class decorator**

This time we are going to declare a class decorator that will add some metadata to a class when we applied to it:

**function** addMetadata(target: any) {

*// Add some metadata*

target. customMetadata = { someKey: "someValue"

};

*// Return target*

**return** target;

}

We can then apply the class decorator:

@addMetadata class Person {

private \_name: string;

public constructor(name: string) {

**this**.\_name = name;

}

public greet() {

**return this**.\_name;

}

}

**function** getMetadataFromClass(target: any) {

**return** target. customMetadata;

}

console.log(getMetadataFromClass(Person));

The decorator is applied when the class is declared not when we create instances of the class. This means that the metadata is shared across all the instances of a class:

**function** getMetadataFromInstance(target: any) {

**return** target.constructor. customMetadata;

}

**let** person1 = **new** Person("John");

**let** person2 = **new** Person("Lisa");

console.log(getMetadataFromInstance(person1)); console.log(getMetadataFromInstance(person2));

**Section 8.2: Passing arguments to a class decorator**

We can wrap a class decorator with another function to allow customization:

**function** addMetadata(metadata: any) {

**return function** log(target: any) {

*// Add metadata*

target. customMetadata = metadata;

*// Return target*

**return** target;

}

}

The addMetadata takes some arguments used as conﬁguration and then returns an unnamed function which is the actual decorator. In the decorator we can access the arguments because there is a closure in place.

We can then invoke the decorator passing some conﬁguration values:

@addMetadata({ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" }) class Person {

private \_name: string;

public constructor(name: string) {

**this**.\_name = name;

}

public greet() {

**return this**.\_name;

}

}

We can use the following function to access the generated metadata:

**function** getMetadataFromClass(target: any) {

**return** target. customMetadata;

}

console.log(getMetadataFromInstance(Person));

If everything went right the console should display:

{ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" }

**Section 8.3: Basic class decorator**

A class decorator is just a function that takes the class as its only argument and returns it after doing something with it:

**function** log<T>(target: T) {

*// Do something with target*

console.log(target);

*// Return target*

**return** target;

}

We can then apply the class decorator to a class:

@log

class Person {

private \_name: string;

public constructor(name: string) {

**this**.\_name = name;

}

public greet() {

**return this**.\_name;

}

}

**Chapter 9: Interfaces**

An interfaces speciﬁes a list of ﬁelds and functions that may be expected on any class implementing the interface. Conversely, a class cannot implement an interface unless it has every ﬁeld and function speciﬁed on the interface.

The primary beneﬁt of using interfaces, is that it allows one to use objects of diﬀerent types in a polymorphic way. This is because any class implementing the interface has at least those ﬁelds and functions.

**Section 9.1: Extending Interface**

Suppose we have an interface:

interface IPerson { name: string; age: number;

breath(): **void**;

}

And we want to create more speciﬁc interface that has the same properties of the person, we can do it using the

extends keyword:

interface IManager extends IPerson { managerId: number;

managePeople(people: IPerson[]): **void**;

}

In addition it is possible to extend multiple interfaces.

**Section 9.2: Class Interface**

Declare public variables and methods type in the interface to deﬁne how other typescript code can interact with it.

interface ISampleClassInterface { sampleVariable: string;

sampleMethod(): **void**; optionalVariable?: string;

}

Here we create a class that implements the interface.

class SampleClass implements ISampleClassInterface { public sampleVariable: string;

private answerToLifeTheUniverseAndEverything: number;

constructor() {

**this**.sampleVariable = 'string value';

**this**.answerToLifeTheUniverseAndEverything = 42;

}

public sampleMethod(): **void** {

*// do nothing*

}

private answer(q: any): number {

**return this**.answerToLifeTheUniverseAndEverything;

}

}

The example shows how to create an interface ISampleClassInterface and a class SampleClass that implements

the interface.

**Section 9.3: Using Interfaces for Polymorphism**

The primary reason to use interfaces to achieve polymorphism and provide developers to implement on their own way in future by implementing interface's methods.

Suppose we have an interface and three classes:

interface Connector{ doConnect(): boolean;

}

This is connector interface. Now we will implement that for Wiﬁ communication.

export class WifiConnector implements Connector{

public doConnect(): boolean{ console.log("Connecting via wifi"); console.log("Get password"); console.log("Lease an IP for 24 hours"); console.log("Connected");

**return true**

}

}

Here we have developed our concrete class named WifiConnector that has its own implementation. This is now type Connector.

Now we are creating our System that has a component Connector. This is called dependency injection.

export class System {

constructor(private connector: Connector){ #inject Connector type connector.doConnect()

}

}

constructor(private connector: Connector) this line is very important here. Connector is an interface and must have doConnect(). As Connector is an interface this class System has much more ﬂexibility. We can pass any Type which has implemented Connector interface. In future developer achieves more ﬂexibility. For example, now developer want to add Bluetooth Connection module:

export class BluetoothConnector implements Connector{

public doConnect(): boolean{ console.log("Connecting via Bluetooth"); console.log("Pair with PIN"); console.log("Connected");

**return true**

}

}

See that Wiﬁ and Bluetooth have its own implementation. Their own diﬀerent way to connect. However, hence both have implemented Type Connector the are now Type Connector. So that we can pass any of those to System class as the constructor parameter. This is called polymorphism. The class System is now not aware of whether it is Bluetooth / Wiﬁ even we can add another Communication module like Infrared, Bluetooth5 and whatsoever by just implementing Connector interface.

This is called [Duck typing](https://en.wikipedia.org/wiki/Duck_typing). Connector type is now dynamic as doConnect() is just a placeholder and developer implement this as his/her own.

if at constructor(private connector: WifiConnector) where WifiConnector is a concrete class what will happen? Then System class will tightly couple only with WiﬁConnector nothing else. Here interface solved our problem by polymorphism.

**Section 9.4: Generic Interfaces**

Like classes, interfaces can receive polymorphic parameters (aka Generics) too.

##### Declaring Generic Parameters on Interfaces

interface IStatus<U> { code: U;

}

interface IEvents<T> { list: T[]; emit(event: T): **void**; getAll(): T[];

}

Here, you can see that our two interfaces take some generic parameters, **T** and **U**. **Implementing Generic Interfaces**

We will create a simple class in order to implements the interface **IEvents**.

class State<T> implements IEvents<T> { list: T[];

constructor() {

**this**.list = [];

}

emit(event: T): **void** {

**this**.list.push(event);

}

getAll(): T[] {

**return this**.list;

}

}

Let's create some instances of our **State** class.

In our example, the State class will handle a generic status by using IStatus<T>. In this way, the interface

IEvent<T> will also handle a IStatus<T>.

**const** s = **new** State<IStatus<number>>();

*// The 'code' property is expected to be a number, so:*

s.emit({ code: 200 }); *// works*

s.emit({ code: '500' }); *// type error*

s.getAll().forEach(event => console.log(event.code));

Here our State class is typed as IStatus<number>.

**const** s2 = **new** State<IStatus<Code>>();

*//We are able to emit code as the type Code*

s2.emit({ code: { message: 'OK', status: 200 } });

s2.getAll().map(event => event.code).forEach(event => { console.log(event.message); console.log(event.status);

});

Our State class is typed as IStatus<Code>. In this way, we are able to pass more complex type to our emit method. As you can see, generic interfaces can be a very useful tool for statically typed code.

**Section 9.5: Add functions or properties to an existing interface**

Let's suppose we have a reference to the JQuery type deﬁnition and we want to extend it to have additional functions from a plugin we included and which doesn't have an oﬃcial type deﬁnition. We can easily extend it by declaring functions added by plugin in a separate interface declaration with the same JQuery name:

interface JQuery { pluginFunctionThatDoesNothing(): **void**;

*// create chainable function*

manipulateDOM(HTMLElement): JQuery;

}

The compiler will merge all declarations with the same name into one - see [declaration merging](https://www.typescriptlang.org/docs/handbook/declaration-merging.html) for more details.

**Section 9.6: Implicit Implementation And Object Shape**

TypeScript supports interfaces, but the compiler outputs JavaScript, which doesn't. Therefore, interfaces are eﬀectively lost in the compile step. This is why type checking on interfaces relies on the *shape* of the object - meaning whether the object supports the ﬁelds and functions on the interface - and not on whether the interface is actually implemented or not.

interface IKickable { kick(distance: number): **void**;

}

class Ball {

kick(distance: number): **void** { console.log("Kicked", distance, "meters!");

}

}

**let** kickable: IKickable = **new** Ball(); kickable.kick(40);

So even if Ball doesn't explicitly implement IKickable, a Ball instance may be assigned to (and manipulated as) an

IKickable, even when the type is speciﬁed.

**Section 9.7: Using Interfaces to Enforce Types**

One of the core beneﬁts of TypeScript is that it enforces data types of values that you are passing around your code to help prevent mistakes.

Let's say you're making a pet dating application.

You have this simple function that checks if two pets are compatible with each other...

checkCompatible(petOne, petTwo) {

**if** (petOne.species === petTwo.species && Math.abs(petOne.age - petTwo.age) <= 5) {

**return true**;

}

}

This is completely functional code, but it would be far too easy for someone, especially other people working on this application who didn't write this function, to be unaware that they are supposed to pass it objects with 'species' and 'age' properties. They may mistakenly try checkCompatible(petOne.species, petTwo.species) and then be left to ﬁgure out the errors thrown when the function tries to access petOne.species.species or petOne.species.age!

One way we can prevent this from happening is to specify the properties we want on the pet parameters:

checkCompatible(petOne: {species: string, age: number}, petTwo: {species: string, age: number}) {

*//...*

}

In this case, TypeScript will make sure everything passed to the function has 'species' and 'age' properties (it is okay if they have additional properties), but this is a bit of an unwieldy solution, even with only two properties speciﬁed. With interfaces, there is a better way!

First we deﬁne our interface:

interface Pet { species: string; age: number;

*//We can add more properties if we choose.*

}

Now all we have to do is specify the type of our parameters as our new interface, like so...

checkCompatible(petOne: Pet, petTwo: Pet) {

*//...*

}

... and TypeScript will make sure that the parameters passed to our function contain the properties speciﬁed in the Pet interface!

**Chapter 10: Generics**

**Section 10.1: Generic Interfaces**

##### Declaring a generic interface

interface IResult<T> { wasSuccessful: boolean; error: T;

}

**var** result: IResult<string> = ....

**var** error: string = result.error;

##### Generic interface with multiple type parameters

interface IRunnable<T, U> { run(input: T): U;

}

**var** runnable: IRunnable<string, number> = ...

**var** input: string;

**var** result: number = runnable.run(input);

##### Implementing a generic interface

interface IResult<T>{ wasSuccessful: boolean; error: T;

clone(): IResult<T>;

}

Implement it with generic class:

class Result<T> implements IResult<T> {

constructor(public result: boolean, public error: T) {

}

public clone(): IResult<T> {

**return new** Result<T>(**this**.result, **this**.error);

}

}

Implement it with non generic class:

class StringResult implements IResult<string> { constructor(public result: boolean, public error: string) {

}

public clone(): IResult<string> {

**return new** StringResult(**this**.result, **this**.error);

}

}

**Section 10.2: Generic Class**

class Result<T> {

constructor(public wasSuccessful: boolean, public error: T) {

}

public clone(): Result<T> {

...

}

}

**let** r1 = **new** Result(**false**, 'error: 42'); *// Compiler infers T to string* **let** r2 = **new** Result(**false**, 42); *// Compiler infers T to number* **let** r3 = **new** Result<string>(**true**, **null**); *// Explicitly set T to string*

**let** r4 = **new** Result<string>(**true**, 4); *// Compilation error because 4 is not a string*

**Section 10.3: Type parameters as constraints**

With TypeScript 1.8 it becomes possible for a type parameter constraint to reference type parameters from the same type parameter list. Previously this was an error.

**function** assign<T extends U, U>(target: T, source: U): T {

**for** (**let** id **in** source) { target[id] = source[id];

}

**return** target;

}

**let** x = { a: 1, b: 2, c: 3, d: 4 };

assign(x, { b: 10, d: 20 });

assign(x, { e: 0 }); *// Error*

**Section 10.4: Generics Constraints**

Simple constraint:

interface IRunnable { run(): **void**;

}

interface IRunner<T extends IRunnable> { runSafe(runnable: T): **void**;

}

More complex constraint:

interface IRunnble<U> { run(): U;

}

interface IRunner<T extends IRunnable<U>, U> { runSafe(runnable: T): U;

}

Even more complex:

interface IRunnble<V> { run(parameter: U): V;

}

interface IRunner<T extends IRunnable<U, V>, U, V> {

runSafe(runnable: T, parameter: U): V;

}

Inline type constraints:

interface IRunnable<T extends { run(): **void** }> { runSafe(runnable: T): **void**;

}

**Section 10.5: Generic Functions**

In interfaces:

interface IRunner {

runSafe<T extends IRunnable>(runnable: T): **void**;

}

In classes:

class Runner implements IRunner {

public runSafe<T extends IRunnable>(runnable: T): **void** {

**try** {

runnable.run();

} **catch**(e) {

}

}

}

Simple functions:

**function** runSafe<T extends IRunnable>(runnable: T): **void** {

**try** {

runnable.run();

} **catch**(e) {

}

}

**Section 10.6: Using generic Classes and Functions:**

Create generic class instance:

**var** stringRunnable = **new** Runnable<string>();

Run generic function:

**function** runSafe<T extends Runnable<U>, U>(runnable: T);

*// Specify the generic types:*

runSafe<Runnable<string>, string>(stringRunnable);

*// Let typescript figure the generic types by himself:*

runSafe(stringRunnable);

**Chapter 11: Strict null checks**

**Section 11.1: Strict null checks in action**

By default, all types in TypeScript allow **null**:

**function** getId(x: Element) {

**return** x.id;

}

getId(**null**); *// TypeScript does not complain, but this is a runtime error.*

TypeScript 2.0 adds support for strict null checks. If you set --strictNullChecks when running tsc (or set this ﬂag in your tsconfig.json), then types no longer permit **null**:

**function** getId(x: Element) {

**return** x.id;

}

getId(**null**); *// error: Argument of type 'null' is not assignable to parameter of type 'Element'.*

You must permit **null** values explicitly:

**function** getId(x: Element|**null**) {

**return** x.id; *// error TS2531: Object is possibly 'null'.*

}

getId(**null**);

With a proper guard, the code type checks and runs correctly:

**function** getId(x: Element|**null**) {

**if** (x) {

**return** x.id; *// In this branch, x's type is Element*

} **else** {

**return null**; *// In this branch, x's type is null.*

}

}

getId(**null**);

**Section 11.2: Non-null assertions**

The non-null assertion operator, !, allows you to assert that an expression isn't **null** or **undefined** when the TypeScript compiler can't infer that automatically:

type ListNode = { data: number; next?: ListNode; };

**function** addNext(node: ListNode) {

**if** (node.next === **undefined**) { node.next = {data: 0};

}

}

**function** setNextValue(node: ListNode, value: number) { addNext(node);

*// Even though we know `node.next` is defined because we just called `addNext`,*

*// TypeScript isn't able to infer this in the line of code below:*

*// node.next.data = value;*

*// So, we can use the non-null assertion operator, !,*

*// to assert that node.next isn't undefined and silence the compiler warning*

node.next!.data = value;

}

**Chapter 12: User-deﬁned Type Guards**

**Section 12.1: Type guarding functions**

You can declare functions that serve as type guards using any logic you'd like. They take the form:

**function** functionName(variableName: any): variableName is DesiredType {

*// body that returns boolean*

}

If the function returns true, TypeScript will narrow the type to DesiredType in any block guarded by a call to the function.

For example ([try it](https://goo.gl/xV4pLK)):

**function** isString(test: any): test is string {

**return typeof** test === "string";

}

**function** example(foo: any) {

**if** (isString(foo)) {

*// foo is type as a string in this block*

console.log("it's a string: " + foo);

} **else** {

*// foo is type any in this block*

console.log("don't know what this is! [" + foo + "]");

}

}

example("hello world");

*// prints "it's a string: hello world"*

example({ something: "else" }); *// prints "don't know what this is! [[object Object]]"*

A guard's function type predicate (the foo is Bar in the function return type position) is used at compile time to narrow types, the function body is used at runtime. The type predicate and function must agree, or your code won't work.

Type guard functions don't have to use **typeof** or **instanceof**, they can use more complicated logic. For example, this code determines if you've got a jQuery object by checking for its version string.

**function** isJQuery(foo): foo is JQuery {

*// test for jQuery's version string*

**return** foo.jquery !== **undefined**;

}

**function** example(foo) {

**if** (isJQuery(foo)) {

*// foo is typed JQuery here*

foo.eq(0);

}

}

**Section 12.2: Using instanceof**

**instanceof** requires that the variable is of type any. This code ([try it](https://goo.gl/p7Ywos)):

class Pet { }

class Dog extends Pet { bark() {

console.log("woof");

}

}

class Cat extends Pet { purr() {

console.log("meow");

}

}

**function** example(foo: any) {

**if** (foo **instanceof** Dog) {

*// foo is type Dog in this block*

foo.bark();

}

**if** (foo **instanceof** Cat) {

*// foo is type Cat in this block*

foo.purr();

}

}

example(**new** Dog()); example(**new** Cat());

prints

woof meow

to the console.

**Section 12.3: Using typeof**

**typeof** is used when you need to distinguish between types number, string, boolean, and symbol. Other string constants will not error, but won't be used to narrow types either.

Unlike **instanceof**, **typeof** will work with a variable of any type. In the example below, foo could be typed as number

| string without issue.

This code ([try it](https://goo.gl/a9zg07)):

**function** example(foo: any) {

**if** (**typeof** foo === "number") {

*// foo is type number in this block*

console.log(foo + 100);

}

**if** (**typeof** foo === "string") {

*// foo is type string in this block*

console.log("not a number: " + foo);

}

}

example(23); example("foo");

prints

123

not a number: foo

**Chapter 13: TypeScript basic examples**

**Section 13.1: 1 basic class inheritance example using extends and super keyword**

A generic Car class has some car property and a description method

class Car{

name:string; engineCapacity:string;

constructor(name:string,engineCapacity:string){

**this**.name = name;

**this**.engineCapacity = engineCapacity;

}

describeCar(){

console.log(`${**this**.name} car comes with ${**this**.engineCapacity} displacement`);

}

}

**new** Car("maruti ciaz","1500cc").describeCar();

HondaCar extends the existing generic car class and adds new property.

class HondaCar extends Car{ seatingCapacity:number;

constructor(name:string,engineCapacity:string,seatingCapacity:number){ super(name,engineCapacity);

**this**.seatingCapacity=seatingCapacity;

}

describeHondaCar(){ super.describeCar();

console.log(`**this** cars comes with seating capacity of ${**this**.seatingCapacity}`);

}

}

**new** HondaCar("honda jazz","1200cc",4).describeHondaCar();

**Section 13.2: 2 static class variable example - count how many time method is being invoked**

here countInstance is a static class variable

class StaticTest{

**static** countInstance : number= 0; constructor(){

StaticTest.countInstance++;

}

}

**new** StaticTest();

**new** StaticTest(); console.log(StaticTest.countInstance);

**Chapter 14: Importing external libraries**

**Section 14.1: Finding deﬁnition ﬁles**

for typescript 2.x:

deﬁnitions from [DeﬁnitelyTyped](https://github.com/DefinitelyTyped/DefinitelyTyped) are available via [@types npm](https://www.npmjs.com/%7Etypes) package

npm i --save lodash

npm i --save-dev **@**types**/**lodash

but in case if you want use types from other repos then can be used old way:

for typescript 1.x:

[Typings](https://github.com/typings/typings) is an npm package that can automatically install type deﬁnition ﬁles into a local project. I recommend that you read the [quickstart](https://github.com/typings/typings#quick-start).

npm **install** -global typings

Now we have access to the typings cli.

1. The ﬁrst step is to search for the package used by the project

typings search lodash

NAME UPDATED

lodash

SOURCE HOMEPAGE

DESCRIPTION VERSIONS

dt

[http:**//**lodash.com**/**](http://lodash.com/)

2

2016-07-20T00:13:09.000Z

lodash global 1

2016-07-01T20:51:07.000Z

lodash npm https:**//**[www.npmjs.com**/**package**/**lodash](http://www.npmjs.com/package/lodash) 1

2016-07-01T20:51:07.000Z

1. Then decide which source you should install from. I use dt which stands for [DeﬁnitelyTyped](https://github.com/DefinitelyTyped/DefinitelyTyped) a GitHub repo where the community can edit typings, it's also normally the most recently updated.
2. Install the typings ﬁles

typings install dt~lodash --global --save

Let's break down the last command. We are installing the DeﬁnitelyTyped version of lodash as a global typings ﬁle in our project and saving it as a dependency in the typings.json. Now wherever we import lodash, typescript will load the lodash typings ﬁle.

1. If we want to install typings that will be used for development environment only, we can supply the --save- dev ﬂag:

typings install chai --save-dev

**Section 14.2: Importing a module from npm**

If you have a type deﬁnition ﬁle (d.ts) for the module, you can use an import statement.

import \_ = require('lodash');

If you don't have a deﬁnition ﬁle for the module, TypeScript will throw an error on compilation because it cannot ﬁnd the module you are trying to import.

In this case, you can import the module with the normal runtime require function. This returns it as the any type, however.

*// The \_ variable is of type any, so TypeScript will not perform any type checking.*

**const** \_: any = require('lodash');

As of TypeScript 2.0, you can also use a *shorthand ambient module declaration* in order to tell TypeScript that a module exists when you don't have a type deﬁnition ﬁle for the module. TypeScript won't be able to provide any meaningful typechecking in this case though.

declare module "lodash";

*// you can now import from lodash in any way you wish:*

import { flatten } from "lodash"; import \* as \_ from "lodash";

As of TypeScript 2.1, the rules have been relaxed even further. Now, as long as a module exists in your node\_modules directory, TypeScript will allow you to import it, even with no module declaration anywhere. (Note that if using the --noImplicitAny compiler option, the below will still generate a warning.)

*// Will work if `node\_modules/someModule/index.js` exists, or if*

*`node\_modules/someModule/package.json` has a valid "main" entry point*

import { foo } from "someModule";

**Section 14.3: Using global external libraries without typings**

Although modules are ideal, if the library you are using is referenced by a global variable (like $ or \_), because it was loaded by a script tag, you can create an ambient declaration in order to refer to it:

declare **const** \_: any;

**Section 14.4: Finding deﬁnition ﬁles with TypeScript 2.x**

With the 2.x versions of TypeScript, typings are now available from the [npm @types repository](https://www.npmjs.com/%7Etypes). These are automatically resolved by the TypeScript compiler and are much simpler to use.

To install a type deﬁnition you simply install it as a dev dependency in your projects package.json e.g.

npm i -S lodash

npm i -D **@**types**/**lodash

after install you simply use the module as before

import \* as \_ from 'lodash'

**Chapter 15: Modules - exporting and importing**

**Section 15.1: Hello world module**

*//hello.ts*

export **function** hello(name: string){ console.log(`Hello ${name}!`);

}

**function** helloES(name: string){ console.log(`Hola ${name}!`);

}

export {helloES}; export **default** hello;

##### Load using directory index

If directory contains ﬁle named index.ts it can be loaded using only directory name (for index.ts ﬁlename is optional).

*//welcome/index.ts*

export **function** welcome(name: string){ console.log(`Welcome ${name}!`);

}

##### Example usage of deﬁned modules

import {hello, helloES} from "./hello"; *// load specified elements*

import defaultHello from "./hello"; *// load default export into name defaultHello*

import \* as Bundle from "./hello"; *// load all exports as Bundle*

import {welcome} from "./welcome"; *// note index.ts is omitted*

hello("World"); helloES("Mundo");

defaultHello("World");

*// Hello World!*

*// Hola Mundo!*

*// Hello World!*

Bundle.hello("World");

Bundle.helloES("Mundo");

*// Hello World!*

*// Hola Mundo!*

welcome("Human");

*// Welcome Human!*

**Section 15.2: Re-export**

TypeScript allow to re-export declarations.

*//Operator.ts*

interface Operator {

eval(a: number, b: number): number;

}

export **default** Operator;

*//Add.ts*

import Operator from "./Operator"; export class Add implements Operator {

eval(a: number, b: number): number {

**return** a + b;

}

}

*//Mul.ts*

import Operator from "./Operator"; export class Mul implements Operator {

eval(a: number, b: number): number {

**return** a \* b;

}

}

You can bundle all operations in single library

*//Operators.ts*

import {Add} from "./Add"; import {Mul} from "./Mul";

export {Add, Mul};

**Named declarations** can be re-exported using shorter syntax

*//NamedOperators.ts*

export {Add} from "./Add"; export {Mul} from "./Mul";

**Default exports** can also be exported, but no short syntax is available. Remember, only one default export per module is possible.

*//Calculator.ts*

export {Add} from "./Add"; export {Mul} from "./Mul";

import Operator from "./Operator";

export **default** Operator;

Possible is re-export of **bundled import**

*//RepackedCalculator.ts*

export \* from "./Operators";

When re-exporting bundle, declarations may be overridden when declared explicitly.

*//FixedCalculator.ts*

export \* from "./Calculator"

import Operator from "./Calculator"; export class Add implements Operator {

eval(a: number, b: number): number {

**return** 42;

}

}

Usage example

*//run.ts*

import {Add, Mul} from "./FixedCalculator";

**const** add = **new** Add();

**const** mul = **new** Mul();

console.log(add.eval(1, 1)); *// 42*

console.log(mul.eval(3, 4)); *// 12*

**Section 15.3: Exporting/Importing declarations**

Any declaration (variable, const, function, class, etc.) can be exported from module to be imported in other module. TypeScript oﬀer two export types: named and default.

##### Named export

*// adams.ts*

export **function** hello(name: string){ console.log(`Hello ${name}!`);

}

export **const** answerToLifeTheUniverseAndEverything = 42; export **const** unused = 0;

When importing named exports, you can specify which elements you want to import.

import {hello, answerToLifeTheUniverseAndEverything} from "./adams"; hello(answerToLifeTheUniverseAndEverything); *// Hello 42!*

##### Default export

Each module can have one default export

*// dent.ts*

**const** defaultValue = 54; export **default** defaultValue;

which can be imported using

import dentValue from "./dent"; console.log(dentValue); *// 54*

##### Bundled import

TypeScript oﬀers method to import whole module into variable

*// adams.ts*

export **function** hello(name: string){ console.log(`Hello ${name}!`);

}

export **const** answerToLifeTheUniverseAndEverything = 42;

import \* as Bundle from "./adams"; Bundle.hello(Bundle.answerToLifeTheUniverseAndEverything); *// Hello 42!* console.log(Bundle.unused); *// 0*

**Chapter 16: Publish TypeScript deﬁnition ﬁles**

**Section 16.1: Include deﬁnition ﬁle with library on npm**

Add typings to your package.json

{

...

"typings": "path/file.d.ts"

...

}

Now whenever that library is imported typescript will load the typings ﬁle

**Chapter 17: Using TypeScript with webpack**

**Section 17.1: webpack.conﬁg.js**

install loaders npm **install** --save-dev ts-loader source-map-loader

##### tsconﬁg.json

{

"compilerOptions": { "sourceMap": **true**, "noImplicitAny": **true**, "module": "commonjs", "target": "es5",

"jsx": "react" *// if you want to use react jsx*

}

}

module.exports = {

entry: "./src/index.ts", output: {

filename: "./dist/bundle.js",

},

*// Enable sourcemaps for debugging webpack's output.*

devtool: "source-map",

resolve: {

*// Add '.ts' and '.tsx' as resolvable extensions.*

extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]

},

module: {

loaders: [

*// All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.*

{test: */\.tsx?$/*, loader: "ts-loader"}

],

preLoaders: [

*// All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.*

{test: */\.js$/*, loader: "source-map-loader"}

]

},

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*\* If you want to use react \**

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

*// When importing a module whose path matches one of the following, just*

*// assume a corresponding global variable exists and use that instead.*

*// This is important because it allows us to avoid bundling all of our*

*// dependencies, which allows browsers to cache those libraries between builds.*

*// externals: {*

*// "react": "React",*

*// "react-dom": "ReactDOM"*

*// },*

};

**Chapter 18: Mixins**

##### Parameter Description

derivedCtor The class that you want to use as the composition class baseCtors An array of classes to be added to the composition class

**Section 18.1: Example of Mixins**

To create mixins, simply declare lightweight classes that can be used as "behaviours".

class Flies {

fly() {

alert('Is it a bird? Is it a plane?');

}

}

class Climbs { climb() {

alert('My spider-sense is tingling.');

}

}

class Bulletproof { deflect() {

alert('My wings are a shield of steel.');

}

}

You can then apply these behaviours to a composition class:

class BeetleGuy implements Climbs, Bulletproof { climb: () => **void**;

deflect: () => **void**;

}

applyMixins (BeetleGuy, [Climbs, Bulletproof]);

The applyMixins function is needed to do the work of composition.

**function** applyMixins(derivedCtor: any, baseCtors: any[]) { baseCtors.forEach(baseCtor => {

Object.getOwnPropertyNames(baseCtor.**prototype**).forEach(name => {

**if** (name !== 'constructor') {

derivedCtor.**prototype**[name] = baseCtor.**prototype**[name];

}

});

});

}

**Chapter 19: How to use a JavaScript library without a type deﬁnition ﬁle**

While some existing JavaScript libraries have [type deﬁnition ﬁles](https://github.com/DefinitelyTyped/DefinitelyTyped), there are many that don't. TypeScript oﬀers a couple patterns to handle missing declarations.

**Section 19.1: Make a module that exports a default any**

For more complicated projects, or in cases where you intend to gradually type a dependency, it may be cleaner to create a module.

Using JQuery (although it [does have typings available](https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/jquery)) as an example:

*// place in jquery.d.ts* declare **let** $: any; export **default** $;

And then in any ﬁle in your project, you can import this deﬁnition with:

*// some other .ts file*

import $ from "jquery";

After this import, $ will be typed as any.

If the library has multiple top-level variables, export and import by name instead:

*// place in jquery.d.ts*

declare module "jquery" {

**let** $: any;

**let** jQuery: any;

export { $ }; export { jQuery };

}

You can then import and use both names:

*// some other .ts file*

import {$, jQuery} from "jquery";

$.doThing(); jQuery.doOtherThing();

**Section 19.2: Declare an any global**

It is sometimes easiest to just declare a global of type any, especially in simple projects. If jQuery didn't have type declarations ([it does](https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/jquery)), you could put

declare **var** $: any;

Now any use of $ will be typed any.

**Section 19.3: Use an ambient module**

If you just want to indicate the *intent* of an import (so you don't want to declare a global) but don't wish to bother with any explicit deﬁnitions, you can import an ambient module.

*// in a declarations file (like declarations.d.ts)*

declare module "jquery"; *// note that there are no defined exports*

You can then import from the ambient module.

*// some other .ts file*

import {$, jQuery} from "jquery";

Anything imported from the declared module (like $ and jQuery) above will be of type any

**Chapter 20: TypeScript installing typescript and running the typescript compiler tsc**

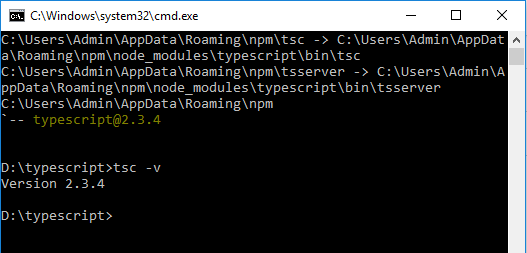
How to install TypeScript and run the TypeScript compiler against a .ts ﬁle from the command line.

**Section 20.1: Steps**

##### Installing TypeScript and running typescript compiler. To install TypeScript Compiler

npm **install** -g typescript

##### To check with the typescript version

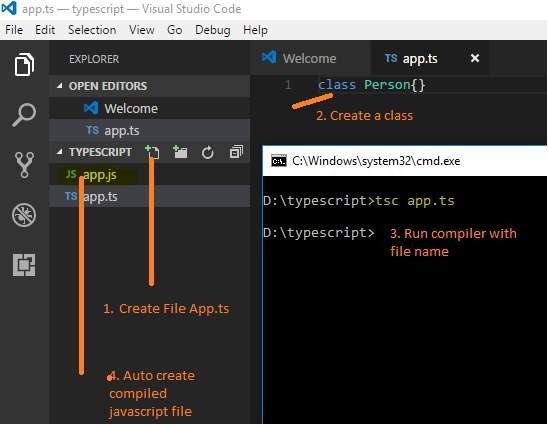


tsc -v

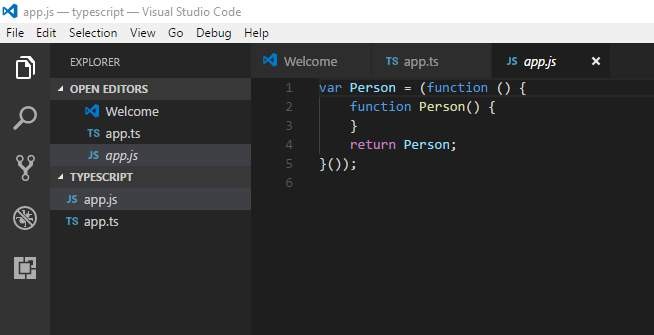
##### Download Visual Studio Code for Linux/Windows

[Visual Code Download Link](https://code.visualstudio.com/)

1. Open Visual Studio Code
2. Open Same Folder where you have installed TypeScript compiler
3. Add File by clicking on plus icon on left pane
4. Create a basic class.
5. Compile your type script ﬁle and generate output.



See the result in compiled javascript of written typescript code.



Thank you.

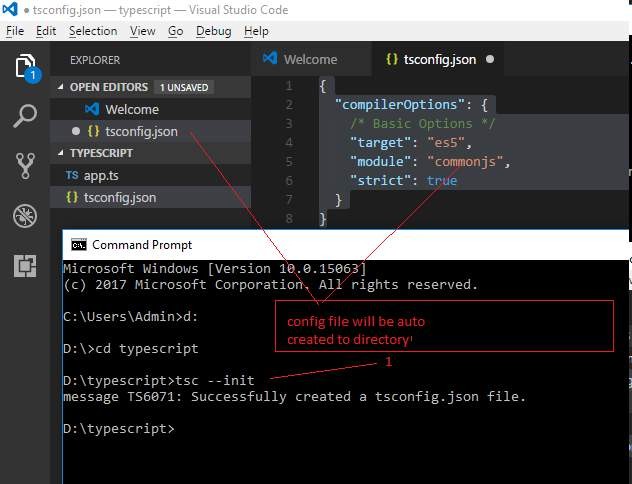
**Chapter 21: Conﬁgure typescript project to compile all ﬁles in typescript.**

creating your ﬁrst .tsconﬁg conﬁguration ﬁle which will tell the TypeScript compiler how to treat your .ts ﬁles

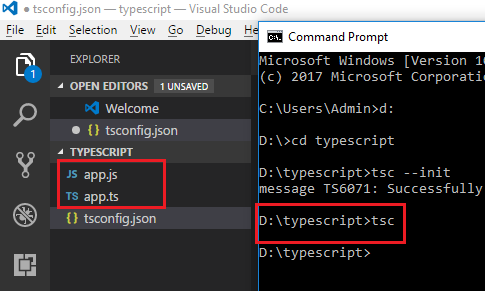
**Section 21.1: TypeScript Conﬁguration ﬁle setup**

Enter command "**tsc --init**" and hit enter.

Before that we need to compile ts ﬁle with command "**tsc app.ts**" now it is all deﬁned in below conﬁg ﬁle automatically.



Now, You can compile all typescripts by command "**tsc**". it will automatically create ".js" ﬁle of your typescript ﬁle.



If you will create another typescript and hit "tsc" command in command prompt or terminal javascript ﬁle will be automatically created for typescript ﬁle.

Thank you,

**Chapter 22: Integrating with Build Tools**

**Section 22.1: Browserify**

##### Install

npm **install** tsify

##### Using Command Line Interface

browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js

##### Using API

**var** browserify = require("browserify");

**var** tsify = require("tsify");

browserify()

.add("main.ts")

.plugin("tsify", { noImplicitAny: **true** })

.bundle()

.pipe(process.stdout);

More details: [smrq/tsify](https://github.com/smrq/tsify)

**Section 22.2: Webpack**

##### Install

npm **install** ts-loader --save-dev

##### Basic webpack.conﬁg.js

##### webpack 2.x, 3.x

module.exports = { resolve: {

extensions: ['.ts', '.tsx', '.js']

},

module: {

rules: [

{

*// Set up ts-loader for .ts/.tsx files and exclude any imports from node\_modules.*

test: */\.tsx?$/*, loaders: ['ts-loader'], exclude: */node\_modules/*

}

]

},

entry: [

*// Set index.tsx as application entry point.*

'./index.tsx'

],

output: {

filename: "bundle.js"

}

};

##### webpack 1.x

module.exports = {

entry: "./src/index.tsx", output: {

filename: "bundle.js"

},

resolve: {

*// Add '.ts' and '.tsx' as a resolvable extension.*

extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]

},

module: {

loaders: [

*// all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'*

{ test: */\.ts(x)?$/*, loader: "ts-loader", exclude: */node\_modules/* }

]

}

}

See more details on [ts-loader here](https://www.npmjs.com/package/ts-loader). Alternatives:

[awesome-typescript-loader](https://www.npmjs.com/package/awesome-typescript-loader)

**Section 22.3: Grunt**

##### Install

npm **install** grunt-ts

##### Basic Gruntﬁle.js

module.exports = **function**(grunt) { grunt.initConfig({

ts: {

**default** : {

src: ["\*\*/\*.ts", "!node\_modules/\*\*/\*.ts"]

}

}

});

grunt.loadNpmTasks("grunt-ts"); grunt.registerTask("default", ["ts"]);

};

More details: [TypeStrong/grunt-ts](https://github.com/TypeStrong/grunt-ts)

**Section 22.4: Gulp**

##### Install

npm **install** gulp-typescript

##### Basic gulpﬁle.js

**var** gulp = require("gulp");

**var** ts = require("gulp-typescript");

gulp.task("default", **function** () {

**var** tsResult = gulp.src("src/\*.ts")

.pipe(ts({

noImplicitAny: **true**, out: "output.js"

}));

**return** tsResult.js.pipe(gulp.dest("built/local"));

});

##### gulpﬁle.js using an existing tsconﬁg.json

**var** gulp = require("gulp");

**var** ts = require("gulp-typescript");

**var** tsProject = ts.createProject('tsconfig.json', {

noImplicitAny: **true** *// You can add and overwrite parameters here*

});

gulp.task("default", **function** () {

**var** tsResult = tsProject.src()

.pipe(tsProject());

**return** tsResult.js.pipe(gulp.dest('release'));

});

More details: [ivogabe/gulp-typescript](https://github.com/ivogabe/gulp-typescript)

**Section 22.5: MSBuild**

Update project ﬁle to include locally installed Microsoft.TypeScript.Default.props (at the top) and

Microsoft.TypeScript.targets (at the bottom) ﬁles:

**<?xml** version="1.0" encoding="utf-8"**?>**

**<Project** ToolsVersion="4.0" DefaultTargets="Build" xmlns=["http://schemas.microsoft.com/developer/msbuild/2003"](http://schemas.microsoft.com/developer/msbuild/2003)**>**

*<!-- Include default props at the bottom -->*

**<Import**

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Micr osoft.TypeScript.Default.props"

Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeS cript\Microsoft.TypeScript.Default.props')" **/>**

*<!-- TypeScript configurations go here -->*

**<PropertyGroup** Condition="'$(Configuration)' == 'Debug'"**>**

**<TypeScriptRemoveComments>**false**</TypeScriptRemoveComments>**

**<TypeScriptSourceMap>**true**</TypeScriptSourceMap>**

**</PropertyGroup>**

**<PropertyGroup** Condition="'$(Configuration)' == 'Release'"**>**

**<TypeScriptRemoveComments>**true**</TypeScriptRemoveComments>**

**<TypeScriptSourceMap>**false**</TypeScriptSourceMap>**

**</PropertyGroup>**

*<!-- Include default targets at the bottom -->*

**<Import**

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Micr osoft.TypeScript.targets"

Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeS cript\Microsoft.TypeScript.targets')" **/>**

**</Project>**

More details about deﬁning MSBuild compiler options: [Setting Compiler Options in MSBuild projects](http://www.typescriptlang.org/docs/handbook/compiler-options-in-msbuild.html)

**Section 22.6: NuGet**

Right-Click -> Manage NuGet Packages Search for Microsoft.TypeScript.MSBuild Hit Install

When install is complete, rebuild!

More details can be found at [Package Manager Dialog](http://docs.nuget.org/Consume/Package-Manager-Dialog) and [using nightly builds with NuGet](https://github.com/Microsoft/TypeScript/wiki/Nightly-drops#using-nuget-with-msbuild)

**Section 22.7: Install and conﬁgure webpack + loaders**

Installation

npm **install** -D webpack typescript ts-loader

webpack.conﬁg.js

module.exports = { entry: {

app: ['./src/'],

},

output: {

path: dirname,

filename: './dist/[name].js',

},

resolve: {

extensions: ['', '.js', '.ts'],

},

module: { loaders: [{

test: */\.ts(x)$/*, loaders: ['ts-loader'], exclude: */node\_modules/*

}],

}

};

**Chapter 23: Using TypeScript with RequireJS**

RequireJS is a JavaScript ﬁle and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Rhino and Node. Using a modular script loader like RequireJS will improve the speed and quality of your code.

Using TypeScript with RequireJS requires conﬁguration of tsconﬁg.json, and including an snippet in any HTML ﬁle. Compiler will traduce imports from the syntax of TypeScript to RequireJS' format.

**Section 23.1: HTML example using RequireJS CDN to include an already compiled TypeScript ﬁle**

**<body** onload=" init();"**>**

...

**<script** src=["http://requirejs.org/docs/release/2.3.2/comments/require.js"](http://requirejs.org/docs/release/2.3.2/comments/require.js)**></script>**

**<script>**

function init() { require(["view/index.js"]);

}

**</script>**

**</body>**

**Section 23.2: tsconﬁg.json example to compile to view folder using RequireJS import style**

{

"module": "amd",

*// Using AMD module code generator which works with RequireJS*

"rootDir": "./src", *// Change this to your source folder*

"outDir": "./view",

...

}

**Chapter 24: TypeScript with AngularJS**

##### Name Description

controllerAs is an alias name, to which variables or functions can be assigned to. @see: <https://docs.angularjs.org/guide/directive>

$inject Dependency Injection list, it is resolved by angular and passing as an argument to constructor functions.

**Section 24.1: Directive**

interface IMyDirectiveController {

*// specify exposed controller methods and properties here*

getUrl(): string;

}

class MyDirectiveController implements IMyDirectiveController {

*// Inner injections, per each directive*

public **static** $inject = ["$location", "toaster"];

constructor(private $location: ng.ILocationService, private toaster: any) {

*// $location and toaster are now properties of the controller*

}

public getUrl(): string {

**return this**.$location.url(); *// utilize $location to retrieve the URL*

}

}

*/\**

* *Outer injections, for run once control.*
* *For example we have all templates in one value, and we want to use it.*

*\*/*

export **function** myDirective(templatesUrl: ITemplates): ng.IDirective {

**return** {

controller: MyDirectiveController, controllerAs: "vm",

link: (scope: ng.IScope,

element: ng.IAugmentedJQuery, attributes: ng.IAttributes,

controller: IMyDirectiveController): **void** => {

**let** url = controller.getUrl(); element.text("Current URL: " + url);

},

replace: **true**, require: "ngModel", restrict: "A",

templateUrl: templatesUrl.myDirective,

};

}

myDirective.$inject = [ Templates.**prototype**.slug,

];

*// Using slug naming across the projects simplifies change of the directive name*

myDirective.**prototype**.slug = "myDirective";

*// You can place this in some bootstrap file, or have them at the same file*

angular.module("myApp"). directive(myDirective.**prototype**.slug, myDirective);

**Section 24.2: Simple example**

export **function** myDirective($location: ng.ILocationService): ng.IDirective {

**return** {

link: (scope: ng.IScope, element: ng.IAugmentedJQuery,

attributes: ng.IAttributes): **void** => { element.text("Current URL: " + $location.url());

},

replace: **true**, require: "ngModel", restrict: "A",

templateUrl: templatesUrl.myDirective,

};

}

*// Using slug naming across the projects simplifies change of the directive name*

myDirective.**prototype**.slug = "myDirective";

*// You can place this in some bootstrap file, or have them at the same file*

angular.module("myApp"). directive(myDirective.**prototype**.slug, [

Templates.**prototype**.slug,

myDirective

]);

**Section 24.3: Component**

For an easier transition to Angular 2, it's recommended to use Component, available since Angular 1.5.8

##### myModule.ts

import { MyModuleComponent } from "./components/myModuleComponent"; import { MyModuleService } from "./services/MyModuleService";

angular

.module("myModule", [])

.component("myModuleComponent", **new** MyModuleComponent())

.service("myModuleService", MyModuleService);

##### components/myModuleComponent.ts

import IComponentOptions = angular.IComponentOptions;

import IControllerConstructor = angular.IControllerConstructor; import Injectable = angular.Injectable;

import { MyModuleController } from "../controller/MyModuleController";

export class MyModuleComponent implements IComponentOptions {

public templateUrl: string = "./app/myModule/templates/myComponentTemplate.html"; public controller: Injectable<IControllerConstructor> = MyModuleController;

public bindings: {[boundProperty: string]: string} = {};

}

##### templates/myModuleComponent.html

**<div** class="my-module-component"**>**

{{$ctrl.someContent}}

**</div>**

##### controller/MyModuleController.ts

import IController = angular.IController;

import { MyModuleService } from "../services/MyModuleService";

export class MyModuleController implements IController {

public **static** readonly $inject: string[] = ["$element", "myModuleService"]; public someContent: string = "Hello World";

constructor($element: JQuery, private myModuleService: MyModuleService) { console.log("element", $element);

}

public doSomething(): **void** {

*// implementation..*

}

}

##### services/MyModuleService.ts

export class MyModuleService {

public **static** readonly $inject: string[] = [];

constructor() {

}

public doSomething(): **void** {

*// do something*

}

}

##### somewhere.html

**<my-module-component></my-module-component>**

**Chapter 25: TypeScript with SystemJS**

**Section 25.1: Hello World in the browser with SystemJS**

##### Install systemjs and plugin-typescript

npm **install** systemjs

npm **install** plugin-typescript

NOTE: this will install typescript 2.0.0 compiler which is not released yet. For TypeScript 1.8 you have to use plugin-typescript 4.0.16

**Create hello.ts ﬁle**

export **function** greeter(person: String) {

**return** 'Hello, ' + person;

}

**Create hello.html ﬁle**

<!doctype html**>**

**<html>**

**<head>**

**<title>**Hello World in TypeScript**</title>**

**<script** src="node\_modules/systemjs/dist/system.src.js"**></script>**

**<script** src="config.js"**></script>**

**<script>**

window.addEventListener('load', function() { System.import('./hello.ts').then(function(hello) {

document.body.innerHTML = hello.greeter('World');

});

});

**</script>**

**</head>**

**<body>**

**</body>**

**</html>**

**Create config.js - SystemJS conﬁguration ﬁle**

System.config**({** packages: **{**

"plugin-typescript": **{**

"main": "plugin.js"

**}**,

"typescript": **{**

"main": "lib/typescript.js", "meta": **{**

"lib/typescript.js": **{**

"exports": "ts"

**}**

**}**

**}**

**}**,

map: **{**

"plugin-typescript": "node\_modules/plugin-typescript/lib/",

**/\*** NOTE: this is **for** npm 3 **(**node 6**) \*/**

**/\* for** npm 2, typescript path will be **\*/**

**/\*** node\_modules**/**plugin-typescript**/**node\_modules**/**typescript **\*/**

"typescript": "node\_modules/typescript/"

**}**,

transpiler: "plugin-typescript", meta: **{**

"./hello.ts": **{**

format: "esm",

loader: "plugin-typescript"

**}**

**}**,

typescriptOptions: **{**

typeCheck: 'strict'

**}**

**})**;

NOTE: if you don't want type checking, remove loader: "plugin-typescript" and typescriptOptions from

config.js. Also note that it will never check javascript code, in particular code in the **<script>** tag in html example.

##### Test it

npm **install** live-server

.**/**node\_modules**/**.bin**/**live-server --open=hello.html

##### Build it for production

npm **install** systemjs-builder

Create build.js ﬁle:

**var** Builder = require('systemjs-builder');

**var** builder = **new** Builder(); builder.loadConfig('./config.js').then(**function**() {

builder.bundle('./hello.ts', './hello.js', {minify: **true**});

});

build hello.js from hello.ts

node build.js

##### Use it in production

Just load hello.js with a script tag before ﬁrst use

hello-production.html ﬁle:

<!doctype html**>**

**<html>**

**<head>**

**<title>**Hello World in TypeScript**</title>**

**<script** src="node\_modules/systemjs/dist/system.src.js"**></script>**

**<script** src="config.js"**></script>**

**<script** src="hello.js"**></script>**

**<script>**

window.addEventListener('load', function() { System.import('./hello.ts').then(function(hello) {

document.body.innerHTML = hello.greeter('World');

});

});

**</script>**

**</head>**

**<body>**

**</body>**

**</html>**

**Chapter 26: Using TypeScript with React (JS & native)**

**Section 26.1: ReactJS component written in TypeScript**

You can use ReactJS's components easily in TypeScript. Just rename the 'jsx' ﬁle extension to 'tsx':

*//helloMessage.tsx:*

**var** HelloMessage = React.createClass({ render: **function**() {

**return** <div>Hello {**this**.props.name}</div>;

}

});

ReactDOM.render(<HelloMessage name="John" />, mountNode);

But in order to make full use of TypeScript's main feature (static type checking) you must do a couple things:

##### convert React.createClass to an ES6 Class:

*//helloMessage.tsx:*

class HelloMessage extends React.Component { render() {

**return** <div>Hello {**this**.props.name}</div>;

}

}

ReactDOM.render(<HelloMessage name="John" />, mountNode);

For more info on converting to ES6 look [here](http://www.newmediacampaigns.com/blog/refactoring-react-components-to-es6-classes)

##### Add Props and State interfaces:

interface Props { name:string; optionalParam?:number;

}

interface State {

*//empty in our case*

}

class HelloMessage extends React.Component<Props, State> { render() {

**return** <div>Hello {**this**.props.name}</div>;

}

}

*// TypeScript will allow you to create without the optional parameter*

ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);

*// But it does check if you pass in an optional parameter of the wrong type*

ReactDOM.render(<HelloMessage name="Sebastian" optionalParam='foo' />, mountNode);

Now TypeScript will display an error if the programmer forgets to pass props. Or if trying to pass in props that are not deﬁned in the interface.

**Section 26.2: TypeScript & react & webpack**

Installing typescript, typings and webpack globally

npm **install** -g typescript typings webpack

Installing loaders and linking typescript

npm **install** --save-dev ts-loader source-map-loader npm **link** typescript

Linking TypeScript allows ts-loader to use your global installation of TypeScript instead of needing a separate local copy [typescript doc](https://www.typescriptlang.org/docs/handbook/react-%26-webpack.html)

installing .d.ts ﬁles with typescript 2.x

npm i **@**types**/**react --save-dev

npm i **@**types**/**react-dom --save-dev

installing .d.ts ﬁles with typescript 1.x

typings install --global --save dt~react typings install --global --save dt~react-dom

tsconfig.json conﬁguration ﬁle

{

"compilerOptions": { "sourceMap": **true**, "noImplicitAny": **true**, "module": "commonjs", "target": "es5",

"jsx": "react"

}

}

webpack.config.js conﬁguration ﬁle

module.exports = {

entry: "<path to entry point>",*// for example ./src/helloMessage.tsx*

output: {

filename: "<path to bundle file>", *// for example ./dist/bundle.js*

},

*// Enable sourcemaps for debugging webpack's output.*

devtool: "source-map",

resolve: {

*// Add '.ts' and '.tsx' as resolvable extensions.*

extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]

},

module: {

loaders: [

*// All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.*

{test: */\.tsx?$/*, loader: "ts-loader"}

],

preLoaders: [

*// All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.*

{test: */\.js$/*, loader: "source-map-loader"}

]

},

*// When importing a module whose path matches one of the following, just*

*// assume a corresponding global variable exists and use that instead.*

*// This is important because it allows us to avoid bundling all of our*

*// dependencies, which allows browsers to cache those libraries between builds.*

externals: {

"react": "React", "react-dom": "ReactDOM"

},

};

ﬁnally run webpack or webpack -w (for watch mode)

**Note**: React and ReactDOM are marked as external

**Chapter 27: TSLint - assuring code quality and consistency**

TSLint performs static analysis of code and detect errors and potential problems in code.

**Section 27.1: Conﬁguration for fewer programming errors**

This tslint.json example contains a set of conﬁguration to enforce more typings, catch common errors or otherwise confusing constructs that are prone to producing bugs and following more the [Coding Guidelines for TypeScript](https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines) [Contributors](https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines).

To enforce this rules, include tslint in your build process and check your code before compiling it with tsc.

{

"rules": {

*// TypeScript Specific*

"member-access": **true**, *// Requires explicit visibility declarations for class members.*

"no-any": **true**, *// Disallows usages of any as a type declaration.*

*// Functionality*

"label-position": **true**, *// Only allows labels in sensible locations.*

"no-bitwise": **true**, *// Disallows bitwise operators.*

"no-eval": **true**, *// Disallows eval function invocations.*

"no-null-keyword": **true**, *// Disallows use of the null keyword literal.*

"no-unsafe-finally": **true**, *// Disallows control flow statements, such as return, continue,*

*break and throws in finally blocks.*

"no-var-keyword": **true**, *// Disallows usage of the var keyword.*

"radix": **true**, *// Requires the radix parameter to be specified when calling parseInt.*

"triple-equals": **true**, *// Requires === and !== in place of == and !=.*

"use-isnan": **true**, *// Enforces use of the isNaN() function to check for NaN references instead of a comparison to the NaN constant.*

*// Style*

"class-name": **true**, *// Enforces PascalCased class and interface names.*

"interface-name": [ **true**, "never-prefix" ], *// Requires interface names to begin with a capital*

‘*I*’

"no-angle-bracket-type-assertion": **true**, *// Requires the use of as Type for type assertions*

*instead of <Type>.*

"one-variable-per-declaration": **true**, *// Disallows multiple variable definitions in the same declaration statement.*

"quotemark": [ **true**, "double", "avoid-escape" ], *// Requires double quotes for string literals.*

"semicolon": [ **true**, "always" ], *// Enforces consistent semicolon usage at the end of every statement.*

"variable-name": [**true**, "ban-keywords", "check-format", "allow-leading-underscore"] *// Checks variable names for various errors. Disallows the use of certain TypeScript keywords (any, Number, number, String, string, Boolean, boolean, undefined) as variable or parameter. Allows only camelCased or UPPER\_CASED variable names. Allows underscores at the beginning (only has an effect if* “*check- format*” *specified).*

}

}

**Section 27.2: Installation and setup**

To install [tslint](https://github.com/palantir/tslint) run command

npm **install** -g tslint

Tslint is conﬁgured via ﬁle tslint.json. To initialize default conﬁguration run command

tslint --init

To check ﬁle for possible errors in ﬁle run command

tslint filename.ts

**Section 27.3: Sets of TSLint Rules**

[tslint-microsoft-contrib](https://github.com/Microsoft/tslint-microsoft-contrib) [tslint-eslint-rules](https://github.com/buzinas/tslint-eslint-rules) [codelyzer](https://github.com/mgechev/codelyzer)

Yeoman generator supports all these presets and can be extends also: [generator-tslint](https://github.com/greybax/generator-tslint)

**Section 27.4: Basic tslint.json setup**

This is a basic tslint.json setup which prevents use of any

requires curly braces for if/**else**/**for**/do/while statements

requires double quotes (") to be used for strings

{

"rules": {

"no-any": **true**,

"curly": **true**,

"quotemark": [**true**, "double"]

}

}

**Section 27.5: Using a predeﬁned ruleset as default**

tslint can extend an existing rule set and is shipped with the defaults tslint:recommended and tslint:latest.

tslint:recommended is a stable, somewhat opinionated set of rules which we encourage for general TypeScript programming. This conﬁguration follows semver, so it will not have breaking changes across minor or patch releases.

tslint:latest extends tslint:recommended and is continuously updated to include conﬁguration for the latest rules in every TSLint release. Using this conﬁg may introduce breaking changes across minor releases as new rules are enabled which cause lint failures in your code. When TSLint reaches a major version bump, tslint:recommended will be updated to be identical to tslint:latest.

[Docs](https://www.npmjs.com/package/tslint) and [source code of predeﬁned ruleset](https://github.com/palantir/tslint/tree/master/src/configs) So one can simply use:

{

"extends": "tslint:recommended"

}

to have a sensible starting conﬁguration.

One can then overwrite rules from that preset via rules, e.g. for node developers it made sense to set no-console

to **false**:

{

"extends": "tslint:recommended", "rules": {

"no-console": **false**

}

}

**Chapter 28: tsconﬁg.json**

**Section 28.1: Create TypeScript project with tsconﬁg.json**

The presence of a **tsconﬁg.json** ﬁle indicates that the current directory is the root of a TypeScript enabled project. Initializing a TypeScript project, or better put tsconﬁg.json ﬁle, can be done through the following command:

tsc --init

As of TypeScript v2.3.0 and higher this will create the following tsconﬁg.json by default:

{

"compilerOptions": {

*/\* Basic Options \*/*

"target": "es5", */\* Specify ECMAScript target version: 'ES3' (default),*

*'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. \*/*

"module": "commonjs", */\* Specify module code generation: 'commonjs', 'amd', 'system', 'umd' or 'es2015'. \*/*

*// "lib": [], /\* Specify library files to be included in the compilation: \*/*

*// "allowJs": true, /\* Allow javascript files to be compiled. \*/*

*// "checkJs": true, /\* Report errors in .js files. \*/*

*// "jsx": "preserve", /\* Specify JSX code generation: 'preserve', 'react- native', or 'react'. \*/*

*// "declaration": true, /\* Generates corresponding '.d.ts' file. \*/*

*// "sourceMap": true, /\* Generates corresponding '.map' file. \*/*

*// "outFile": "./", /\* Concatenate and emit output to single file. \*/*

*// "outDir": "./", /\* Redirect output structure to the directory. \*/*

*// "rootDir": "./", /\* Specify the root directory of input files. Use to control the output directory structure with --outDir. \*/*

*// "removeComments": true, /\* Do not emit comments to output. \*/*

*// "noEmit": true, /\* Do not emit outputs. \*/*

*// "importHelpers": true, /\* Import emit helpers from 'tslib'. \*/*

*// "downlevelIteration": true, /\* Provide full support for iterables in 'for-of', spread, and destructuring when targeting 'ES5' or 'ES3'. \*/*

*// "isolatedModules": true, /\* Transpile each file as a separate module (similar to 'ts.transpileModule'). \*/*

*/\* Strict Type-Checking Options \*/*

"strict": **true** */\* Enable all strict type-checking options. \*/*

*// "noImplicitAny": true, /\* Raise error on expressions and declarations with an implied 'any' type. \*/*

*// "strictNullChecks": true, /\* Enable strict null checks. \*/*

*// "noImplicitThis": true, /\* Raise error on 'this' expressions with an implied 'any' type. \*/*

*// "alwaysStrict": true, /\* Parse in strict mode and emit "use strict" for each source file. \*/*

*/\* Additional Checks \*/*

*// "noUnusedLocals": true, /\* Report errors on unused locals. \*/*

*// "noUnusedParameters": true, /\* Report errors on unused parameters. \*/*

*// "noImplicitReturns": true, /\* Report error when not all code paths in function return a value. \*/*

*// "noFallthroughCasesInSwitch": true, /\* Report errors for fallthrough cases in switch statement. \*/*

*/\* Module Resolution Options \*/*

*// "moduleResolution": "node", /\* Specify module resolution strategy: 'node' (Node.js)*

*or 'classic' (TypeScript pre-1.6). \*/*

*// "baseUrl": "./", /\* Base directory to resolve non-absolute module names.*

*\*/*

*// "paths": {}, /\* A series of entries which re-map imports to lookup*

*locations relative to the 'baseUrl'. \*/*

*// "rootDirs": [], /\* List of root folders whose combined content represents the structure of the project at runtime. \*/*

*// "typeRoots": [], /\* List of folders to include type definitions from. \*/*

*// "types": [], /\* Type declaration files to be included in compilation. \*/*

*// "allowSyntheticDefaultImports": true, /\* Allow default imports from modules with no default export. This does not affect code emit, just typechecking. \*/*

*/\* Source Map Options \*/*

*// "sourceRoot": "./", /\* Specify the location where debugger should locate TypeScript files instead of source locations. \*/*

*// "mapRoot": "./", /\* Specify the location where debugger should locate map files instead of generated locations. \*/*

*// "inlineSourceMap": true, /\* Emit a single file with source maps instead of having a separate file. \*/*

*// "inlineSources": true, /\* Emit the source alongside the sourcemaps within a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. \*/*

*/\* Experimental Options \*/*

*// "experimentalDecorators": true, /\* Enables experimental support for ES7 decorators. \*/*

*// "emitDecoratorMetadata": true, /\* Enables experimental support for emitting type metadata for decorators. \*/*

}

}

Most, if not all, options are generated automatically with only the bare necessities left uncommented. Older versions of TypeScript, like for example v2.0.x and lower, would generate a tsconﬁg.json like this:

{

"compilerOptions": { "module": "commonjs", "target": "es5", "noImplicitAny": **false**, "sourceMap": **false**

}

}

**Section 28.2: Conﬁguration for fewer programming errors**

There are very good conﬁgurations to force typings and get more helpful errors which are not activated by default.

{

"compilerOptions": {

"alwaysStrict": **true**, *// Parse in strict mode and emit "use strict" for each source file.*

*// If you have wrong casing in referenced files e.g. the filename is Global.ts and you have a ///*

*<reference path="global.ts" /> to reference this file, then this can cause to unexpected errors.*

*Visite:* [*http://stackoverflow.com/questions/36628612/typescript-transpiler-casing-issue*](http://stackoverflow.com/questions/36628612/typescript-transpiler-casing-issue)

"forceConsistentCasingInFileNames": **true**, *// Disallow inconsistently-cased references to the same file.*

*// "allowUnreachableCode": false, // Do not report errors on unreachable code. (Default: False)*

*// "allowUnusedLabels": false, // Do not report errors on unused labels. (Default: False)*

"noFallthroughCasesInSwitch": **true**, *// Report errors for fall through cases in switch statement.*

"noImplicitReturns": **true**, *// Report error when not all code paths in function return a value.*

"noUnusedParameters": **true**, *// Report errors on unused parameters.*

"noUnusedLocals": **true**, *// Report errors on unused locals.*

"noImplicitAny": **true**, *// Raise error on expressions and declarations with an implied "any" type.*

"noImplicitThis": **true**, *// Raise error on this expressions with an implied "any" type.*

"strictNullChecks": **true**, *// The null and undefined values are not in the domain of every type and are only assignable to themselves and any.*

*// To enforce this rules, add this configuration.*

"noEmitOnError": **true** *// Do not emit outputs if any errors were reported.*

}

}

Not enough? If you are a hard coder and want more, then you may be interested to check your TypeScript ﬁles with tslint before compiling it with tsc. Check how to conﬁgure tslint for even stricter code.

**Section 28.3: compileOnSave**

Setting a top-level property compileOnSave signals to the IDE to generate all ﬁles for a given **tsconﬁg.json** upon saving.

{

"compileOnSave": **true**, "compilerOptions": {

...

},

"exclude": [

...

]

}

This feature is available since TypeScript 1.8.4 and onward, but needs to be directly supported by IDE's. Currently, examples of supported IDE's are:

Visual Studio 2015 [with Update 3](https://github.com/Microsoft/TypeScript/issues/6782) [JetBrains WebStorm](https://blog.jetbrains.com/webstorm/2016/03/how-to-compile-typescript-in-webstorm/)

Atom [with atom-typescript](https://github.com/TypeStrong/atom-typescript/blob/master/docs/tsconfig.md#compileonsave)

**Section 28.4: Comments**

A tsconﬁg.json ﬁle can contain both line and block comments, using the same rules as ECMAScript.

*//Leading comment*

{

"compilerOptions": {

*//this is a line comment*

"module": "commonjs", *//eol line comment*

"target" */\*inline block\*/* : "es5",

*/\* This is a block comment \*/*

}

}

*/\* trailing comment \*/*

**Section 28.5: preserveConstEnums**

TypeScript supports constant enumerables, declared through **const** enum.

This is usually just syntax sugar as the constant enums are inlined in compiled JavaScript. For instance the following code

**const** enum Tristate { True,

False, Unknown

}

**var** something = Tristate.True;

compiles to

**var** something = 0;

Although the performance beneﬁt from inlining, you may prefer to keep enums even if constant (ie: you may wish readability on development code), to do this you have to set in **tsconﬁg.json** the preserveConstEnums clause into the compilerOptions to **true**.

{

"compilerOptions": { "preserveConstEnums" = **true**,

...

},

"exclude": [

...

]

}

By this way the previous example would be compiled as any other enums, as shown in following snippet.

**var** Tristate; (**function** (Tristate) {

Tristate[Tristate["True"] = 0] = "True"; Tristate[Tristate["False"] = 1] = "False"; Tristate[Tristate["Unknown"] = 2] = "Unknown";

})(Tristate || (Tristate = {}));

**var** something = Tristate.True

**Chapter 29: Debugging**

There are two ways of running and debugging TypeScript:

**Transpile to JavaScript**, run in node and use mappings to link back to the TypeScript source ﬁles or

**Run TypeScript directly** using [ts-node](https://www.npmjs.com/package/ts-node)

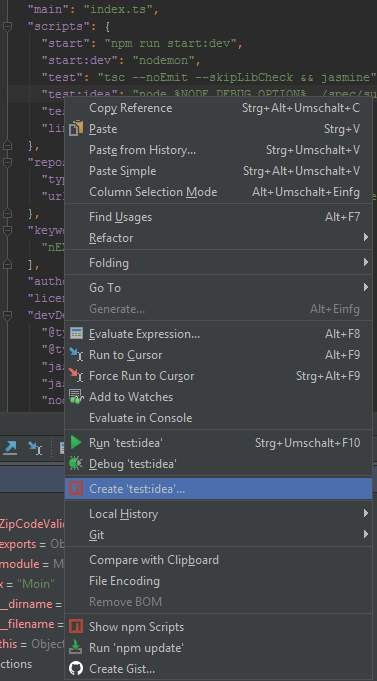
This article describes both ways using [Visual Studio Code](https://code.visualstudio.com/) and [WebStorm](https://www.jetbrains.com/webstorm/). All examples presume that your main ﬁle is *index.ts*.

**Section 29.1: TypeScript with ts-node in WebStorm**

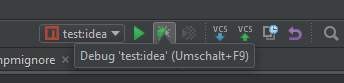
Add this script to your package.json:

"start:idea": "ts-node %NODE\_DEBUG\_OPTION% --ignore false index.ts",

Right click on the script and select *Create 'test:idea'...* and conﬁrm with 'OK' to create the debug conﬁguration:



Start the debugger using this conﬁguration:



**Section 29.2: TypeScript with ts-node in Visual Studio Code**

Add ts-node to your TypeScript project:

npm i ts-node

Add a script to your package.json:

"start:debug": "ts-node --inspect=5858 --debug-brk --ignore false index.ts"

The launch.json needs to be conﬁgured to use the *node2* type and start npm running the start:debug script:

{

"version": "0.2.0", "configurations": [

{

"type": "node2",

"request": "launch", "name": "Launch Program", "runtimeExecutable": "npm", "windows": {

"runtimeExecutable": "npm.cmd"

},

"runtimeArgs": [ "run-script", "start:debug"

],

"cwd": "${workspaceRoot}/server", "outFiles": [],

"port": 5858, "sourceMaps": **true**

}

]

}

**Section 29.3: JavaScript with SourceMaps in Visual Studio Code**

In the tsconfig.json set

"sourceMap": **true**,

to generate mappings alongside with js-ﬁles from the TypeScript sources using the tsc command. The [launch.json](https://code.visualstudio.com/Docs/editor/debugging) ﬁle:

{

"version": "0.2.0", "configurations": [

{

"type": "node",

"request": "launch", "name": "Launch Program",

"program": "${workspaceRoot}**\\**index.js", "cwd": "${workspaceRoot}",

"outFiles": [], "sourceMaps": **true**

}

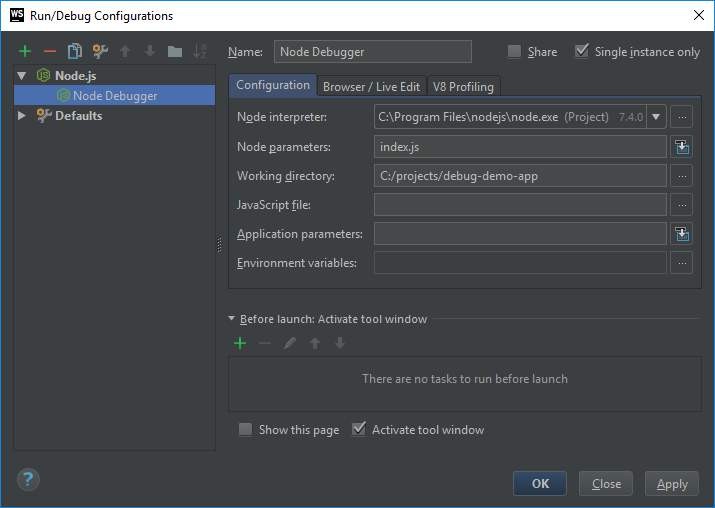
]

}

This starts node with the generated index.js (if your main ﬁle is index.ts) ﬁle and the debugger in Visual Studio Code which halts on breakpoints and resolves variable values within your TypeScript code.

**Section 29.4: JavaScript with SourceMaps in WebStorm**

Create a *Node.js* [debug conﬁguration](https://www.jetbrains.com/help/webstorm/2016.3/creating-and-editing-run-debug-configurations.html#createExplicitly) and use index.js as *Node parameters*.



**Chapter 30: Unit Testing**

**Section 30.1: tape**

[tape](https://github.com/substack/tape) is minimalistic JavaScript testing framework, it outputs [TAP-compliant](https://testanything.org/) markup. To install tape using npm run command

npm **install** --save-dev tape **@**types**/**tape

To use tape with TypeScript you need to install ts-node as global package, to do this run command

npm **install** -g ts-node

Now you are ready to write your ﬁrst test

*//math.test.ts*

import \* as test from "tape";

test("Math test", (t) => {

t.equal(4, 2 + 2);

t.**true**(5 > 2 + 2);

t.end();

});

To execute test run command

ts-node node\_modules/tape/bin/tape math.test.ts

In output you should see

TAP version 13 # Math test

ok 1 should be equal ok 2 should be truthy

1..2

# tests 2

# pass 2 # ok

Good job, you just ran your TypeScript test.

##### Run multiple test ﬁles

You can run multiple test ﬁles at once using path wildcards. To execute all TypeScript tests in tests directory run command

ts-node node\_modules/tape/bin/tape tests*/\*\*/*\*.ts

**Section 30.2: jest (ts-jest)**

[jest](https://facebook.github.io/jest/) is painless JavaScript testing framework by Facebook, with [ts-jest](https://www.npmjs.com/package/ts-jest) can be used to test TypeScript code. To install jest using npm run command

npm **install** --save-dev jest **@**types**/**jest ts-jest typescript

For ease of use install jest as global package

npm **install** -g jest

To make jest work with TypeScript you need to add conﬁguration to package.json

*//package.json*

{

...

"jest": {

"transform": {

".(ts|tsx)": "<rootDir>/node\_modules/ts-jest/preprocessor.js"

},

"testRegex": "(/ tests /.\*|**\\**.(test|spec))**\\**.(ts|tsx|js)$", "moduleFileExtensions": ["ts", "tsx", "js"]

}

}

Now jest is ready. Assume we have sample ﬁzz buz to test

*//fizzBuzz.ts*

export **function** fizzBuzz(n: number): string {

**let** output = "";

**for** (**let** i = 1; i <= n; i++) {

**if** (i % 5 && i % 3) { output += i + ' ';

}

**if** (i % 3 === 0) {

output += 'Fizz ';

}

**if** (i % 5 === 0) {

output += 'Buzz ';

}

}

**return** output;

}

Example test could look like

*//FizzBuzz.test.ts*

*/// <reference types="jest" />*

import {fizzBuzz} from "./fizzBuzz"; test("FizzBuzz test", () =>{

expect(fizzBuzz(2)).toBe("1 2 "); expect(fizzBuzz(3)).toBe("1 2 Fizz ");

});

To execute test run

jest

In output you should see

PASS ./fizzBuzz.test.ts

* FizzBuzz test (3ms)

Test Suites: 1 passed, 1 total

Tests: 1 passed, 1 total

Snapshots: 0 total

Time: 1.46s, estimated 2s Ran all test suites.

##### Code coverage

jest supports generation of code coverage reports.

To use code coverage with TypeScript you need to add another conﬁguration line to package.json.

{

...

"jest": {

...

"testResultsProcessor": "<rootDir>/node\_modules/ts-jest/coverageprocessor.js"

}

}

To run tests with generation of coverage report run

jest --coverage

If used with our sample ﬁzz buzz you should see

PASS ./fizzBuzz.test.ts

* FizzBuzz test (3ms)

-------------|----------|----------|----------|----------|----------------|

File | % Stmts | % Branch | % Funcs | % Lines |Uncovered Lines |

-------------|----------|----------|----------|----------|----------------|

All files | 92.31 | 87.5 | 100 |

fizzBuzz.ts | 92.31 | 87.5 | 100 |

91.67 |

91.67 |

|

13 |

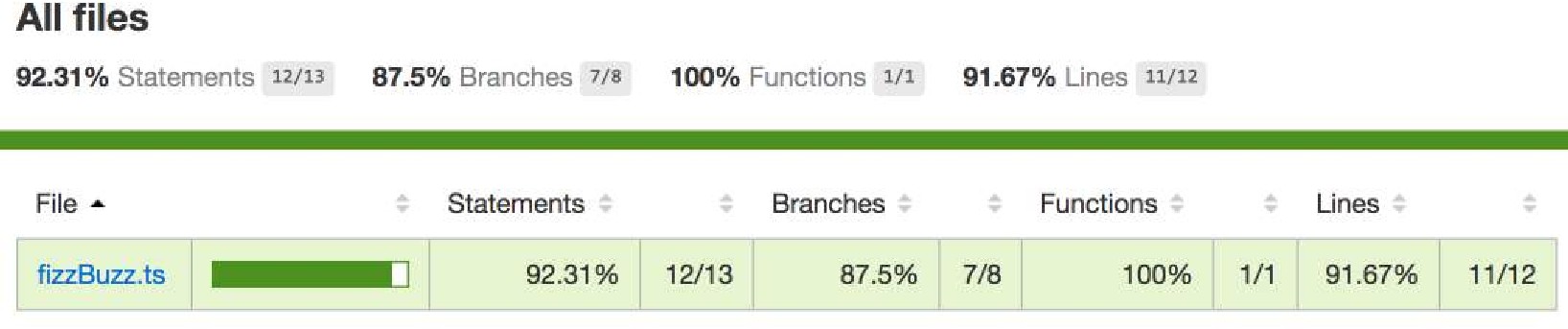
-------------|----------|----------|----------|----------|----------------|

Test Suites: 1 passed, 1 total

Tests: 1 passed, 1 total

Snapshots: 0 total Time: 1.857s Ran all test suites.

jest also created folder coverage which contains coverage report in various formats, including user friendly html report in coverage/lcov-report/index.html



**Section 30.3: Alsatian**

[Alsatian](https://github.com/alsatian-test/alsatian) is a unit testing framework written in TypeScript. It allows for usage of Test Cases, and outputs [TAP-](https://testanything.org/) [compliant](https://testanything.org/) markup.

To use it, install it from npm:

npm **install** alsatian --save-dev

Then set up a test ﬁle:

import { Expect, Test, TestCase } from "alsatian"; import { SomeModule } from "../src/some-module";

export SomeModuleTests { @Test()

public statusShouldBeTrueByDefault() {

**let** instance = **new** SomeModule();

Expect(instance.status).toBe(**true**);

}

@Test("Name should be null by default") public nameShouldBeNullByDefault() {

**let** instance = **new** SomeModule();

Expect(instance.name).toBe(**null**);

}

@TestCase("first name") @TestCase("apples")

public shouldSetNameCorrectly(name: string) {

**let** instance = **new** SomeModule();

instance.setName(name); Expect(instance.name).toBe(name);

}

}

For a full documentation, see [Alsatian's GitHub repo](https://github.com/alsatian-test/alsatian).

**Section 30.4: chai-immutable plugin**

* 1. Install from npm chai, chai-immutable, and ts-node

npm **install** --save-dev chai chai-immutable ts-node

* 1. Install types for mocha and chai

npm **install** --save-dev **@**types**/**mocha **@**types**/**chai

* 1. Write simple test ﬁle:

import {List, Set} from 'immutable'; import \* as chai from 'chai';

import \* as chaiImmutable from 'chai-immutable'; chai.use(chaiImmutable);

describe('chai immutable example', () => { it('example', () => {

expect(Set.of(1,2,3)).to.not.be.empty;

expect(Set.of(1,2,3)).to.include(2);

expect(Set.of(1,2,3)).to.include(5);

})

})

* 1. Run it in the console:

mocha --compilers ts:ts-node/register,tsx:ts-node/register 'test/\*\*/\*.spec.@(ts|tsx)'

**Credits**

Thank you greatly to all the people from Stack Overﬂow Documentation who helped provide this content, more changes can be sent to [web@petercv.com](mailto:web@petercv.com) for new content to be published or updated

[2426021684](https://stackoverflow.com/users/6369276/) Chapters 1, 14 and 16

[ABabin](https://stackoverflow.com/users/5386466/) Chapter 9

[Alec Hansen](https://stackoverflow.com/users/5776572/) Chapter 1

[Alex Filatov](https://stackoverflow.com/users/2173016/) Chapters 22 and 27

[Almond](https://stackoverflow.com/users/1603/) Chapter 14

[Aminadav](https://stackoverflow.com/users/1229624/) Chapter 9

[Aron](https://stackoverflow.com/users/3229534/) Chapter 9

[artem](https://stackoverflow.com/users/43848/) Chapters 9, 14 and 25

[Blackus](https://stackoverflow.com/users/1919388/) Chapter 14

[bnieland](https://stackoverflow.com/users/279393/) Chapter 28

[br4d](https://stackoverflow.com/users/636942/) Chapter 6

[BrunoLM](https://stackoverflow.com/users/340760/) Chapters 1, 17 and 22

[Brutus](https://stackoverflow.com/users/1133284/) Chapter 14

[ChanceM](https://stackoverflow.com/users/6621156/) Chapter 1

[Cobus Kruger](https://stackoverflow.com/users/127826/) Chapter 9

[danvk](https://stackoverflow.com/users/388951/) Chapters 1, 2 and 11

[dimitrisli](https://stackoverflow.com/users/185723/) Chapter 5

[dublicator](https://stackoverflow.com/users/414789/) Chapter 14

[Equiman](https://stackoverflow.com/users/812915/) Chapter 7

[Fenton](https://stackoverflow.com/users/75525/) Chapters 3 and 18

[Florian Hämmerle](https://stackoverflow.com/users/8088868/) Chapter 5

[Fylax](https://stackoverflow.com/users/2752442/) Chapters 1, 3 and 28

[goenning](https://stackoverflow.com/users/396119/) Chapter 28

[hansmaad](https://stackoverflow.com/users/498298/) Chapters 7 and 10

[Harry](https://stackoverflow.com/users/1108828/) Chapter 14

[irakli khitarishvili](https://stackoverflow.com/users/5606083/) Chapters 17 and 26

[islandman93](https://stackoverflow.com/users/3403018/) Chapters 1, 6, 9, 14 and 26

[James Monger](https://stackoverflow.com/users/1916362/) Chapters 7, 27 and 30

[JKillian](https://stackoverflow.com/users/3124288/) Chapters 11 and 14

[Joel Day](https://stackoverflow.com/users/146765/) Chapter 14

[John Ruddell](https://stackoverflow.com/users/2733506/) Chapter 22

[Joshua Breeden](https://stackoverflow.com/users/5423311/) Chapters 1 and 9

[Juliën](https://stackoverflow.com/users/1039247/) Chapters 3 and 28

[Justin Niles](https://stackoverflow.com/users/4655486/) Chapter 7

[k0pernikus](https://stackoverflow.com/users/457268/) Chapters 1 and 27

[Kevin Montrose](https://stackoverflow.com/users/80572/) Chapters 5, 12 and 19

[Kewin Dousse](https://stackoverflow.com/users/1841827/) Chapter 22

[KnottytOmo](https://stackoverflow.com/users/2440747/) Chapters 1, 10 and 14

[Kuba Beránek](https://stackoverflow.com/users/1107768/) Chapter 1

[Lekhnath](https://stackoverflow.com/users/2071612/) Chapter 1

[leonidv](https://stackoverflow.com/users/224222/) Chapter 30

[lilezek](https://stackoverflow.com/users/2832398/) Chapter 23

[Magu](https://stackoverflow.com/users/4516689/) Chapters 3, 27 and 28

[Matt Lishman](https://stackoverflow.com/users/3379536/) Chapter 1

[Matthew Harwood](https://stackoverflow.com/users/1191635/) Chapter 30

[Mikhail](https://stackoverflow.com/users/5526354/) Chapters 1 and 3

[mleko](https://stackoverflow.com/users/3552932/) Chapters 1, 15, 22, 27 and 30

[muetzerich](https://stackoverflow.com/users/4446383/) Chapter 6

[Muhammad Awais](https://stackoverflow.com/users/3901944/) Chapter 10

[Paul Boutes](https://stackoverflow.com/users/4844421/) Chapter 9

[Peopleware](https://stackoverflow.com/users/1649124/) Chapter 29

[Rahul](https://stackoverflow.com/users/1135770/) Chapters 20 and 21

[Rajab Shakirov](https://stackoverflow.com/users/3914072/) Chapters 14 and 26

[RationalDev](https://stackoverflow.com/users/2800482/) Chapters 1 and 3

[Remo H. Jansen](https://stackoverflow.com/users/606821/) Chapter 8

[Robin](https://stackoverflow.com/users/371530/) Chapter 7

[Roman M. Koss](https://stackoverflow.com/users/1181479/) Chapter 24

[Roy Dictus](https://stackoverflow.com/users/651188/) Chapter 1

[Saiful Azad](https://stackoverflow.com/users/1919324/) Chapters 1 and 9

[Sam](https://stackoverflow.com/users/721250/) Chapter 1

[samAlvin](https://stackoverflow.com/users/6638533/) Chapter 1

[SilentLupin](https://stackoverflow.com/users/2348288/) Chapter 6

[Slava Shpitalny](https://stackoverflow.com/users/4225948/) Chapters 6, 9, 10 and 14

[smnbbrv](https://stackoverflow.com/users/1990451/) Chapter 5

[Stefan Rein](https://stackoverflow.com/users/4641479/) Chapter 24

[Sunnyok](https://stackoverflow.com/users/4290193/) Chapter 9

[Taytay](https://stackoverflow.com/users/544130/) Chapter 10

[Udlei Nati](https://stackoverflow.com/users/3583600/) Chapter 4

[user3893988](https://stackoverflow.com/users/3893988/) Chapter 28

[vashishth](https://stackoverflow.com/users/1568755/) Chapter 13

[Wasabi Fan](https://stackoverflow.com/users/2422874/) Chapter 1

**You may also like**

