LEARNING

TypeScript

#typescript

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: typescript

It is an unofficial and free TypeScript ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official TypeScript.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with TypeScript

Remarks

TypeScript aims to be a superset of JavaScript that transpiles to JavaScript. By generating ECMAScript compliant code, TypeScript can introduce new language features while retaining compatibility with existing JavaScript engines. ES3, ES5 and ES6 are currently supported targets.

Optional types are a primary feature. Types allow static checking with the goal of finding errors early and can enhance tooling with features like code refactoring.

TypeScript is an open source and cross platform programming language developed by Microsoft. The source code is available on GitHub.

Versions

Version	Release Date
2.4.1	2017-06-27
2.3.2	2017-04-28
2.3.1	2017-04-25
2.3.0 beta	2017-04-04
2.2.2	2017-03-13
2.2	2017-02-17
2.1.6	2017-02-07
2.2 beta	2017-02-02
2.1.5	2017-01-05
2.1.4	2016-12-05
2.0.8	2016-11-08
2.0.7	2016-11-03
2.0.6	2016-10-23
2.0.5	2016-09-22
2.0 Beta	2016-07-08

Version	Release Date
1.8.10	2016-04-09
1.8.9	2016-03-16
1.8.5	2016-03-02
1.8.2	2016-02-17
1.7.5	2015-12-14
1.7	2015-11-20
1.6	2015-09-11
1.5.4	2015-07-15
1.5	2015-07-15
1.4	2015-01-13
1.3	2014-10-28
1.1.0.1	2014-09-23

Examples

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.
- `Visual Studio 2013 Update 2` or later includes TypeScript, or you can download TypeScript for earlier versions.

Visual Studio Code

- Visual Studio Code (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` comes with TypeScript and a built-in compiler. [Webstorm is not free]

IntelliJ IDEA

- `IntelliJ IDEA 2016.2` has support for Typescript and a compiler via a plugin maintained by the Jetbrains team. [IntelliJ is not free]

Atom & atom-typescript

- `Atom` supports TypeScript with the atom-typescript package.

Sublime Text

- `Sublime Text` supports TypeScript with the typescript package.

Installing the command line interface

Install Node.js

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` 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.

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.

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
- {[key: string]: number} - a dictionary of numbers indexed by string
- enum - { Red = 0, Blue, Green } - enumeration mapped to numbers
- Function. You specify types for the parameters and return value:
  - (param: number) => string - function taking one number parameter returning string
  - () => number - function with no parameters returning an number.
  - (a: string, b?: boolean) => void - function taking a string and optionally a boolean with no return 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 void type.
- 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 thrown 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.

Classes

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

TypeScript REPL in Node.js

For use TypeScript REPL in Node.js you can use tsun package

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
```

Running TypeScript using ts-node

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.

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.

Read Getting started with TypeScript online: https://riptutorial.com/typescript/topic/764/getting-started-with-typescript

Chapter 2: Arrays

Examples

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 }
```

Read Arrays online: https://riptutorial.com/typescript/topic/9562/arrays

Chapter 3: Class Decorator

Parameters

Parameter	Details
target	The class being decorated

Examples

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;
    }
}
```

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));
```

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" }
```

Read Class Decorator online: https://riptutorial.com/typescript/topic/4592/class-decorator

Chapter 4: Classes

Introduction

TypeScript, like ECMA Script 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.

Examples

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
```

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`.

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
```

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
```

Abstract Classes

```
abstract class Machine {
    constructor(public manufacturer: string) {
    }

    // An abstract class can define methods of it's 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**.

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"
```

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.
- 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.

Read Classes online: https://riptutorial.com/typescript/topic/1560/classes

Chapter 5: Configure typescript project to compile all files in typescript.
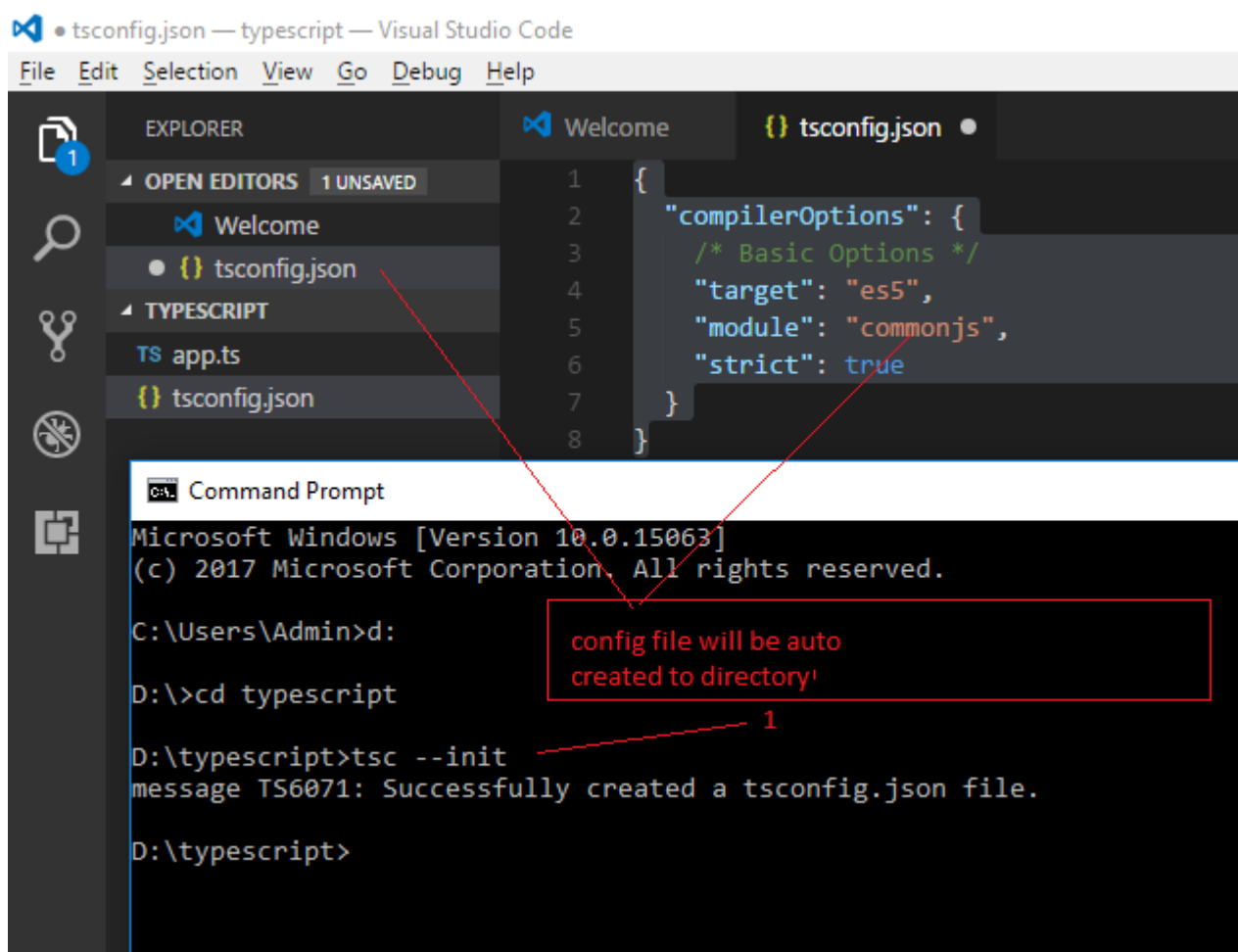
Introduction

creating your first .tsconfig configuration file which will tell the TypeScript compiler how to treat your .ts files

Examples

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.
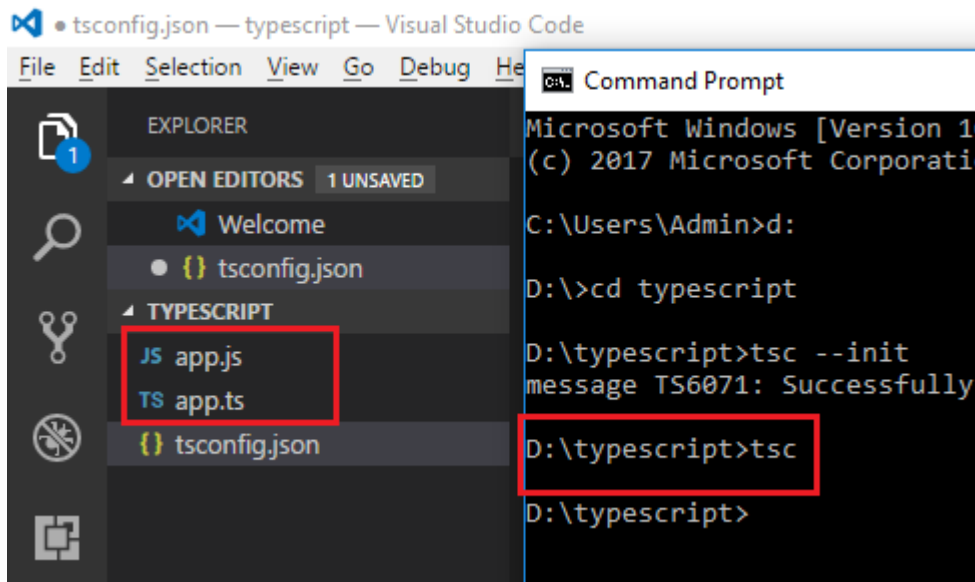
- If you will create another typescript and hit "tsc" command in command prompt or terminal javascript file will be automatically created for typescript file.

Thank you,

Read Configure typescript project to compile all files in typescript. online:
https://riptutorial.com/typescript/topic/10537/configure-typescript-project-to-compile-all-files-in-typescript-

Chapter 6: Debugging

Introduction

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

This article describes both ways using Visual Studio Code and WebStorm. All examples presume that your main file is *index.ts*.

Examples

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 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.

JavaScript with SourceMaps in WebStorm

Create a *Node.js* debug configuration and use `index.js` as *Node parameters*.
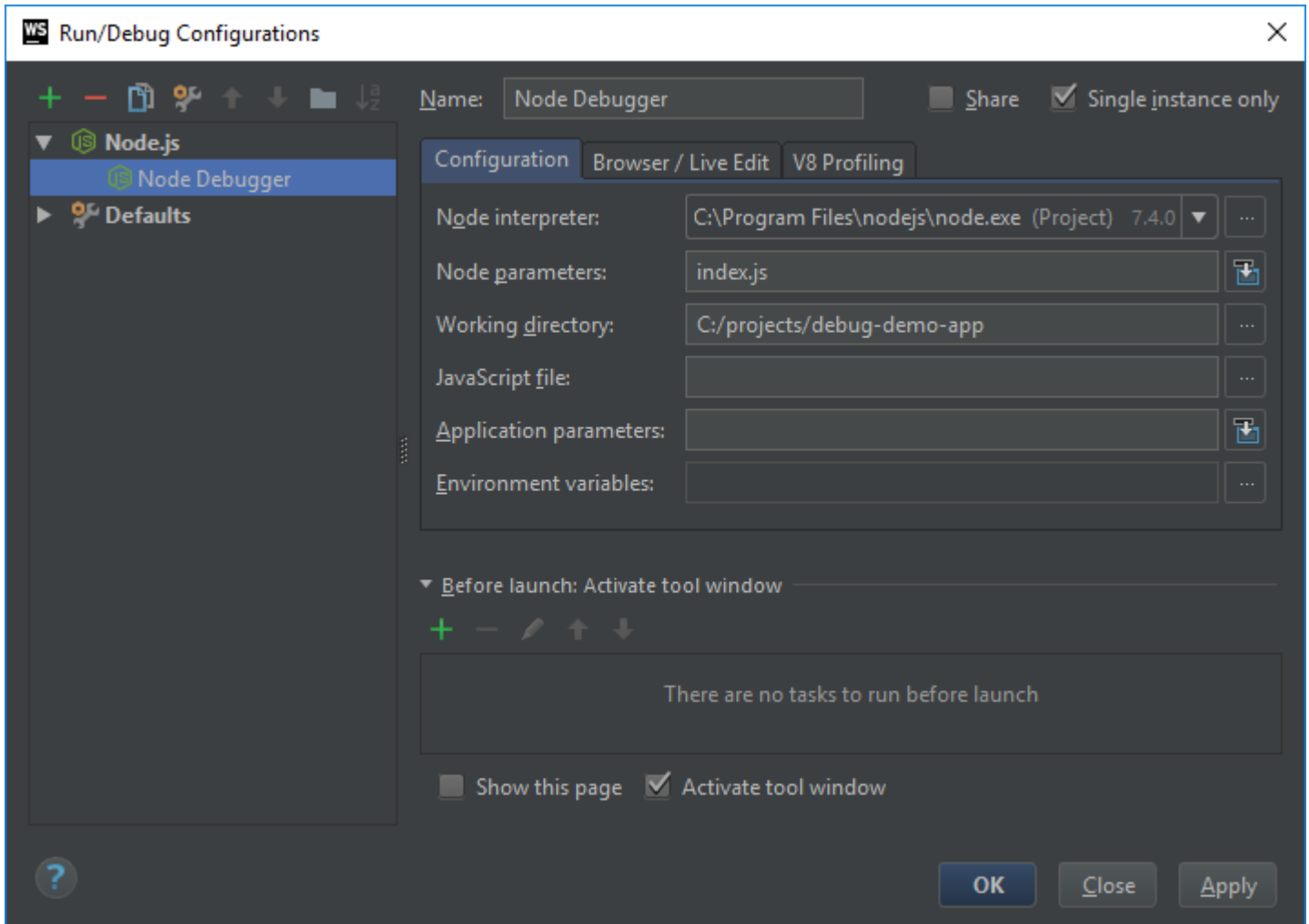
TypeScript with ts-node in Visual Studio Code

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
        }
    ]
}
```

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:
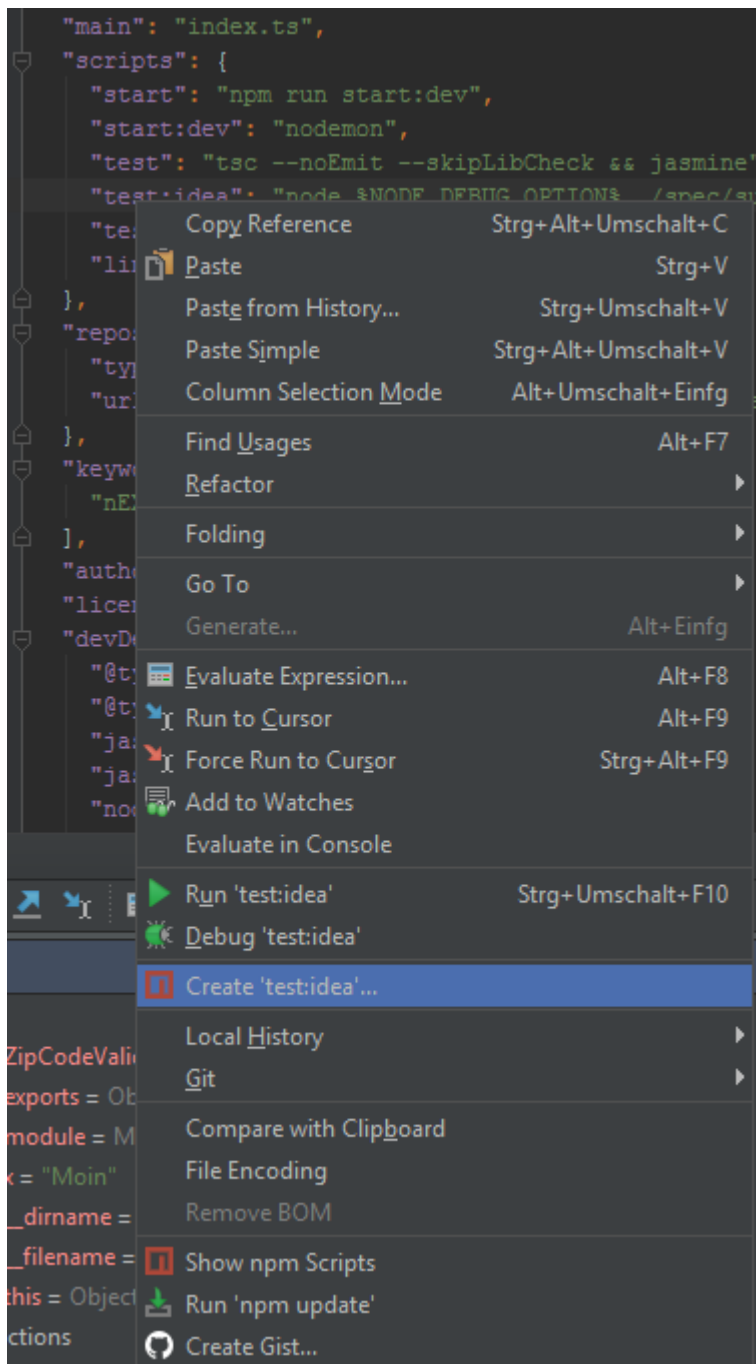
Start the debugger using this configuration:

Read Debugging online: https://riptutorial.com/typescript/topic/9131/debugging

---
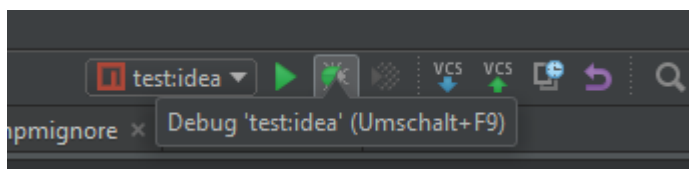
Chapter 7: Enums

Examples

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
```

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:

```
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 = {}));
```

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');
```

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));
```

Chapter 8: Functions

Remarks

Typescript documentation link for Functions

Examples

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'
```

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; };
```

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);
}
```

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)          //'1:30'
whatTime('1',30)        //'1:30'
```

```
whatTime(1,'30')        //'1:30'
whatTime('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;
    }
}
```

Read Functions online: https://riptutorial.com/typescript/topic/1841/functions

Chapter 9: Generics

Syntax

- The generic types declared within the triangle brackets: `<T>`
- Constrainting the generic types is done with the extends keyword: `<T extends Car>`

Remarks

The generic parameters are not available at runtime, they are just for the compile time. This means you can't do something like this:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (T instanceof Executable1) {    // Compilation error
            ...
        } else if (U instanceof Executable2){    // Compilation error
            ...
        }
    }
}
```

However, class information is still preserved, so you can still test for the type of a variable as you have always been able to:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (executable instanceof Executable1) {
            ...
        } else if (executable instanceof Executable2){
            ...
        } // But in this method, since there is no parameter of type `U` it is non-sensical to
ask about U's "type"
    }
}
```

Examples

Generic Interfaces

Declaring a generic interface

```
interface IResult<T> {
    wasSuccessfull: 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>{
    wasSuccessfull: 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);
    }
}
```

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
```

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;
}
```

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) {
    }
}
```

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);
```

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
```

Chapter 10: How to use a javascript library without a type definition file

Introduction

While some existing JavaScript libraries have type definition files, there are many that don't.

TypeScript offers a couple patterns to handle missing declarations.

Examples

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), you could put

```
declare var $: any;
```

Now any use of $ will be typed `any`.

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) 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();
```

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`

Read How to use a javascript library without a type definition file online:
https://riptutorial.com/typescript/topic/8249/how-to-use-a-javascript-library-without-a-type-definition-file

Chapter 11: Importing external libraries

Syntax

- import {component} from 'libName'; // Will import the class "component"
- import {component as c} from 'libName'; // Will import the class "component" into a "c" object
- import component from 'libname'; // Will import the default export from libName
- import * as lib from 'libName'; // Will import everything from libName into a "lib" object
- import lib = require('libName'); // Will import everything from libName into a "lib" object
- const lib: any = require('libName'); // Will import everything from libName into a "lib" object
- import 'libName'; // Will import libName module for its side effects only

Remarks

It might seem that the syntax

```
import * as lib from 'libName';
```

and

```
import lib = require('libName');
```

are the same thing, but they are not!

Let us consider that we want to import a class **Person** exported with TypeScript-specific `export =` syntax :

```
class Person {
...
}
export = Person;
```

In this case it is not possible to import it with es6 syntax (we would get an error at compile time), TypeScript-specific `import =` syntax must be used.

```
import * as Person from 'Person';  //compile error
import Person = require('Person');  //OK
```

The converse is true: classic modules can be imported with the second syntax, so, in a way, the last syntax is more powerful since it is able to import all exports.

For more information see the official documentation.

Examples

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";
```

Finding definition files

for typescript 2.x:

definitions from DefinitelyTyped are available via @types npm 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 is an npm package that can automatically install type definition files into a local project. I recommend that you read the quickstart.

```
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                SOURCE HOMEPAGE                                  DESCRIPTION
VERSIONS UPDATED
lodash              dt     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                1
2016-07-01T20:51:07.000Z
```

2. Then decide which source you should install from. I use dt which stands for DefinitelyTyped a GitHub repo where the community can edit typings, it's also normally the most recently updated.

3. 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.

4. 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
```

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;
```

Finding definition files with typescript 2.x

With the 2.x versions of typescript, typings are now available from the npm @types repository. 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'
```

Read Importing external libraries online: https://riptutorial.com/typescript/topic/1542/importing-external-libraries

Chapter 12: Integrating with Build Tools

Remarks

For more information you can go on official web page typescript integrating with build tools

Examples

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/
    }],
  }
};
```

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

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

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

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.

Alternatives:

- awesome-typescript-loader

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">
  <!-- Include default props at the bottom -->
  <Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft

Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript
/>
```

```
  <!-- 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\Microsoft

Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript
/>
</Project>
```

More details about defining MSBuild compiler options: Setting Compiler Options in MSBuild
projects

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 and using nightly builds with NuGet

Read Integrating with Build Tools online: https://riptutorial.com/typescript/topic/2860/integrating-
with-build-tools

Chapter 13: Interfaces

Introduction

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.

Syntax

- interface InterfaceName {
-   parameterName: parameterType;
-   optionalParameterName?: parameterType;
- }

Remarks

Interfaces vs Type Aliases

Interfaces are good for specifying the shape of an object, eg for a person object you could specify

```
interface person {
    id?: number;
    name: string;
    age: number;
}
```

However what if you want to represent, say, the way a person is stored in an SQL database? Seeing as each DB entry consists of a row of shape `[string, string, number]` (so an array of strings or numbers), there is no way you could represent this as an object shape, because the row doesn't have any *properties* as such, it's just an array.

This is an occasion where types come in useful. Instead of specifying in every function that accepts a row parameter `function processRow(row: [string, string, number])`, you can create a separate type alias for a row and then use that in every function:

```
type Row = [string, string, number];
function processRow(row: Row)
```

Official interface documentation

Examples

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 for more details.

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.

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.

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!

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 `ISatus<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.

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. There 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 Inferade, Bluetooth5 and whatsoever by just implementing `Connector` interface.

This is called 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.

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.

Read Interfaces online: https://riptutorial.com/typescript/topic/2023/interfaces

Chapter 14: Mixins

Syntax

- class BeetleGuy implements Climbs, Bulletproof { }
- applyMixins (BeetleGuy, [Climbs, Bulletproof]);

Parameters

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

Remarks

There are three rules to bear in mind with mixins:

- You use the `implements` keyword, not the `extends` keyword when you write your composition class
- You need to have a matching signature to keep the compiler quiet (but it doesn't need any real implementation – it will get that from the mixin).
- You need to call `applyMixins` with the correct arguments.

Examples

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];
            }
        });
    });
}
```

Read Mixins online: https://riptutorial.com/typescript/topic/4727/mixins

Chapter 15: Modules - exporting and importing

Examples

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");                            // Hello World!
helloES("Mundo");                          // Hola Mundo!
defaultHello("World");                     // Hello World!

Bundle.hello("World");                     // Hello World!
Bundle.helloES("Mundo");                   // Hola Mundo!

welcome("Human");                          // Welcome Human!
```

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
```

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
```

Read Modules - exporting and importing online:
https://riptutorial.com/typescript/topic/9054/modules---exporting-and-importing

Chapter 16: Publish TypeScript definition files

Examples

Include definition file with library on npm

Add typings to your package.json

```
{
...
"typings": "path/file.d.ts"
...
}
```

Now when ever that library is imported typescript will load the typings file

Read Publish TypeScript definition files online: https://riptutorial.com/typescript/topic/2931/publish-typescript-definition-files

Chapter 17: Strict null checks

Examples

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);
```

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;
}
```

Read Strict null checks online: https://riptutorial.com/typescript/topic/1727/strict-null-checks

Chapter 18: tsconfig.json

Syntax

- Uses JSON file format
- Can also accept JavaScript style comments

Remarks

Overview

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

Using tsconfig.json

- By invoking tsc with no input files, in which case the compiler searches for the tsconfig.json file starting in the current directory and continuing up the parent directory chain.
- By invoking tsc with no input files and a --project (or just -p) command line option that specifies the path of a directory containing a tsconfig.json file. When input files are specified on the command line, tsconfig.json files are

Details

The `"compilerOptions"` property can be omitted, in which case the compiler's defaults are used. See our full list of supported Compiler Options.

If no `"files"` property is present in a tsconfig.json, the compiler defaults to including all TypeScript (*.ts or *.tsx) files in the containing directory and subdirectories. When a "files" property is present, only the specified files are included.

If the `"exclude"` property is specified, the compiler includes all TypeScript (*.ts or *.tsx) files in the containing directory and subdirectories except for those files or folders that are excluded.

The `"files"` property cannot be used in conjunction with the "exclude" property. If both are specified then the "files" property takes precedence.

Any files that are referenced by those specified in the `"files"` property are also included. Similarly, if a file B.ts is referenced by another file A.ts, then B.ts cannot be excluded unless the referencing file A.ts is also specified in the "exclude" list.

A `tsconfig.json` file is permitted to be completely empty, which compiles all files in the containing directory and subdirectories with the default compiler options.

Compiler options specified on the command line override those specified in the tsconfig.json file.

Schema

Schema can be found at: http://json.schemastore.org/tsconfig

Examples

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
    }
}
```

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
- JetBrains WebStorm
- Atom with atom-typescript

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 */
```

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
    "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.

preserveConstEnums

Typescript supports costant enumerables, declared through `const enum`.

This is usually just syntax sugar as the costant 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 perfomance benefit from inlining, you may prefer to keep enums even if costant (ie:
you may wish readability on development code), to do this you have to set in **tsconfig.json** the
`preserveConstEnums` clausole 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
```

Read tsconfig.json online: https://riptutorial.com/typescript/topic/4720/tsconfig-json

Chapter 19: TSLint - assuring code quality and consistency

Introduction

TSLint performs static analysis of code and detect errors and potential problems in code.

Examples

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"]
    }
}
```

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 Contributors.

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, // Diallows 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).
  }
}
```

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 and source code of predfined ruleset

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
  }
}
```

Installation and setup

To install 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
```

Sets of TSLint Rules

- tslint-microsoft-contrib
- tslint-eslint-rules
- codelyzer

Yeoman genearator supports all these presets and can be extends also:

- generator-tslint

Read TSLint - assuring code quality and consistency online:
https://riptutorial.com/typescript/topic/7457/tslint---assuring-code-quality-and-consistency

Chapter 20: Typescript basic examples

Remarks

This is a basic example which extends a generic car class and defines a car description method.

Find more TypeScript examples here - TypeScript Examples GitRepo

Examples

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();
```

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);
```

Read Typescript basic examples online: https://riptutorial.com/typescript/topic/7721/typescript-basic-examples

Chapter 21: TypeScript Core Types

Syntax

- let variableName: VariableType;
- function functionName(parameterName: VariableType, parameterWithDefault: VariableType = ParameterDefault, optionalParameter?: VariableType, ...variardicParameter: VariableType[]): ReturnType { /*...*/};

Examples

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;
```

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
```

String

Textual data type:

```
let singleQuotes: string = 'single';
let doubleQuotes: string = "double";
let templateString: string = `I am ${ singleQuotes }`; // I am single
```

Array

An array of values:

```
let threePigs: number[] = [1, 2, 3];
let genericStringArray: Array<string> = ['first', '2nd', '3rd'];
```

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 }
```

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
```

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`.

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'
```

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 arguments `x` 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

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!
```

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 a 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 don't 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";
}
```

```typescript
function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet) {
    console.log(`Hey ${pet.name}, let's 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, let's play.
//         Rocky eats.
//         Rocky walks.
//         Rocky sleeps.
```

Intersection Types

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

```typescript
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();
}
```

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, specifig
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 normale 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 normale 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"
```

Read TypeScript Core Types online: https://riptutorial.com/typescript/topic/2776/typescript-core-types

Chapter 22: TypeScript with AngularJS

Parameters

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 constuctor functions.

Remarks

While doing the directive in TypeScript, keep in mind, that power of this language of custom type and interfaces that you can create. This is extremely helpfull when developing huge applications. Code completion supported by many IDE will show you the possible value by corresponding type you are working with, so there is far more less that should be kept in mind (comparing to VanillaJS).

"Code Against Interfaces, Not Implementations"

Examples

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 controll.
 * For example we have all templates in one value, and we wan't 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);
```

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
    ]);
```

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>
```

Read TypeScript with AngularJS online: https://riptutorial.com/typescript/topic/6569/typescript-with-angularjs

Chapter 23: TypeScript with SystemJS

Examples

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:

```html
<!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>
```

Read TypeScript with SystemJS online: https://riptutorial.com/typescript/topic/6664/typescript-with-systemjs

Chapter 24: Typescript-installing-typescript-and-running-the-typescript-compiler-tsc

Introduction

How to install TypeScript and run the TypeScript compiler against a .ts file from the command line.

Examples

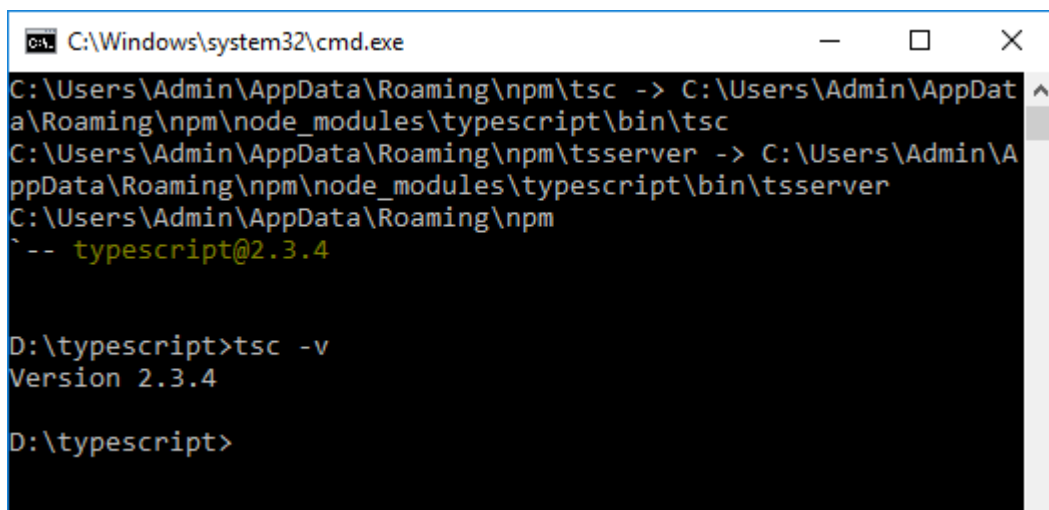### Steps.

Installing Typescript and running typescript compiler.

To install Typescript Comiler

```
npm install -g typescript
```

To check with the typescript version
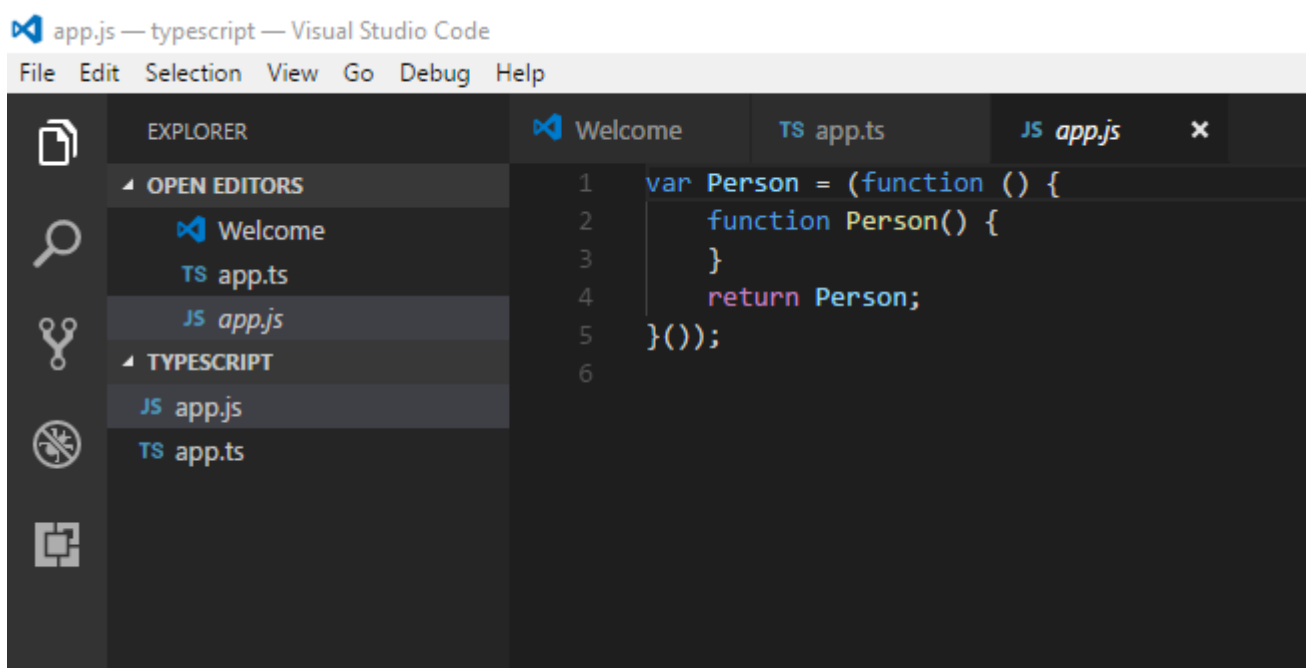
```
tsc -v
```


Download Visual Studio Code for Linux/Windows

Visual Code Download Link

1. Open Visual Studio Code
2. Open Same Folde 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.
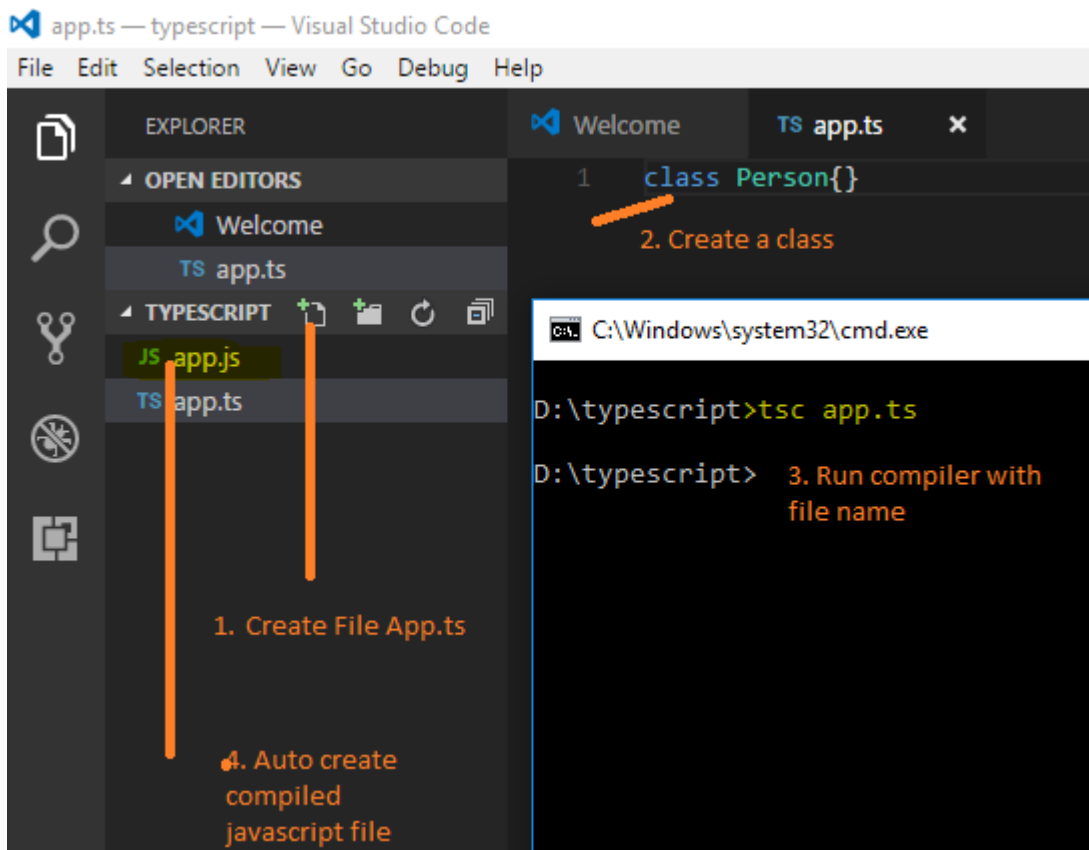
See the result in compiled javascript of written typescript code.

Thank you.

Chapter 25: Unit Testing

Examples

Alsatian

Alsatian is a unit testing framework written in TypeScript. It allows for usage of Test Cases, and outputs TAP-compliant 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.

chai-immutable plugin

1. Install from npm chai, chai-immutable, and ts-node

```
npm install --save-dev chai chai-immutable ts-node
```

2. Install types for mocha and chai

```
npm install --save-dev @types/mocha @types/chai
```

3. 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);
  })
})
```

4. Run it in the console:

```
mocha --compilers ts:ts-node/register,tsx:ts-node/register 'test/**/*.spec.@(ts|tsx)'
```

tape

tape is minimalistic JavaScript testing framework, it outputs TAP-compliant 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
```

jest (ts-jest)

jest is painless JavaScript testing framework by Facebook, with 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    |  91.67   |                |
 fizzBuzz.ts |  92.31   |  87.5    |   100    |  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`

All files

92.31% Statements 12/13   **87.5%** Branches 7/8   **100**

File ▲		Statements	
fizzBuzz.ts		92.31%	12/1

Chapter 26: User-defined Type Guards

Syntax

- typeof x === "type name"
- x instanceof TypeName
- function(foo: any): foo is TypeName { /* code returning boolean */ }

Remarks

Using type annotations in TypeScript constrains the possible types your code will need to deal with, but it is still common to need to take different code paths based on the runtime type of a variable.

Type guards let you write code that discriminates based on the runtime type of a variable, while remaining strongly typed and avoiding casts (also known as type assertions).

Examples

Using instanceof

`instanceof` requires that the variable is of type `any`.

This code (try it):

```
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
meom
```

to the console.

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):

```
function example(foo: any) {
    if (typeof foo === "number") {
        // foo is type number in this block
        console.log(foo + 100);
    }

    if (typeof foo === "string") {
        // fooi is type string in this block
        console.log("not a number: " + foo);
    }
}

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

prints

```
123
not a number: foo
```

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):

```
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 it's 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);
    }
}
```

Chapter 27: Using Typescript with React (JS & native)

Examples

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:

1) 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

2) 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.

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](#)

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

Read Using Typescript with React (JS & native) online:
https://riptutorial.com/typescript/topic/1835/using-typescript-with-react--js---native-

Chapter 28: Using Typescript with RequireJS

Introduction

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.

Examples

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"></script>
    <script>
      function __init() {
        require(["view/index.js"]);
      }
    </script>
</body>
```

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",
    ...
}
```

Read Using Typescript with RequireJS online: https://riptutorial.com/typescript/topic/10773/using-typescript-with-requirejs

Chapter 29: Using TypeScript with webpack

Examples

webpack.config.js

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

tsconfig.json

```json
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react" // if you want to use react jsx
  }
}
```

```js
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"
    // },
};
```

Read Using TypeScript with webpack online: https://riptutorial.com/typescript/topic/2024/using-typescript-with-webpack

Chapter 30: Why and when to use TypeScript

Introduction

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).

Remarks

The merits of typed vs. untyped languages have been debated for decades. Arguments for static types include:

1. Safety: type systems allow many errors to be caught early, without running the code. TypeScript can be configured to allow fewer programming errors
2. Readability: explicit types make code easier for humans to understand. As Fred Brooks wrote, "Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious."
3. Tooling: type systems make code easier for computers to understand. This allows tools like IDEs and linters to be more powerful.
4. Performance: type systems make code run faster by reducing the need for runtime type checking.

Because TypeScript's output is independent of its types, TypeScript has no impact on performance. The argument for using TypeScript rests on the other three advantages.

Arguments against type systems include:

1. Added complexity: type systems can be more complex than the language runtime that they describe. Higher order functions can be easy to implement correctly but difficult to type. Dealing with type definitions creates additional barriers to using external libraries.
2. Added verbosity: type annotations can add boilerplate to your code, making the underlying logic harder to follow.
3. Slower iteration: by introducing a build step, TypeScript slows down the edit/save/reload cycle.
4. Incompleteness: A type system cannot be both sound and complete. There are correct programs which TypeScript does not allow. And programs which TypeScript accepts can still contain bugs. A type system doesn't alleviate the need for testing. If you use TypeScript, you may have to wait longer to use new ECMAScript language features.

TypeScript offers some ways to address all of these issues:

1. Added complexity. If typing part of a program is too difficult, TypeScript can be largely

disabled using an opaque `any` type. The same is true for external modules.

2. Added verbosity. This can partially be addressed through type aliases and TypeScript's ability to infer types. Writing clear code is as much an art as a science: remove too many type annotations and the code may no longer be clear to human readers.

3. Slower iteration: A build step is relatively common in modern JS development and TypeScript already integrates with most build tools. And if TypeScript catches an error early, it can save you an entire iteration cycle!

4. Incompleteness. While this problem cannot be completely solved, TypeScript has been able to capture more and more expressive JavaScript patterns over time. Recent examples include the addition of mapped types in TypeScript 2.1 and mixins in 2.2.

The arguments for and against type systems in general apply equally well to TypeScript. Using TypeScript increases the overhead to starting a new project. But over time, as the project increases in size and gains more contributors, the hope is that the pros of using it (safety, readability, tooling) become stronger and outweigh the cons. This is reflected in TypeScript's motto: "JavaScript that scales."

Examples
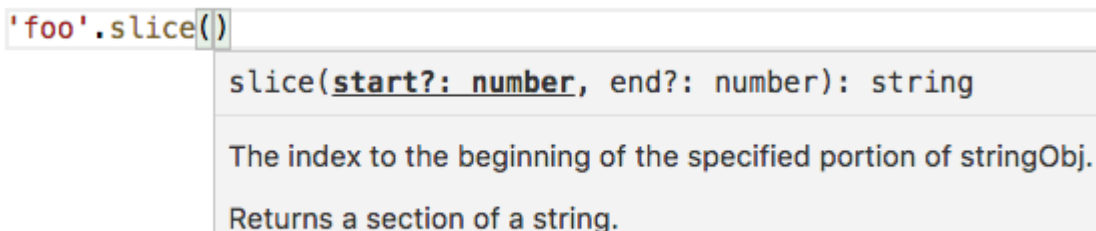
Safety

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'.
```

Readability

TypeScript enables editors to provide contextual documentation:

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

Tooling

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

```
let foo = '123';

{
  const foo = (x: number) => {
    return 2 * x;
  }

  foo(2);
}
```

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.

Read Why and when to use TypeScript online: https://riptutorial.com/typescript/topic/9073/why-and-when-to-use-typescript

Credits

S. No	Chapters	Contributors
1	Getting started with TypeScript	2426021684, Alec Hansen, Blackus, BrunoLM, cdbajorin, ChanceM, Community, danvk, Florian Hämmerle, Fylax, goenning, islandman93, jengeb, Joshua Breeden, k0pernikus, Kiloku, KnottytOmo, Kuba Beránek, Lekhnath, Matt Lishman, Mikhail, mleko, RationalDev, Roy Dictus, Saiful Azad, Sam, samAlvin, Wasabi Fan, zigzag
2	Arrays	Udlei Nati
3	Class Decorator	bruno, Remo H. Jansen, Stefan Rein
4	Classes	adamboro, apricity, Cobus Kruger, Equiman, hansmaad, James Monger, Jeff Huijsmans, Justin Niles, KnottytOmo, Robin
5	Configure typescript project to compile all files in typescript.	Rahul
6	Debugging	Peopleware
7	Enums	dimitrisli, Florian Hämmerle, Kevin Montrose, smnbbrv
8	Functions	br4d, hansmaad, islandman93, KnottytOmo, muetzerich, SilentLupin, Slava Shpitalny
9	Generics	danvk, hansmaad, KnottytOmo, Mathias Rodriguez, Muhammad Awais, Slava Shpitalny, Taytay
10	How to use a javascript library without a type definition file	Bruno Krebs, Kevin Montrose
11	Importing external libraries	2426021684, Almond, artem, Blackus, Brutus, Dean Ward, dublicator, Harry, islandman93, JKillian, Joel Day, KnottytOmo, lefb766, Rajab Shakirov, Slava Shpitalny, tuvokki
12	Integrating with Build Tools	Alex Filatov, BrunoLM, Dan, dublicator, John Ruddell, mleko, Protectator, smnbbrv, void
13	Interfaces	ABabin, Aminadav, Aron, artem, Cobus Kruger, Fabian Lauer, islandman93, Joshua Breeden, Paul Boutes, Robin, Saiful Azad

		, Slava Shpitalny, Sunnyok
14	Mixins	Fenton
15	Modules - exporting and importing	mleko
16	Publish TypeScript definition files	2426021684
17	Strict null checks	bnieland, danvk, JKillian, Yaroslav Admin
18	tsconfig.json	bnieland, Fylax, goenning, Magu, Moriarty, user3893988
19	TSLint - assuring code quality and consistency	Alex Filatov, James Monger, k0pernikus, Magu, mleko
20	Typescript basic examples	vashishth
21	TypeScript Core Types	dublicator, Fenton, Fylax, Magu, Mikhail, Moriarty, RationalDev
22	TypeScript with AngularJS	Chic, Roman M. Koss, Stefan Rein
23	TypeScript with SystemJS	artem
24	Typescript-installing-typescript-and-running-the-typescript-compiler-tsc	Rahul
25	Unit Testing	James Monger, leonidv, Louie Bertoncin, Matthew Harwood, mleko
26	User-defined Type Guards	Kevin Montrose
27	Using Typescript with React (JS & native)	Aleh Kashnikau, irakli khitarishvili, islandman93, Rajab Shakirov, tBX
28	Using Typescript with RequireJS	lilezek
29	Using TypeScript	BrunoLM, irakli khitarishvili, John Ruddell

	with webpack	
30	Why and when to use TypeScript	danvk