TypeScript

Introduction

TypeScript is a superset of Javascript that tightens up some of the fast-and-loose paradigms that make Javascript error-prone and difficult to use. For example, TypeScript gives you the ability to use…

- statically type variables / strong typing
- generics
- enums
- interfaces
- classes (explicitly defined constructors, fields, methods, and visibility)
- packages (called namespaces/modules in TypeScript)
- lambdas (called arrow functions in TypeScript)

The upside to all of this is that your code will be more maintainable and less error-prone. It's also more friendly for developers coming from other languages (e.g. Java) and allows for better tooling support (e.g. better Intellisense support and more immediate feedback on warnings and errors).

The downside is that you may lose access to some of the features of Javascript that other normal Javascript developers use regularly.

Ultimately, your TypeScript code gets “compiled” down to pure Javascript. Since TypeScript is a superset of Javascript, you can still insert Javascript directly into your TypeScript code and call Javascript from TypeScript.

Setup

The following subsections discuss common setup and usage instructions for TypeScript.

Install

The easiest way to install TypeScript is via the Node Package Manager (npm): npm install -g typescript...

```
~/test $ npm install -g typescript
/home/user/.nvm/versions/node/v8.9.1/bin/tsc ->
/home/user/.nvm/versions/node/v8.9.1/lib/node_modules/typescript/bin/tsc
/home/user/.nvm/versions/node/v8.9.1/bin/tsserver ->
/home/user/.nvm/versions/node/v8.9.1/lib/node_modules/typescript/bin/tsserv
er
+ typescript@2.6.2
added 1 package in 1.66s
```

> **NOTE**: If you need a refresher on NodeJS or npm, check out the other documents. You may need to run as sudo.

Once installed, you can manually run the TypeScript compiler via tsc...

```
~/test $ tsc
Version 2.6.2
Syntax:   tsc [options] [file ...]

Examples: tsc hello.ts
          tsc --outFile file.js file.ts
          tsc @args.txt
```

IDEs

Instead of compiling manually, you’d be better off using an IDE. VSCode seems to be main IDE to use. I’ve tried plugins for Eclipse but they don’t seem to work, and the plugin for NetBeans looks to be missing features.

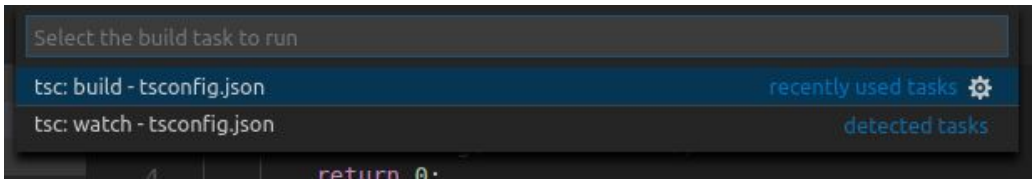
NOTE: You may have issues setting up VSCode with TypeScript if you're using node version manager.

To setup VSCode to work with typescript, you first need to have a tsconfig.json file in the root of your project. This is where you define your configuration options for the TypeScript compiler…

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

NOTE: tsconfig.json is used to configure the TypeSciprt compiler for a project. It's typically placed in the root of a project. A full list of options for tsconfig.json can be found at https://www.typescriptlang.org/docs/handbook/tsconfig-json.html, but the example above is a good base to work off of. It may be a good idea to set the allowJs option to false and the out option to some subdir where the resulting js files can be dumped.

Once that's up, in VSCode you can hit Ctrl+Shift+B and get a list of self-explanatory tasks that you can run…



If you try to Debug your code from VSCode (F5), you'll get a .vscode/launch.json file in your root folder with launch configurations. You can add to this file if you want to use different TypeScript files to debug…

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit:
https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [

        {
            "type": "node",
            "request": "launch",
```

```
            "name": "Launch Program",
            "program": "${workspaceFolder}/helloWorld.ts",
            "outFiles": [
                "${workspaceFolder}/**/*.js"
            ]
        },
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Test Script",
            "program": "${workspaceFolder}/test.ts",
            "outFiles": [
                "${workspaceFolder}/**/*.js"
            ]
        }
    ]
}
```

Example

The following is a simple example of TypeScript code...

```
class TestClass {
    name: String;
    count: Number;
    public constructor (name: string, count: number) {
        this.name = name;
        this.count = count;
    }

    public output() {
        console.log(this.name + ' ' + this.count);
    }
}

new TestClass('nametest', 99).output();
```

The code above gets translated to Javascript…

```
"use strict";
var TestClass = /** @class */ (function () {
    function TestClass(name, count) {
        this.name = name;
        this.count = count;
```

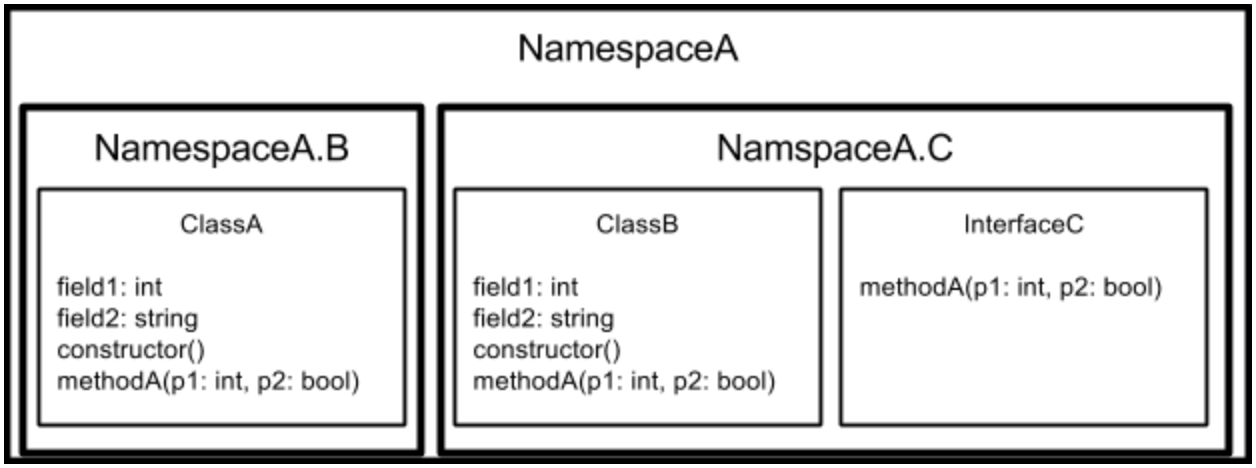
```
    }
    TestClass.prototype.output = function () {
        console.log(this.name + ' ' + this.count);
    };
    return TestClass;
}());
new TestClass('nametest', 99).output();
//# sourceMappingURL=helloWorld.js.map
```

Where you can run it directly from NodeJS…

```
~/test $ node helloWorld.js
nametest 99
```

Organization

Much like Java and C#, the general hierarchy of code written in TypeScript is breakdown by namespace, followed by classes and interfaces, followed by fields and methods within those classes…



> **NOTE**: Modules/namespaces in TypeScript are somewhat equivalent to Java packages. Unlike Java packages, you explicitly have to export functionality from your namespaces/modules.

Types

TypeScript's type system give you following basic types…

- basic primitives: e.g. number, string, boolean
- object types: e.g. functions, classes, interfaces, modules, literals
- union types: e.g. string | boolean | Window

- any type: can be anything
- null type: must be set to null?
- undefined type: must be set to undefined?
- void type: for functions/methods that return nothing

  NOTE: null and undefined may seem useless but see the section on union types -- they're really important if you want nullable types.

Arrays

This is how you specify array types and provide initial values in TypeScript...

```
var a: number[];
var b: number[] = [1, 2, 3];
```

You can access arrays just like you would in Javascript/Java. In addition, arrays provide several functions/methods that you can call to do common things such as sort/slice/reverse/etc…

```
console.log(b);
console.log(b[2]);
console.log(b.slice(0,2));
```

```
~/test $ node helloWorld.js
[ 1, 2, 3 ]
3
[ 1, 2 ]
```

NOTE: Another way of declaring an array is via the generics syntax. For example, if I wanted to declare an array of numbers, I'd do Array<number> = [0,1,2].

Tuples

TypeScript has built-in support for tuples. For example…

```
var x: [number, string];
x = [1, 'a'];
console.log(x);
console.log(x[0]);
console.log(x[1]);
```

Enums

TypeScript has built-in support for enums. Internally, enums are treated as numbers (just like in C/C++). For example…

```
enum Category { A = 9, B, C };
var x: Category = Category.B;
console.log(x);
console.log(Category.B);
```

```
~/test $ node helloWorld.js
10
10
```

String Literal

String literal types are types that can only be assigned to the string literal defined by the type. For example…

```
var x: 'Test';
x = 'Test';
```

In the above example, you can't set the variable x to anything other than the string Test. This may seem useless until you combine them with union types -- at which point you get something close to an enumeration. For example…

```
var x: 'Started' | 'Stopped' | 'Running';
x = 'Running';
```

Objects

TypeScript has built-in support for a Object type. For example…

```
var x:Object = {
   h: 10,
   w: 20
};
```

The Object type is a generic type. It's very similar to java.lang.Object in that it can point to anything -- functions, classes, interfaces, objects, modules, numbers, strings, etc... For example, the following code is perfectly valid…

```
class c {
   h:number=10;
```

```
   w:number=20;
};
var x:Object = c;
var y:Object = new c();
var z:Object = 'hi';

console.log(x);
console.log(y);
console.log(z);
```

Here's what it outputs…

```
~/test $ node helloWorld.js
[Function: c]
c { h: 10, w: 20 }
hi
```

> **NOTE**: How is this different from the any type? According to https://stackoverflow.com/a/18961904/1196226, Object is more restrictive than any in that it will throw compile-time errors if you try to invoke a method on it. If you use any as the type, it won't throw a compile-time error. So the moral of the story is always use Object over any?

Functions

Since callback functions are used frequently in Javascript, you can create function types in TypeScript. For example…

```
var a = function func1(x:number, y:number): string {
   return String(x + y);
}
var b: (x:number, y:number)=>string;

b=a;

console.log(b(5,5));
```

There'll be more on types in the syntax section, but right now all you need to know is that when you're type is a function, you…

- don't include the function keyword
- don't include the function name
- replace the : before the return type with a =>

NOTE: The generic type of a function is Function. Just like how Object can point to pretty much any TypeScript object, Function can point to pretty much any TypeScript function as well as can be invoked as a function. The problem with that is no safety checks happen at design-time if you do invocations on Functions.

Generics

Just like with Java, we can use generics inside of TypeScript. A good example of generics is the Array class: the class which all arrays in typescript derive from. You can see that generic types are declared and used in pretty much the same way…

```
interface Array<T> {
...
   reverse(): T[];
   /**
     * Removes the first element from an array and returns it.
     */
   shift(): T | undefined;
   /**
     * Returns a section of an array.
     * @param start The beginning of the specified portion of the array.
     * @param end The end of the specified portion of the array.
     */
   slice(start?: number, end?: number): T[];
...
}
```

Just like with Java generics, you can pass in multiple type parameters and even have your type parameters be constrained to some higher-level type…

```
function getProperty<T, K extends MyClass>(obj: T, key: K) {
    ...
}
```

Just like with Java generics, you can declare the parameter type directly on the method/function instead of on a class/interface…

```
function func<T extends number>(val: T): number {
   return val + 10;
}
```

Unions

Union types are types that can be one of many types. For example…

```
var x: number | string;

x = 'aaa';
x = 4;
```

In many cases, you won't be able to set a variable to null/undefined (even if the type is non-primitive). The whole point of having null and undefined types is that you can use union types to have nullable/undefinable variables. For example…

```
var x: number | null;
x = null;
```

Intersections

Intersection types are types where the value must be of all the declared types. For example...

```
var x: Serializable & Loggable;
```

Intersection types must include all the members from the types being intersected. Remember that TypeScript doesn't have runtime type information, so as long as all the expected members are there, this will work. That means that what you're assigning to the example variable above doesn't have to be a class instance that implements Serializable and Loggable, it just has to have all the same members as those interfaces.

> **NOTE**: This seems to be an niche feature. The documentation indicates that this feature is useful for mixins, which is some Javascript way of combining functionality from many classes together.

Aliases

Type aliases are the same thing as typedefs. To declare a typedef / type alias, use the type keyword. For example…

```
type nullableString = string | null;

var x: nullableString = 'mystr';
x = null;
```

Syntax

TypeScript's syntax is very similar to most other object oriented languages (e.g. Java or C#).

The following is a brief overview of the TypeScript keywords and what they're for. A lot of this should be self explanatory. The subsections below will cover these in a bit more detail.

- curly brackets ({ }) for wrapping blocks of code
- semicolons (;) for ending statements
- while/for/do-while for loops
- if/else/switch for conditionals
- class/interface for defining classes or interfaces
- construct for defining class constructor methods
- public/protected/private for defining visibility of class properties
- extends for inheriting from another class
- implements for implementing an interface
- import to import some functionality
- ... for specifying varargs (called rest parameters in TypeScript)
- module/namespace for defining packages
- => { … } for lambdas
- <typename> for casting a variable
- name: type for defining class fields and function parameters
- const for declaring a constant variable
- readonly for declaring a final variable
- export for making things visible from a module

Variables

Declaring variables inside of TypeScript is similar to Javascript. The main difference is that you can pass in a type when you declare a variable…

```
var a: number = 2;
```

We can also leave out the type, but when we do that it will implicitly type that variable to whatever we shoved into it...

```
var a = 2; // is automatically typed as number
var c = a + 2; // is automatically typed as number
```

If we don't initialize the variable and also leave out the type, the variable will automatically get typed as any.

> **NOTE**: In addition to var, typescript also supports let from the newer ECMA standards. Using let will give you variable scoping rules similar to Java/C#, so you might want to switch to using let instead of var.

Constants

Declare constants inside of TypeScript similarly to how you declare variables, but use the const keyword instead…

```
const a = 2;
const b:number = Math.random();
var c:number = a + b;
```

> **NOTE**: A constant and a final/readonly field are different things.

Casting

You can cast one type to another using the greater than / less than braces (< >). For example…

```
class Type1 { }
class Type2 extends Type1 {
   x: number = 5;
}

var inst: Type1 = new Type2();
const val = (<Type2> inst).x;
```

Just like with Java, the TypeScript casting system will stop you from casting to things that aren't possible. For example, if you tried to cast a number to a string you would get back a compile-time error.

> **NOTE**: To convert a number to a string, use number.toString(). For the opposite, use parseInt() or parseFloat().

Instanceof

Type guards are pretty much the same thing as instanceof in Java…

```
class C {}
class P {}

let a: C | P;
a = new C();

console.log(a instanceof P);
```

NOTE: These only work on classes because classes have constructors and that's whati t looks for. They don't work on interfaces. Interfaces don't have constructors or any kidn of runtime type information stored. For interfaces, you have to use user-defined type guards where you specifically have to have a function do the checks. See the bottom of https://basarat.gitbooks.io/typescript/docs/types/typeGuard.html for an example.

One feature specific to TypeScript is that if you're using instanceof in a if/else (or other kind of branching operation?), it'll automatically cast if the branch you goto is for the instanceof passing. For example…

```
class C {}
class P {
   public pField: number = 10;
}

let a: C | P;
a = new P();

if (a instanceof P) {
   console.log(a.pField);
}
```

Notice how in the example above, there was no explicit cast to P in the if statement. The compiler automatically figures it out.

Functions

Functions can be declared as such…

```
function func1(x:number, y:number): string {
   return String(x + y);
};
```

The important things to note here are that…

- the parameters are typed -- if they aren't they'll be inferred (most likely as any)
- there is a return type defined -- if it isn't there it'll be inferred
- you must pass args for all the parameters unless otherwise specified (discussed further in subsections below)

You can also assign a anonymous function to a variable, just like you would in normal Javascript…

```
var func1 = function(x:number, y:number) {
   return String(x + y);
```

```
};
```

Optional Parameters

Unlike Javascript, In TypeScript all the parameters for a function are required. However, you can explicitly set parameters to be optional by adding a ? directly after the name. For example…

```
var func1 = function(x:number, y?:number) {
   if (typeof(y) === 'number') {
       return String(x + y);
   } else {
       throw "error";
   }
};
```

It looks like adding a ? is just shorthand for making the type a union type that includes undefined. So the function signature in the would be equivalent to…

```
var func1 = function(x:number, y:number|undefined) {
    ...
}
```

> **NOTE**: The question mark syntax is specific to functions -- you cannot use them outside of parameters? For example, var x?:number will fail.

Default Parameters

Default parameters are like optional parameters, but if the value passed into the parameter is undefined, it'll use the default type. For example, the following code will print out 21 on each console.log()…

```
var func1 = function(x:number, y:number=11) {
   return String(x + y);
};

console.log(func1(10, 11));
console.log(func1(10));
console.log(func1(10, undefined));
```

Rest Parameters

Rest parameters are TypeScript's term for varargs. They're similar to Java's varargs in that it's captured as an array, but…

- you need to explicitly declare that parameter as an array
- the … goes before the name of the parameter

For example...

```
var func1 = function(name: string, ...values: number[]) {
   console.log(name + values.join(' '))
}

func1("Test", 1, 2, 3, 4, 5, 6)
```

Overloads

Overloads in TypeScript are a bit different than other languages. To overload a method, you can provide multiple signatures but only 1 implementation. So long as the signatures can shift their parameter/return types around to the point where they can be fed into your implementation, everything will work fine. The following example is from https://stackoverflow.com/a/13212871/1196226…

```
function myMethod(a: string): string;
function myMethod(a: number): string;
function myMethod(a: number, b: string): string;
function myMethod(a: string|number, b?: string): string {
   return a.toString();
}
```

> **NOTE**: If you're using TypeScript compiler's strict mode, you're going to have a tough time with this. It'll error out if any of your parameter types or return type resolve to any.

Another way of faking an overload is to create a single implementation, but use union types / optional types for the parameter types and return types. Here's the same example as above but without all the different signatures…

```
function myMethod(a: string|number, b?: string) {
   return a.toString();
}
```

Lambdas

You can define functions using the following notation…

```
var a = (x: number, y: number): number => x * y;
```

Compare this to the normal way functions are defined…

```
var b = function(x: number, y: number): number {
   return x * y;
}
```

Note the differences…

- function keyword missing in lambda
- return keyword missing in lambda

Why is this useful? Think of doing things to streams such as mapping or filtering…

```
let arr:number[] = [1,2,3,4,5,6,7,8,9];
arr = arr
       .filter(val => val % 2)
       .map(val => val * 10);
```

In the above example we omitted the parameter types and return type because they were inferred, but we could have just as easily added them in. It would just make things slightly less readable…

```
let arr:number[] = [1,2,3,4,5,6,7,8,9];
arr = arr
       .filter((val: number): number => val % 2)
       .map((val: number): number => val * 10);
```

The rules here are similar to lambdas in Java. For defining parameters, if you're running a function that...

- takes in 0 params: () => false
    - you MUST have an empty set of parentheses no signify no parameters.
- takes in 1 param: x => x*x
    - you MAY wrap the parameter in parenthesis if you want, or leave parenthesis out.
- takes in >=2 params: (x, y) => x*x
    - you MUST wrap the parameters in parenthesis.
- runs a single statement: x => x*x
    - you MAY omit the curly braces around the body ({ }) -- if you choose to omit the braces, you MUST omit the return keyword as well.
- runs multiple statement: x => { x = x/2; return x*x; }
    - you MUST wrap curly braces around the body ({ }) and MUST provide a return statement.

Remember that this same arrow syntax is used to declare types that hold a function. For example…

```
var a: (x:number) => number = (x: number): number => x * y
```

Note that we when use this syntax to declare the type, we place the return type of the function after the =>, not the computation. The example above declares variable a that accepts a

function with a certain signature, and assigns a function to it with that signature (declared as a lambda).

Interfaces

Just like Java, your code can have interfaces.

Just like Java, an interface…

- can extend multiple other interfaces
- can make use of generic types.
- defaults visibility of interface members to public.

Unlike Java, interfaces in TypeScript...

- can have fields as well as function definitions.
- can't have a visibility scope on the interface itself (e.g. private, public, etc…)
- don't have runtime type information (e.g. can't do instanceof checks on interfaces, but can on classes)

  NOTE: When defining a function in an interface, remove the function keyword.

  NOTE: In Java, you can only implement an interface. In TypeScript, you can implement a class if you want to. If you do, it'll treat the class like an interface, meaning you have to re-implement whatever functions/fields it defines.

Here's an example of a interface in TypeScript….

```
interface p {
   aaaa: string;
}

interface d {
   bbbb: string;
}

interface test<T> extends p, d {
   val?: T;
   func1: (msg: string) => string;
   func2: (msg: string) => number;
}
```

Optional Fields

Just like with functions, your fields can be optional (can be of the type you specified or can be undefined). To make your field optional, simply add a question mark (?) after its name…

```
interface test<T> extends p, d {
   val?: T;
   func1: (msg: string) => string;
   func2: (msg: string) => number;
}
```

Readonly Fields

Readonly fields are like Java final fields. They're fields that can be set during class construction and can't be set again after that. To mark a field as readonly, add the readonly keyword before the field name...

```
interface test<T> extends p, d {
   readonly val?: T;
   func1: (msg: string) => string;
   func2: (msg: string) => number;
}
```

Classes

Just like Java, your code can have classes.

Just like Java, a class…

- can only extend a single class (no multiple inheritance).
- can be abstract.
- can implement multiple interfaces.
- can have a constructor.
- can have final fields (called readonly in TypeScript).
- can have static fields.
- can have static functions/methods.

Unlike Java, a TypeScript class…

- must always use this keyword to access members.
- defaults visibility of class members to public (package-private by default in Java).
- can't be marked as final/sealed.
- can't have multiple constructors (use factory methods as a workaround).
- can't have a visibility scope on the class itself (e.g. private, public, etc…).
- can't have volatile fields (doesn't make sense since Javascript is single threaded).
- can automatically turn constructor parameters into fields.

  NOTE: Although a class can't be sealed, you can technically seal it by giving it a private constructor and having using a factory method to create instances. It looks like the TypeScript guys don't want to support this feature.

NOTE: In Java, you can only implement an interface. In TypeScript, you can implement a class if you want to. If you do, it'll treat the class like an interface, meaning you have to re-implement whatever functions/fields it defines.

NOTE: When defining a function in a class, remove the function keyword.

Here's an example of a class in TypeScript that inherits from another class as well as implements an interface….

```
class c {
    val: string = 'hi';
}

interface d {
    myNumStr?: string;
}

class e extends c implements d {
    myNumStr?: string;
    constructor(public myNum: number) {
        super();
        this.myNumStr = myNum.toString();
    }
}

console.log(new e(5));
```

Constructor Fields

In TypeScript, you can automatically have a constructor parameter be a field on the class as well. You do this by setting a visibility scope with the parameter. For example…

```
class MyClass {
    constructor(public myNum: number) {
    }
}
```

In the above example, myNum will become a field on MyClass that has public visibility. You can also set it to have protected visibility or private visibility.

Optional Fields

Just like with functions, your fields can be optional (can be of the type you specified or can be undefined). To make your field optional, simply add a question mark (?) after its name…

```
interface test<T> extends p, d {
```

```
    val?: T;
    func1: (msg: string) => string;
    func2: (msg: string) => number;
}
```

Readonly Fields

Readonly fields are like Java final fields. They're fields that can be set during class construction and can't be set again after that. To mark a field as readonly, add the readonly keyword before the field name...

```
interface test<T> extends p, d {
    readonly val?: T;
    func1: (msg: string) => string;
    func2: (msg: string) => number;
}
```

Accessor Functions (Getters and Setters)

Instead of providing direct access to a field, you can provide getters and setters like you do in Java. The syntax for setters and getters in TypeScript is a bit different. Here's an example…

```
class MyClass {
    private _myNum: number
    constructor(myNum: number) {
        this._myNum = myNum;
    }

    get myNum(): number {
        return this._myNum;
    }

    set myNum(x: number) {
        this._myNum = x;
    }
}
```

The long and short of it is that the …

- getter and setter function names should be the same (name of the property).
- getter function must be prefixed with get.
- setter function must be prefixed with set.

In the above example, the getters and setters expose a myNum property that directly manipulate the _myNum field.

Constructor Functions

Constructors in TypeScript classes require you to use the constructor keyword. You can only have 1 constructor in a class. If you want more than 1 constructor, your workaround is to provide static factory functions/methods.

Here's an example of a constructor…

```
class MyClass2 {
   protected myInt: number;
   constructor(x: number) {
       this.myInt = x;
   }
}
```

One feature of TypeScript constructors is that you can force a parameter to automatically map to a field. You do this my specifying a visibility modifier directly in the parameter. The following example is equal to the example above…

```
class MyClass2 {
   constructor(protected myInt: number) {
   }
}
```

> **NOTE**: Just like Java, if you're inheriting from another class, you need to call the parent class's constructor using super(). Unlike Java, this isn't done automatically for you in certain cases -- you need to explicitly do it every time.

Static Fields / Functions

Static fields and static functions work just like they do in Java. You define a member as static with the static keyword. Here's an example…

```
class MyClass {
   public static n: number = 5;
   public static func(x:number) : number {
       return 5;
   }
}
```

Just like with Java, you can manipulate protected/private instance fields from within a static function/method. Here's an example...

```
class MyClass {
    private n: number = 5;
```

```
    public static func(x:MyClass, newN: number) : void {
        x.n = newN;
    }
}
```

Namespaces

> **NOTE**: Don't use this. Use modules instead. See the Modules section.

Namespaces are kind of like packages in Java, but not really. They organize code chunks by logically sticking them together. But, unlike Java, chunks of code under the same namespace/package aren't visible to each other unless they're explicitly exported.

That means that you can have 2 classes in different files but under the same namespace and they won't know about each other unless they're exported. When you export them, they'll know about each other, but everyone else will know about them as well. As such, there is no such thing as package-private scope in TypeScript.

Your stuff is either visible to all namespaces or visible to no namespaces.

> **NOTE**: There currently doesn't seem to be a way to forcibly export everything in a namespace.

> **NOTE**: Namespaces were called modules in older versions of TypeScript.

Just like Java, namespaces...

- are hierarchical.
- are separated by dots.

Unlike Java, namespaces…

- don't have an equivalent to package-private visibility.
- aren't bound to directory names.
- can be nested within each other in a single file.
- can be split up within a single file or across multiple files.
- must explicitly choose what is visible via exporting.

To declare a namespace, use the namespace keyword followed by a name and curly braces ({ }). Everything within the curly braces will belong to that namespace. For example…

```
namespace a {
   var x = 10;
   namespace b {
       var y = x;
```

```
        class C {
            f: number = 1;
        }
    }
}
```

> **NOTE**: Code that isn't explicitly under a namespace is put under the global namespace. I'm guessing this is equivalent to adding your classes under the default package in Java.

Exporting

To export some part of a namespace, you need to explicitly export it using the export keyword. If you don't export it, it won't be visible (not even to itself -- see Splitting section). For example...

```
namespace a.b {
    export var z = 10;
}
```

Importing

Once exported, you can use it directly but it's much more common to use the import keyword to get shorthand access (very similar to Java import).

For example…

```
import z = a.b.z;
console.log(z);
```

If the file your importing/using in is different from your the file you're exporting from, you'll need to add a reference to the typescript file that you're exporting from. Otherwise, the TypeScript compiler won't know where to look to get definitions for whatever it is you're importing. For example...

```
/// <reference path="a.b.ts" />

import z = a.b.z;
console.log(z);
```

Splitting

You can technically split a namespace up across multiple files (or across the same file). For example…

```
namespace a {
    var x = 10;
```

```
    namespace b {
        var y = x;
    }
}

namespace a.b {
    var z = 20;
}
```

The problem with splitting is that you don't get access to stuff from the other splits unless you explicitly export those items as well as the namespace itself. For example, in the last namespace block of the above example, we wouldn't be able to access the x or y vars unless they were explicitly exported.

If we wanted to gain access in the bottom split, we would need to export the relevant bits as such…

```
namespace a {
    var x = 10;
    export namespace b {
        export var y = x;
    }
}

namespace a.b {
    export var z = a.b.y;
}

console.log(a.b.z);
```

Spreading

Spreading spreads the contents of an array out, such that you can feed it into a function that expects parameters or used for prepending/appending to another array.

> **NOTE**: This seem to be a niche feature?

For example…

```
var x: number[] = [1, 2, 3]

// params MUST BE OPTIONAL or spreading for func params will error
function f(a?: number, b?: number, c?:number): void {
    console.log(`${a} ${b} ${c}`);
```

```
}
f(...x);

console.log([9,10,...x]);
console.log([...x,9,10]);
```

```
~/test $ node helloWorld.js
1 2 3
[ 9, 10, 1, 2, 3 ]
[ 1, 2, 3, 9, 10 ]
```

> **NOTE**: If spreading to pass into a function, the arguments taking in the spreaded values must be optional. This is because the array could be shorter than the number of parameters, so any parameters which there aren't values for will be marked as undefined.

Destructuring

Destructing automatically pulls pieces of an array or object into individual variables.

> **NOTE**: This seem to be a niche feature?

In the following example, a will contain the first element and be will contain the second element...

```
var myArray: number[] = [3, 2, 1, 0, -1];
var [a, b] = myArray;

console.log(a);
console.log(b);
```

If you want to dump the remainder into a third variable, you can use rest parameters. In the following example, c will be an array containing the remaining elements…

```
var myArray: number[] = [3, 2, 1, 0, -1];
var [a, b, ...c] = myArray;

console.log(a);
console.log(b);
console.log(c);
```

```
~/test $ node helloWorld.js
```

```
3
2
[ 1, 0, -1 ]
```

> **NOTE**: The rest parameter here doesn't require a type with a [] like it does for functions? This may be a spread operator and not a rest parameter??

You can do the same thing with objects, but your variable name must map to the field name you're pulling out. For example…

```
class C {
   public xField: number = 10;
   public yField: number = 20;
   public zField: number = 30;
}

var c = new C();
var { xField, yField } = c;

console.log(xField);
console.log(yField);
```

The names have to map to the fields in the object, or you can explicitly name the fields you want to place in each variable by using the following syntax…

```
class C {
   public xField: number = 10;
   public yField: number = 20;
   public zField: number = 30;
}

var c = new C();
var { "xField": x, "yField": y } = c;

console.log(x);
console.log(y);
```

Decorators

Decorators are TypeScript's version of Java's annotations. They can be attached to..

- classes
- class methods
- class properties (fields / accessors)
- function parameters

> **NOTE**: To use decorators, you must enable it in the compiler via the experimentalDecorators - add it to tsconfig.json.

Decorators are implemented as plain TypeScript functions. The parameters of those functions differ based on what the decorator is for (e.g. class vs class method), but you almost always get access to the structure of whatever it is you're annotating. You can do all kinds of bizarre things thanks to Javascript's loose nature: extend or remove or change functionality, rename class fields and methods, print messages, etc..

Class

Here's an example of how to declare a decorator for a class and apply it…

```
function fancyDecorator(target: Function) {
   console.log("dec was hit!")
}

@fancyDecorator
class MyClass {
   x: number;
   y: number;
}
```

The target parameter takes in a Function, which is the constructor function of the class being decorated.

If you want your decorator to take in parameters, you have to declare your decorator function as a factory. In the following example, the decorator from above has been modified to take in a single string and output it when its applied…

```
function fancyDecorator(name: string) {
   return function(target: Function) {
       console.log(`dec was hit with name ${name}!`)
   }
}

@fancyDecorator('fancyNameHere')
class MyClass {
   x: number;
   y: number;
}
```

```
~/test $ node helloWorld.js
dec was hit with name fancyNameHere!
```

Methods and Accessors

The way to declare a method decorator is similar to a class decorator, except that the parameters are slightly different…

```
function methodDec(target: any, prop: string, desc: PropertyDescriptor) {
   console.log(`dec was hit with ${prop} and ${desc}`);
}

class MyClass {
   x: number;
   y: number;

   @methodDec
   myMethod(x: string) { }
}
```

```
~/test $ node helloWorld.js
dec was hit with myMethod and [object Object]
```

In the case of method decorators, the target parameter can be either a constructor (if the method is a static method) or the prototype of the class (if the method is an instance method). That's why it's set to take in any type.

If you want the decorator to take in parameters, use a factory method just like with the class decorators…

```
function methodDec(str: string) {
   return function(target: any, prop: string, desc: PropertyDescriptor) {
       console.log(`dec was hit with ${prop} and ${desc} -- ${str}`);
   }
}
class MyClass {
   x: number;
   y: number;
   @methodDec('teeeeeeeeeest')
   myMethod(x: string) { }
}
```

```
~/test $ node helloWorld.js
```

```
dec was hit with myMethod and [object Object] -- teeeeeeeeeest
```

Fields

The way to declare a field decorator is similar to a class decorator, except that the parameters are slightly different…

```
function fieldDec(target: any, propKey: string) {
   console.log(`dec was hit with ${propKey}`);
}
class MyClass {
   @fieldDec
   x: number;
   @fieldDec
   y: number;
}
```

```
~/test $ node helloWorld.js
dec was hit with x
dec was hit with y
```

In the case of field decorators, the target parameter can be either a constructor (if the method is a static method) or the prototype of the class (if the method is an instance method). That's why it's set to take in any type.

If you want the decorator to take in parameters, use a factory method just like with the class decorators…

```
function fieldDec(str: string) {
   return function(target: any, propKey: string) {
       console.log(`dec was hit with ${propKey} -- ${str}`);
   }
}
class MyClass {
   @fieldDec('this is for the x field')
   x: number;
   @fieldDec('this is for the y field')
   y: number;
}
```

```
~/test $ node helloWorld.js
dec was hit with x -- this is for the x field
dec was hit with y -- this is for the y field
```

Modules

Modules are pretty much the same thing as namespaces, but external to your application. For example, JQuery would probably be in its own namespace and be its own module.

The difference is between namespaces and modules is that modules may have other modules as dependencies. Those dependencies may also have dependencies, and so on. With TypeScript modules, all the dependencies in the tree get resolved and loaded up in the correct order.

> **NOTE**: Namespaces were also called modules in older versions of TypeScript. These types of modules were called external modules, while namespaces were called internal modules. It sounds like namespaces have been deprecated in favour of modules.

The way modules get loaded up is dependent on which module system TypeScript's compiler is setup to use. The module system used by most people seems to be commonjs (system used by NodeJS). Here's an example tsconfig.json file…

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

> **NOTE**: If you're compiling for a browser, you want to set the module format to either amd and load it up using Require.js, or system and load it up using SystemJS. If you're compiling for Node, stick with commonjs.

Exporting

You create/export a module just by exporting something at the top level using the export keyword. For example…

```
export namespace a.b {
   export var z = 10;
}
```

You can also have a module that only exports a single unnamed class or interface. This is called a default export and is identified by export default keywords. For example…

```
export default class {
```

```
    field1: string;
    field2: number;
};
```

Importing

You can import a module by using import and require(filename), where the file you're requiring exports something at the top-level (see above section). For example…

```
import myModule = require("./a.b");
console.log(myModule.a.b.z);
```

> **NOTE**: Notice how /// <reference path="a.b.ts" /> isn't needed here. It seems to pick out the type information on its own? Also, notice how just like how you use require() just like in NodeJS -- the file extension (.ts in this case) has been removed.

You can also import using the following syntax, which will import specific items from a module...

```
import { a } from "./a.b";
console.log(a.b.z);
```

Or, this syntax, which will import everything from a module…

```
import * as myModule from "./a.b";
console.log(myModule.a.b.z);
```

> **NOTE**: The "import * as" syntax in the example above won't work for default exports. If you use it, you need to use myModule.default to access the interface/class being exported.

Interfacing with Javascript

In certain cases, you may be …

- pulling in raw Javascript instead of a TypeScript module.
- your code is for a platform that gives you access to a bunch of global variables.

Ambient Declarations/Definitions

The problem with this is that the TypeScript compiler doesn't know about things that were never declared. For example, if your code is targeting a browser, you'll most likely be pulling in JQuery. If you try to use JQuery's $ variable outright, the TypeScript compiler will complain because it won't know that $ was declared globally by JQuery.

You can use ambient declarations to work around this issue. To setup a variable as an ambient declaration, use the declare keyword. For example…

```
declare var $: any;

var x:string = $(".div").text();
```

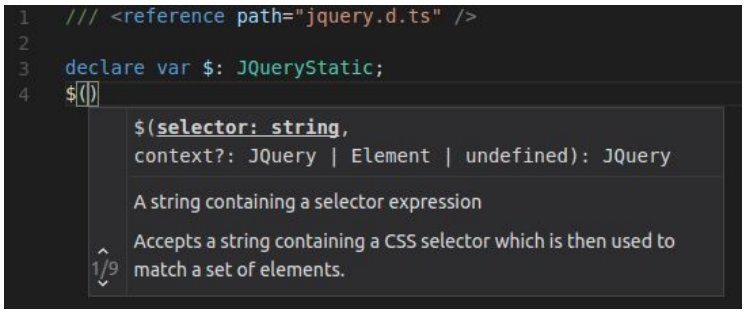
Ambient declarations tell the TypeScript compiler that the variable has been declared somewhere outside of TypeScript and prevents the TypeScript compiler from doing any checks against it.

> **NOTE**: There's also a feature called ambient definitions which will wrap things as a module? The syntax seems to be declare module modulename { ... }, where … contains the a list of things that get exported using the export keyword. It seems to be how most type definitions are provided now?

Type Definitions

In addition to ambient declarations, you can also provide a type definition file. Type definition files define all the function signatures/variables/types/etc.. that TypeScript needs to make sure your usage of the ambient declaration is correct + allows your IDE to provide functionality like Intellisense. For example…

```
/// <reference path="jquery.d.ts" />
declare var $: JQueryStatic;
$('div').text('HI!');
```



> **NOTE**: Type definition files always seem to have a .d.ts extension.

It looks like most type definitions are now provided as ambient definitions (see note in previous section), so after providing the triple dash reference path, you would pull them in as if you're importing any other module.

To automatically install type definitions for a package, you can use npm install

@types/namehere. For example…

```
npm install @types/lodash
+ @types/lodash@4.14.88
added 1 package in 1.7s
```

See http://www.typescriptlang.org/docs/handbook/declaration-files/consumption.html for more information.

> **NOTE**: The repo for type definitions is at https://github.com/DefinitelyTyped/DefinitelyTyped. There was a command line tool that you could use to automatically manage type definition files from this repo called tsd, but that was deprecated in favour of another tool called typings. Typings also ended up being deprecated in favour of using the npm directly (method shown above).

TSLint

TSLint is a TypeScript linting tool, similar to checkstyle/findbugs/PMD for Java. The following is a guide that shows how to install and run tslint, as well as integrate it with VSCode.

Use npm to install TSLint…

```
~/test $ sudo npm install tslint typescript -g
[sudo] password for user:
/usr/bin/tslint -> /usr/lib/node_modules/tslint/bin/tslint
/usr/bin/tsc -> /usr/lib/node_modules/typescript/bin/tsc
/usr/bin/tsserver -> /usr/lib/node_modules/typescript/bin/tsserver
+ tslint@5.8.0
+ typescript@2.6.2
updated 2 packages in 4.076s
```

> **NOTE**: You can also install locally instead of globally. Take out the -g and add a --save-dev to have it saved as a devDependency in your package.json.

Once installed, you need to create a new tslint.json configuration using the tslint --init. The command won't output anything to the shell, but you will get a new tslint.json file in your current directory…

```
{
   "defaultSeverity": "error",
   "extends": [
       "tslint:recommended"
   ],
```

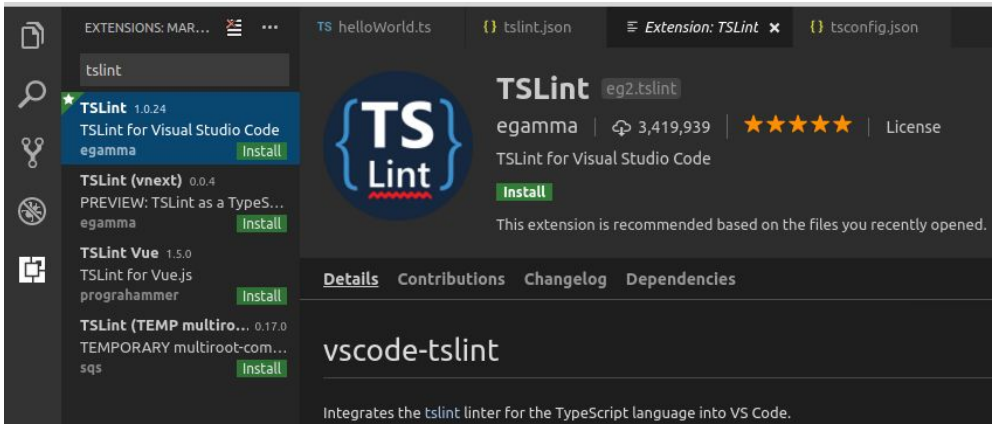
```
    "jsRules": {},
    "rules": {},
    "rulesDirectory": []
}
```

To lint your TypeScript files via command-line, you can run tslint globpattern. For example…

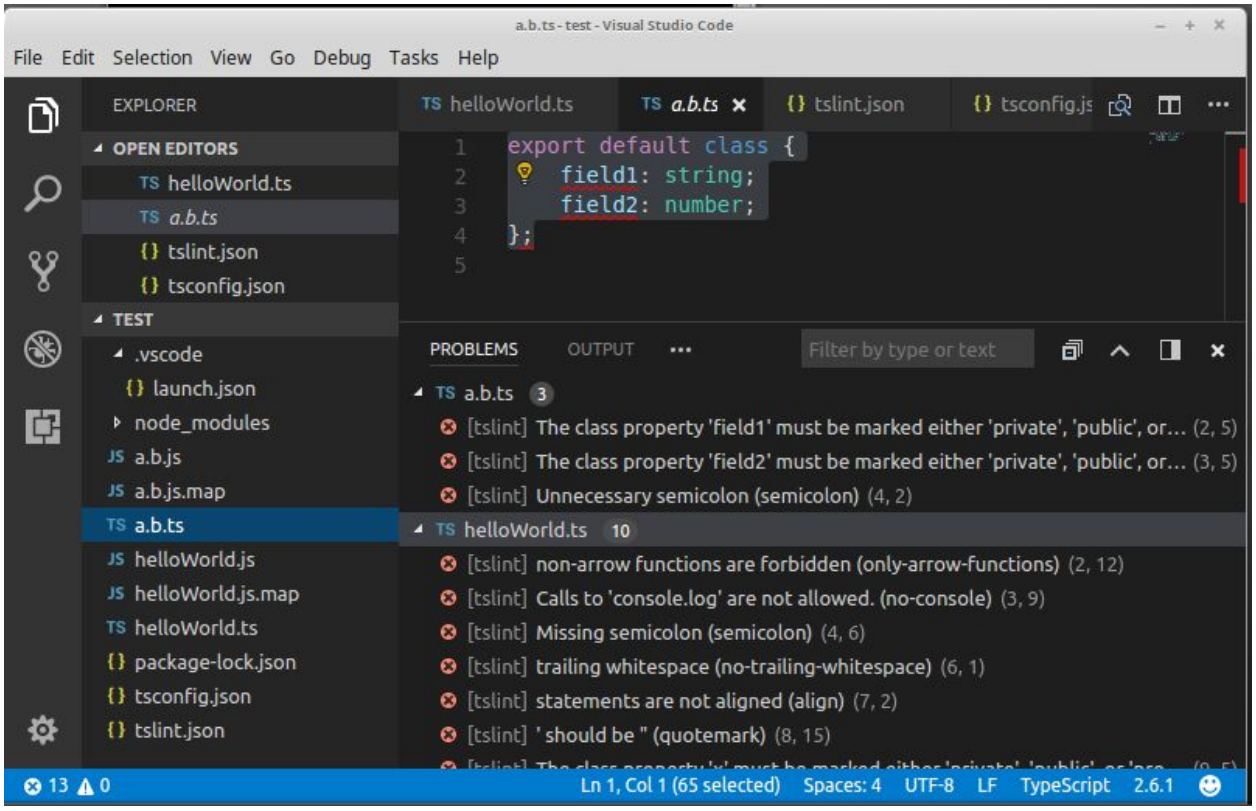
```
~/test $ tslint *.ts

ERROR: a.b.ts[2, 5]: The class property 'field1' must be marked either
'private', 'public', or 'protected'
ERROR: a.b.ts[3, 5]: The class property 'field2' must be marked either
'private', 'public', or 'protected'
ERROR: a.b.ts[4, 2]: Unnecessary semicolon
ERROR: helloWorld.ts[2, 12]: non-arrow functions are forbidden
ERROR: helloWorld.ts[3, 9]: Calls to 'console.log' are not allowed.
ERROR: helloWorld.ts[4, 6]: Missing semicolon
ERROR: helloWorld.ts[6, 1]: trailing whitespace
ERROR: helloWorld.ts[7, 2]: statements are not aligned
ERROR: helloWorld.ts[8, 15]: ' should be "
ERROR: helloWorld.ts[9, 5]: The class property 'x' must be marked either
'private', 'public', or 'protected'
ERROR: helloWorld.ts[10, 15]: ' should be "
ERROR: helloWorld.ts[11, 5]: The class property 'y' must be marked either
'private', 'public', or 'protected'
ERROR: helloWorld.ts[12, 3]: file should end with a newline
```

A better way to lint your files is directly in your IDE. Most IDEs come with TSLint support, either built-in or via extensions. If you're using VSCode, there's a tslint extension that you can install and use…



Once you install it, if you go to the problems tab on the bottom pane of VSCode you'll see all the tslint errors for your open files.



For each error, if you move the carrot to where the error is a little light-bulb will show up. You can click that lightbulb and it'll give you ways to fix the problem identified by the linter…

This list will refresher as you make changes and save.