## Types

**Transpile-time type checking**

Some of the most common mistakes we make while writing JavaScript code is to misspell a property or a method name.

We only find out about the mistake when we get a *runtime error*, that is to say when we are running our application.

 This is why testing is so important for javascript applications, only can surface errors. *running* them

Other languages like C++ or Java have something called *type checking*, during a compilation phase it first checks for some simple mistakes in the code and raises an error if it finds any.

TypeScripts transpilation mechanism also performs *type checking*, however it only works when we tell TypeScript the *type* of things.

e.g. in javascript we might write:

let a;

a = 1;

a = '2';

We expected the type of the a variable to remain a number over its life. But by mistake we assigned a string '2' to a, this doesn’t cause an error in JavaScript.

Only if one of the functions which depends on a starts misbehaving do we even get a clue that something might be wrong.

However with typescript we can clearly state that we want the a variable to always hold a number, like so:

let a: number = 1;

Then if somewhere else in our code we wrote:

a = "1";

We get this error:

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

If we take advantage of types in typescript the transpiler will warn us with a transpile-time errors.

**Supported Types**

### Basic Types

We can support boolean, number and string.

let decimal: number = 6;

let done: boolean = false;

let color: string = "blue";

### Arrays

We have two ways of describing the types of arrays.

The first is to use the brackets notation [], like so:

let list: number[] = [1, 2, 3];

The above indicates to TypeScript that this array should only hold numbers.

The second way uses a generic type specifically Array<Type>, like so:

let list: Array<number> = [1, 2, 3];



We cover

*generics*

in more detail later on in this lecture.

### Functions

We can describe a variable as one that will only point to a function, like so:

let fun: Function = () => console.log("Hello");

With TypeScript we can also define the *expected return type* of a function, like so:

function returnNumber(): number {

return 1;

}

The above lets TypeScript know that the function should only return a number.

### Enum

An Enum is a datatype consisting of a set of named values. The names are usually identifiers that behave as constants. Enums were introduced in ES6.

enum Direction {

Up,

Down,

Left,

Right

}

let go: Direction;

go = Direction.Up;

### Class & Interface

Classes and Interfaces, whether created by you or others, are also types of things. So we can use them where we would use any of the built-in types:

class Person {};

let person: Person;

let people: Person[];

### Any

If we don’t know the type of something we can fall back to using any.

If used it doesn’t provide any type checking but does make it clear that we intentionally *don’t know* the type, rather than we forgot to add the type.

let notsure: any = 1; notsure = "hello"; // This is fine since we don't do type checking with any

### Void

void means no type, it’s typically used to indicate that a function will not return anything at all, like so:

function returnNothing(): void {

console.log("Moo");

}

### Type Assertion

Sometimes we end up in a situation where we know more than TypeScript about the type of something. In those situations we can use *type assertions* as a hint to the transpiler about the type of something, like so:

let value: any = "Asim Hussain";

let length: number = (<string>value).length;

(<string>value) lets TypeScript know that we believe the type of value to be string.

**Generics**

How can we create re-usable bits of code which can still take advantage of typescripts transpiletime type checking?

Take for instance this example:

class Audio {}

class Video {}

class Link {}

class Text {}

class Post {

content: any;

}

We have a class called Post which has a content property. The content might be an instance of Audio, Video, Link or Text classes.

We could ignore the type of content and just use any like the example above but this doesn’t give us the benefit of type checking.

We can create seperate AudioPost, VideoPost, LinkPost and TextPost types, like so:

class AudioPost {

content: Audio;

}

class VideoPost {

content: Video;

}

class LinkPost {

content: Link;

}

class TextPost {

content: Text;

}

But apart from being verbose and time-consuming it assumes we know all the content types upfront. Maybe a consumer later on would like to create a VRPost type.

With *generics* we can dynamically generate a new type by passing into the Post type a *type variable*, like so:

class Audio {}

class Video {}

class Link {}

class Text {}

class Post<T> {

content: any;

}

<T> above is the *generic* syntax and T is the *type variable*. We could name it anything but the convention if there is only one is to call it just T.

Then we can use T wherever we would use a type, like so:

class Post<T> {

content: T;

}

Finally when we want to create a specific type of Post we can declare it like so:

let videoPost: Post<Video>;

**Optional Types**

By default we don’t *have* to add types when using TypeScript, we could just leave them out.

let answer;

answer = 42;

TypeScript won’t perform any type checking for the above and assumes a type of any.

This behaviour is controlled by the noImplicitAny flag in tsconfig.json. If set to false then TypeScript assumes any for missing types, of set to true then TypeScript throws an error if no type is present.



Opinion is split as to whether to keep this to true or false, by default it’s set to false which is more forgiving but then doesn’t force developers to use types which is the point of TypeScript.

**Type Inference.**

If a variable is *declared and initialised on one line* TypeScript will try to infer, *guess*, the type of a variable by how it’s declared, for example:

let answer = 42;

answer = "42";

TypeScript inferred that the type of answer was number by the fact we initialised it with a number. When we later on try to assign a string to answer it throws an error.

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

**Types for external libraries**

This is all fine for code that’s written in TypeScript and has types.

What if you wanted to use code that wasn’t written in TypeScript or which you are not going to include as TypeScript and compile yourself.

We can use something called an *ambient type definition*.

This is a file that contains meta-data about the types in another library, a *meta-type* file.

This repository (<https://github.com/DefinitelyTyped/DefinitelyTyped>) contains type definitions for some of the most popular 3rd party javascript libraries.

To use them in a file we simply:-

1. Download the associated type file, for example jquery.d.ts.
2. In the typescript file where we want to *use* the 3rd party library, add this to the top of the file:

/// <reference path="jquery.d.ts" />

### typings

Thats a bit cumbersome so there is also a command line tool you can use called typings which handles the leg-work for you.

We install it via npm, like so:

npm install -g typings

In our project folder we then initialise a config file by typing typings init this creates a typings.json file.

Then we can install type definition files like so:

typings install jquery --save --source dt --global

This downloads and installs the type definition files to the ./typings folder and conveniently bundles all of the downloaded files into a single file called index.d.ts.

So then to use jQuery with typescript type support we just need to add a reference to index.d.ts to the top the file where we use jQuery.

/// <reference path="./typings/index.d.ts"/>

**Summary**

With transpile-time type checking TypeScript can help uncover bugs much earlier and faster than if they were to manifest at run-time.

Using types is optional but highly recommended by the Angular team.

If using 3rd party libraries that have already been transpiled into javascript, typescript can still perform transpile-time type checking if we include the type definition file for the library.

**Listing**

[https://github.com/codecraftpro/angular2-sample-code/blob/master/2.es6typescript/13.types/script.ts](https://github.com/codecraftpro/angular2-sample-code/blob/master/2.es6-typescript/13.types/script.ts)

*script.js*

"use strict";

// Core

let decimal: number = 6;

let done: boolean = false;

let color: string = "blue";

let list: number[] = [1, 2, 3];

let list2: Array<number> = [1, 2, 3];

// Function

let fun: Function = () => console.log("Hello");

function returnNumber(): number {

return 1;

}

// Void

function returnNothing(): void {

console.log("Moo");

}

// Enum

enum Direction {

Up,

Down,

Left,

Right

}

let go: Direction;

go = Direction.Up;

// Class

class Person {

}

let person: Person;

let people: Person[];

// Any

let notsure: any = 1;

notsure = "hello"; // This is fine since we don't do type checking with any

// Type Assertion

let value: any = "Asim Hussain";

let length: number = (<string>value).length;

// Generics

class Audio {

}

class Video {

}

class Post<T> {

content: T;

}

let audioPost: Post<Audio>;

let videoPost: Post<Video>;