TypeScript

Introduction

THIS DOCUMENT COVERS

- Introduction

Characteristics and Benefits

Scalable Applications

Cheat Sheets

Questions – The Type System

Overview

Basics of Type

TypeScript allows supports the use of type annotations to add explicit static type information to variables, formal parameters and return types. The compiler then uses this information to perform static type checking.

```
function add(x: number, y: number): number {
    return x+y;
}
let a: number = 5;

add(a,"5");

>> error TS2345: Argument of type '"5"' is not assignable to parameter of
type 'number'.
```

The compiler also uses type inference to implicitly statically type return types and variables where appropriate. In this fragment the return type of `add` and the variable `c` both have a static type of number.

```
function add2(x: number, y: number) {
    return x+y;
}
let c = add(4,5);
```

ANY

To allow the developer to use the full scope of JavaScript's dynamic type system, TypeScript allows any type. When parameters, return types or variables are marked with any they can be assigned values of any type. It then becomes the developer's responsibility to ensure only sensible values are assigned.

```
let y: any = 1;
let x:string = "hello";
x = y;
console.log(x.length);

>> undefined
```

When performing type inference to implicitly statically type variables the compiler will, by default, silently assign the type any when it cannot infer a more explicit type. It is often useful to turn this off using the following compiler option.

```
{
    "compilerOptions": {
        "target": "ES2018",
        "outDir": "./dist",
        "rootDir": "./src",
        "noEmitOnError": true,
        "noImplicitAny": true
    }
}
```

UNION

TypeScript supports union types. These define a restricted set of types that a variable can take.

```
function selector(flag: boolean) : number | string {
    return flag ? 1.0 : "1.0"
}

let selected: number|string  = selector(true);
console.log(selected.toString());
```

If a variable has union type the compiler will only allow operations that are supported by every type in the union. In our case above this pretty much limits us to `toString` and `valueOf`.

TYPE ASSERTION

Where we know more than the compiler, we can narrow a union type value to one the types in the union using a type assertion. The compiler will make sure we only narrow to a type which is a member of the relevant union.

```
let n = selector(true) as number;
let str = selector(false) as string;
console.log(str.length);
```

If the developer gets it wrong the code will fail at runtime.

```
function selector(flag: boolean) : number | string {
    return flag ? 1.0 : "1.0"
}

let selected: number|string  = selector(false) as number;
console.log(selected.toFixed(2));

>> C: …\dist\hello.js:5
>> console.log(selected.toFixed(2));
>> TypeError: selected.toFixed is not a function
```

We can also circumvent the compiler as the following piece of horrible code shows.

> It is worth noting that type assertions are a compile time construct. They do not cause runtime type coercion to occur when the actual runtime type does not match the asserted type.

```
function selector(flag: boolean) : boolean | string {
    return flag ? true : "1.0"
}

let x: number|string  = selector(true) as any as number;
console.log(x + 1);

>> 2
```

TYPE GUARDS

The following structure can be used with primitive types. The compiler is clever enough to allow any string operations inside the if block.

```
let s: number|string  = selector(false);

if (typeof s === "string")
    console.log(s.length);
```

UNKNOWN VERSUS ANY

We can do anything with variables of type any. Using unknown is more restrictive. We can only access unknown inside a guard which is safer.

```
function selector(flag: boolean) : number | string {
    return flag ? 1.0 : "1.0"
}

let an: any  = selector(false);
let un: unknown  = selector(false);

// We dont need to check the type of
// any. We can do anything with it
console.log(an.length);

// Unknown can only be accessed inside a guard
if (typeof un === "string")
    console.log(un.length);
```

NULL AND UNDEFINED

Null and undefined are valid values for all types.

```
var undef: number;
console.log(typeof undef);

undef = null;
console.log(typeof undef);

>> undefined
>> object
```

This can cause problems as these values change the type thereby breaking the static type checking

```
function f() : number {
    return null;
}

var x: number = f();
console.log(typeof x);

>> object
```

We can instruct the compiler to prevent the assignment of null and undefined to other types.

```
{
    "compilerOptions": {
        "target": "ES2018",
        "outDir": "./dist",
        "rootDir": "./src",
        "strictNullChecks": true
    }
}
```

Where we really need to support null, we can add it to the union

```
function f() : number | null {
    return null;
}
```

We then need to be able to do something like this. If we know the value cannot be null we can use a non-null assertion using the ! operator.

```
function f(isNull: boolean) : number | null {
    return isNull? null : 100.0;
}

var x: number = f(false)!;
console.log(typeof x);
```

Another alternative is to use a non-null type guard

```
function f(isNull: boolean) : number | null {
    return isNull? null : 100.0;
}

var x = f(false);

// Compiler error
//console.log(x.toFixed());

if (x !== null)
    console.log(x.toFixed());
```

Functions

There is no function overloading in java. If we create a second function with the same name and different parameters, it overrides the first function. TypeScript will give an error in such cases. Unlike JavaScript, TypeScript expects the number of arguments to match the number of formal parameters. Furthermore, we can instruct the compiler to warn us when a function does not use any of its parameters.

```
  {
      "compilerOptions": {
          "target": "ES2018",
          "outDir": "./dist",
          "rootDir": "./src",
          "noUnusedParameters": true
      }
}
```

Optional Parameters

Note how we use || to coalesce the undefined y to zero before we use it. Any optional parameters must come after required parameters.

```
function f(x:number, y?: number) : number  {
    return x + (y ||0);
}

console.log(f(10));
console.log(f(10,10));

>> 10
>> 20
```

Default Parameters

Default parameters are considered the same as optional parameters and must come after any required parameters.

```
function f(x:number, y: number=0) : number  {
    return x + y;
}

console.log(f(10));
console.log(f(10,10));

>> 10
>> 20
```

REST PARAMETERS

A rest parameter can be specified as the final parameter after any required and option parameters.

```
function f(x:number, y: number=0, ...anyextras: number[]) : number  {
    return x + y + anyextras.reduce((a,b)=>a+b,0);
}

console.log(f(10));
console.log(f(10,10));
console.log(f(10,10,10,10));

>> 10
>> 20
>> 40
```

RETURN VALUES

JavaScript return undefined from any paths that do not provide an explicit return value. We can turn this off using a compiler flag.

```
{
    "compilerOptions": {
        "target": "ES2018",
        "outDir": "./dist",
        "rootDir": "./src",
        "noEmitOnError": true,
        "noImplicitReturns": true
    }
}
```

We can use a void type annotation to indicate there is not return value.

```
function f(x:number, y: number=0, ...anyextras: number[]) : void  {
    console.log( x + y + anyextras.reduce((a,b)=>a+b,0));
}
s
f(10,10,10,10,5);
```

Arrays

The following uses explicit annotation but the compiler would have implicitly inferred the type.

```
let nums: number[] = [1,2,3,5];

nums.forEach((val,idx) => {
    console.log(`${idx} = ${val}`);
})

>> 0 = 1
>> 1 = 2
>> 2 = 3
>> 3 = 5
```

Tuples

```
let arrofTuples: [number,string][] = [[1,"One"],[2,"Two"]]

arrofTuples.forEach((val) => {
    console.log(`${val[1]} = ${val[0]}`);
})

>> One = 1
>> Two = 2
```

Enums

Enums generate numbers in JavaScript

```
enum Fruit {Pear, Apple, Orange}

let f1: Fruit = Fruit.Apple;
console.log(f1);
console.log(Fruit[1]);

>> 1
>> Apple
```

Literal Value Types

Type Aliases

Objects

An objects shape is defined by its properties and their types.

```
enum ExerciseType { Put, Call }

let option: { strike: number, exType: ExerciseType }
    = { strike: 100.0, exType: ExerciseType.Call };
```

We can create type aliases for our shapes

```
enum ExerciseType { Put, Call }

type Option =
{
    strike: number,
    exType: ExerciseType
};

let option: Option
    = { strike: 150.35, exType: ExerciseType.Call };

console.log(option.strike);
```

Classes

```
class European {

    public readonly strike: number;
    public readonly underlying: string;
    public readonly exerciseType : ExerciseType;

    constructor(underlying: string, strike:number, exerciseType: Exercise
Type) {
        this.underlying = underlying;
        this.strike = strike;
        this.exerciseType = exerciseType;
    }

    public intrinsicValue(spot: number) : number {
        return this.exerciseType == ExerciseType.Call ?
            Math.max(spot-this.strike,0) :
            Math.max(this.strike-spot,0) ;
    }
}
```

TypeScript enables one to simply the specification of fields initialized by constructors. We can achieve the same result as

```
class European {

    constructor(public readonly underlying: string,
        public readonly strike:number,
        public readonly  exerciseType: ExerciseType) {
    }

    public intrinsicValue(spot: number) : number {
        return this.exerciseType == ExerciseType.Call ?
            Math.max(spot-this.strike,0) :
            Math.max(this.strike-spot,0) ;
    }
}
```

Generics

.vscode/launch.json

```
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=83
0387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "program": "${file}",
            "preLaunchTask": "tsc: build - tsconfig.json",
            "outFiles": ["${workspaceFolder}/dist/**/*.js"]
        }
    ]
}
```

Jest.config.js

```
module.exports = {
    "roots": ["src"],
    "transform":{"^.+\\.tsx?$": "ts-jest"}
}
```

Tsconfig.json

```
{
    "compilerOptions": {
        "target": "ES2018",
        "outDir": "./dist",
        "rootDir": "./src",
        "noEmitOnError": true,
        "sourceMap": true,
        "module": "commonjs",
        "declaration": true,
        "noImplicitAny": true,
        "strictNullChecks": true,
        "noUnusedParameters": true,
        "noImplicitReturns": true,
        "suppressExcessPropertyErrors": true,
        "strictPropertyInitialization": true

    }
}
```

Package.json

```
{
  "name": "intro",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "npx jest --watchAll",
    "start": "tsc-watch --onsuccess \" node dist/hello.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/jest": "^25.1.4",
    "jest": "^25.2.3",
    "ts-jest": "^25.2.1",
    "tsc-watch": "^4.2.3",
    "typescript": "^3.8.3"
  }
}
```

Development Environment

Node Commands

File/Folder/Command	Details
`npm install`	Install all packages specified in `package.json`
`npm list`	Show all local packages and their dependencies
`npm run`	Run a script specified in `package.json`

JavaScript/Typescript Project Structure

File/Folder/Command	Details
`package.json`	Describes a project's top-level dependencies. These are packages that have been added to a project using `npm install`
`package-lock.json`	All package dependencies for the project
`tsconfig.json`	TypeScript compiler configuration

Node Packages

```
npm init -yes
npm install --save-dev typescript ❶
npm install -save-dev tsc-watch ❷
npm install --save-dev jest ❸
npm install --save-dev @types/jest ❹
npm install --save-dev ts-jest   ❺
```

❶ **typescript**	The typescript compiler
❷ **tsc-watch**	Watches typescript files for changes. When it sees a change it compiles and can be configured to run a resulting JavaScript file after compilation
❸ **jest**	JavaScript testing framework
❹ **@types/jest**	Typescript types for the jest framework

❺ **ts-test**	Test utilities for TypeScript

TypeScript Compiler Options

Listing 1 tsconfig.json

```
{
    "compilerOptions": {
        "target": "ES2018", ❷
        "outDir": "./dist",
        "rootDir": "./src",
        "noEmitOnError": true,
        "sourceMap": true,
        "module": "commonjs" ❶
    }
}
```

❶ **module format**	Some environments such as node do not support ES2015 modules so specifying `commonjs` tells the compiler to generate older module code
❷ **target**	The version of JavaScript to target

PACKAGE.JSON

```
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "npx jest --watchAll",
    "start": "tsc-watch --onsuccess \" node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^4.2.3",
    "typescript": "^3.8.3"
  }
}
```

The bold lines specify scripts that can be run by npm. We have added a script called start that monitors files for change and executes the index.js when changed files have been compiled

Debugging

If we want to debug in VSCode we need to add a folder called `.vscode` into which we add a file called `launch.json`

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?link
id=830387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "program": "${workspaceFolder}\\dist\\index.js",
        }
    ]
}
```

We can then run our debugger using F5 in visual studio code

Unit Testing

Unit testing with Jest consists of two parts. The first part is to setup a configuration file called `jest.config.js` at the root level of our project. The following is a good example.

```
module.exports = {
    "roots": ["src"],
    "transform":{"^.+\\.tsx?$": "ts-jest"}
}
```

Then we simply add tests in our source code folder. If we have a module called `adder.ts` as follows

```
export function add(a: number, b: number): number {
    return a+b;
}
```

We can create a test called `adder.test.ts` as follows

```
import {add} from "./adder";

test("do a test", () => {
    let result = add(10,5);
    expect(result).toBe(15);
})
```

Putting it together

Often it is useful to have two terminal windows: one with a file watcher compiling and running our application and one running the tests.

```
npm start
npm test
```