TypeScript Fundamentals

Introduction to the TypeScript Language

Why TypeScript?

- JavaScript is great because of its reach
  - JavaScript is everywhere
- JavaScript is great because of available libraries
  - For server and client
- JavaScript (sometimes) sucks because of missing types
  - Limited editor support (IntelliSense, refactoring)
  - Runtime errors instead of compile-time errors > Our wish: Productivity of robustness of typed languages like C# or Java with reach of JavaScript

What is TypeScript?

- Superset of JavaScript
  - Valid JavaScript is (mostly) valid TypeScript
  - TypeScript defines add-ons to ECMAScript (primarily type information)
- Existing JavaScript code works perfectly with TypeScript
  - TypeScript compiles into JavaScript
  - Use it where you can use JavaScript
  - Compile-time error checking base on type information
- Generated code follows usual JavaScript patterns (e.g. pseudo-classes)
  - Built-in transpiler similar to *babel*
- Great tool support
  - e.g. Visual Studio Code

Install TypeScript

- Install locally: `npm install typescript --save-dev`
  - Run compiler `tsc` from NPM script
  - Run compiler from `node_modules`: `./node_modules/.bin/tsc`
- Install globally: `npm install --global typescript`
  - Run compiler from every folder with `tsc`
- Install TypeScript with development tools
- Tip: Consider *ts-node* to execute TypeScript files directly without compiling

Type Fundamentals

```
let n: number;         // typed variable
let a;                 // no type -> any
const s = "Max";       // contextual typing -> string

n = 5;                 // valid because 5 is a number
a = 5;                 // valid because a is of type any
a = "Hello";           // valid because a is of type any
n = "Hello";           // compile time error because  "Hello" is not a number
```

- Try it in TypeScript Playground
  - Try code navigation (right-click)
  - Try IntelliSense

Type Fundamentals (cont.)

- Types are used during editing and compiling
  - No type information in resulting JavaScript code
- Contextual Typing: Determine result type from expressions automatically
- Copy the following code into TypeScript Playground

    - Watch the transpiler work
    - Try IntelliSense

```
class Person {
    get firstName(): string { return "Tom"; }

    async doSomethingAsync(): Promise<number> {
        const result: number = await Promise.resolve(42);
        return result;
    }

    doSometing(callback: (result: number) => void) { callback(42); }
}

const p = new Person();
p.doSometing(result => console.log(result));
```

Basic Types

- TypeScript Handbook: Basic Types
- Important basic types:
    - `boolean`, `number`, `string`, `array`, `tuple`
    - `enum` = enumerations
    - `any` = type not known at compile time
    - `void` = no type at all
- Type assertions

> Important rule: Forget `var`, always use `const` or `let`

Basic Types (cont.)

```
// Basic data type 'boolean'
let aBoolean: boolean = false;
let anotherBoolean = false;     // Note type inference here
```

```
// Basic data type 'number' (=floating point value)
let decimal: number = 6;
let hex: number = 0xf00d;        // Note hex constant
let binary: number = 0b1010;     // Note binary constant
let octal: number = 0o744;       // Note octal constant

// Basic data type 'string'
let aString: string = "Hello World";
aString = 'Hello World';
let aTemplateString = `I say: ${aString}`;
// Note template string
let aMultilineString = `We like Typescript
    especially with Angular`;
```

Basic Types (cont.)

```
// Basic data type 'any'
let anything: any = false;
anything = 5.0;
let arrayOfAnything: any[] = [1, new Date(), 'Foo Bar', false];

// Note the type assertation here. The following lines do no runtime
 ↪  checking!
let aDecimal: number = <number>anything;
let aSecondDecimal: number = anything as number;

// Basic type 'Array'
let aList: number[] = [1, 2, 3, 4];
let aListWithDifferentTypes: (number | string)[] = [1, 'Hello'];
// Note 'Union Type' here
let anotherList = [1, 2, 3];
let yetAnotherList: Array<number> = [1, 2, 3];

// Note typesafe array operations.
aList.push(5);
//aList.push('Foo Bar');
```

Basic Types (cont.)

```
// Basic type 'Tuple'
let aTuple: [number, string] = [1, 'Hello'];
let aListOfTuples: Array<[number, string]> = [[1, 'Hello'], [2, 'World']];

// Note typesafe access of tuple members.
let numberInTuple: number = aTuple[0];
let stringInTuple: string = aTuple[1];
//numberInTuple = aTuple[1];

// Basic type 'enum'
enum Color { Red, Green, Blue };  // Note that first enum starts with value
↪  0
let anEnum: Color = Color.Green;// Assignment; anEnum gets value 1
enum Color2 { Red = 0b001, Green = 0b010, Blue = 0b100 };
let enumName: string = Color[2];// Note getting string name from enum (here
↪  'Blue')
enum AccessMode {
    Read = 0b01,
    Write = Read << 1,          // Write becomes 0b10
    ReadWrite = Read | Write    // Note computed member
};
console.log(AccessMode[3]);     // Prints 'ReadWrite'
```

Basic Types (cont.)

Note problems of `var` --> avoid it!

```
// If possible, don't use 'var' in your code anymore. 'let' protects
// you from unnecessary mistakes.
function printSquareWithMistake(sideLength: number) {
    for (var i = 0; i < sideLength; i++) {
        var line = 'dummy';
        var line = '';          // This is a mistake, but it works with
↪  'var'

        // Note that 'i' is declared a second time. As 'var' variables
        // are function-scoped, this is a bug!
        for (var i = 0; i < sideLength; i++) {
            line += '*';
        }

        console.log(line);
    }
}
printSquareWithMistake(3);
```

Basic Types (cont.)

```
function printSquare(sideLength: number) {
    for (let i = 0; i < sideLength; i++) {
        let line = '';
        //let line = 'dummy';
        for (let i = 0; i < sideLength; i++) {
            line += '*';
        }

        console.log(line);
    }
}
printSquare(3);
```

Objects

```
// Note that some code lines are commented in this sample. They
// would lead to compiler errors.

const anObject = { firstName: 'Foo', lastName: 'Bar', age: 99 };
anObject.firstName = 'John';

//anObject.anything = '...';
//anObject.age = "99";

// Note optional "age" in the following declaration
let anotherObject: { firstName: string, lastName: string, age?: number };
anotherObject = { firstName: 'Foo', lastName: 'Bar' };
```

Functions

- TypeScript Handbook: Functions
- `function` keyword vs. arrow functions
- Type inferrence
- Parameters (required, optional, default parameters)
- Advanced topics:
    - Rest parameters, details of `this`, overloads

Functions (cont.)

```
// Different types to declare functions
function add(x: number, y: number) { x + y };
const addLambdaWithoutTypes = (x, y) => x + y;

// Note that addLambdaWithoutTypes uses 'any'
const addLambda: (x: number, y: number) => number = (x: number, y: number)
↪   => x + y;
const addLambdaShorter: (x: number, y: number) => number =
    (x, y) => x + y; // Note that 'x' and 'y' are 'number' because of type
↪   inference.

// Optional and default parameters
function greetWithOptional(name: string, greeting?: string) {
    console.log(`${greeting || 'Hello'} ${name}!`);
};
greetWithOptional('John');

function greetWithDefault(name: string, greeting = 'Hello') {
    console.log(`${greeting} ${name}!`);
};
greetWithDefault('John');
```

Interfaces

- TypeScript Handbook: Interfaces
- Works differently compared to many other languages like C#
- "Duck Typing" aka *Structural Subtyping*
- Interfaces can *extend* each other
- Advanced topics:
    - Function types, indexable types, class types, hybrid types

Interfaces (cont.)

```
export interface IPerson {
    firstName: string;
    lastName: string;
    age?: number;               // Note optional member
}

export interface IPersonWithDescription extends IPerson {
    getDescription(): string;
}
```

Interfaces (cont.)

```
import {IPerson} from './interface'

export class Person implements IPerson {
  public firstName: string;
  public lastName: string;
  public age: number;

  // Note that 'Person' does not explicity say that it is
  // compatible with 'IPersonWithDescription', but it implicitly is
  // because all necessary members are implemented. This concept is called
  // 'structural subtyping' (details in
  // http://www.typescriptlang.org/docs/handbook/type-compatibility.html)
  public getDescription(): string {
    return `${this.firstName} ${this.lastName} is ${this.age} years old`;
  }
}
```

Duck Typing (cont.)

```
import {IPerson} from './interface'
import {Person} from './class-with-interface';

class SimplePerson {
    // Note that 'SimplePerson' does not explicity say that it is
    // compatible with 'IPerson', but it still is.
    constructor(public firstName: string, public lastName: string) { }

    public getDescription() { return `I am ${this.firstName}
 ↪  ${this.lastName}`; }

    get fullName() { return `${this.firstName} ${this.lastName}`; }
}

let p: IPerson;
p = new Person();
p = new SimplePerson('Foo', 'Bar');
console.log((<SimplePerson>p).fullName);
p = { firstName: 'Foo', lastName: 'Bar' };
p = { firstName: 'Foo', lastName: 'Bar', age: 99 };
//p = { firstNme: 'Foo', lastName: 'Bar', age: 99 };
```

Compatibility with any (cont.)

- Also note *Type Guard*

```
interface IPerson {
    firstName: string;
    lastName: string;
}
```

```
interface ICustomer extends IPerson {
    creditLimit: number;
}

function isCustomer(person: IPerson | ICustomer): person is ICustomer {
  return (person as ICustomer).creditLimit !== undefined;
}

const p = { firstName: 'Foo', lastName: 'Bar', creditLimit: 42 };
if (isCustomer(p)) { console.log(p.creditLimit); }
```

Classes

- TypeScript Handbook: Classes
- Constructors
- Accessibility of members: `public`, `private`, `protected`
- Static members vs. instance members
- Inheritance
- Abstract classes
- `readonly` properties
- Accessors

`for..of` and `for..in`

- TypeScript Handbook: Iterators and Generators
- `for..of` = iterate over iterable object (e.g. array)
- `for..in` = iterate over all keys of an object (see also *Object.keys()*)

Modules

- TypeScript Handbook: Modules

- Conceptually similar to ECMAScript modules
- `export`/`import`
- Ambient modules
  - `@types` on NPM
- Advanced topics:
  - Code generation for modules, optional module loading

Modules (cont.)

`module.ts`

```
export class MyFirstClass { public greeting: string = 'Hello'; }
```

```
export class MySecondClass { public greeting: string = 'Hi!'; }
```

`anotherModule.ts`

```
class MyThirdClass {
  public greeting: string = 'Yo!';
}

export default MyThirdClass;
```

Modules (cont.)

```
import * as myModule from './module';
import MyThirdClass from './anotherModule';

const c1 = new myModule.MyFirstClass();
console.log(c1.greeting);

const c2 = new myModule.MySecondClass();
console.log(c2.greeting);

const c3 = new MyThirdClass();
console.log(c3.greeting);
```

Exercise: Try this sample with different module systems (e.g. `--module commonjs`)

Declaration Files

- TypeScript Handbook: Declaration Files - Consumption
- Many libraries are written in JavaScript, not TypeScript
  - Black box for TypeScript compiler
- External declarations for globals (e.g. $ in jQuery), interfaces, etc. necessary
- TypeScript declaration files (`.d.ts`)
  - Similar to C++ header files
  - `npm install @types/...` (e.g. `npm install @types/chalk` for `chalk`)

Project Configuration

- TypeScript Handbook: *tsconfig.json*
- Compiler has a large number of compiler options
- Options can be passed...
  - ...on the command line (`tsc --help`)
  - ...in `tsconfig.json` (preferred)
- Tip: Generate basic `tsconfig.json` file with `tsc --init`

Important Compiler Options

- *lib*: List of library files to be included in the compilation (e.g. ES2015, DOM)
- *module*: Specify module code generation (e.g. *CommonJS*, *AMD*, *UMD*; see also What Is AMD, CommonJS, and UMD?)
- *moduleResolution*: Rule of thumb: Set it to *Node* if you include packages from NPM
- *outFile*, *outDir*: File and directory ouput structure
- *sourceMap*: Generate source map files for debugging

- *target*: ECMAScript target version (e.g. *ES2015*, *ES2016*)
- --*watch*: Run the compiler in *watch mode*: Watch input files and trigger recompilation on changes.
- --*version*: Print the compiler's version

Summary: TypeScript Goals

- TypeScript offers you the *reach* of JavaScript
- TypeScript makes you *more productive* (e.g. IntelliSense)
  - Ready for larger projects and larger teams
- TypeScript produces *less runtime errors*
  - Because of compile-time type checking