Chapter 1: Getting started with TypeScript

Section 1.1: 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.

Section 1.2: 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 throw 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.

Section 1.3: 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.

Section 1.4: 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.

Section 1.5: 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
```