Using Typescript to generate JavaScript

I've been playing with Typescript for a while, first because I was curious about the language, then because Angular uses it instead of ES2015 and later and finally because I came to realize that it helps me think through code as I write it rather than when I have to debug it.

Configuration and Installation

I am using Surma's Typescript Boilerplate Gist which is nothing more than a package.json file that you can drop into a directory and run to start your own projects or incorporate into your own existing ones.

The script that I will discuss here is `start`.

Running the `start` script will do the following:

1. Run the init task check if there is a typescript configuration file and, if there isn't one, to configure transpilation and modules for ES6
2. Install the required packages
3. Use the concurrently package to run the watch and serve scripts in parallel

That's all the script does and, if you're only playing with Typescript, you don't need anything else :)

The entire `package.json` file is shown below:

```
{
  "name": "ts-playgr"ts-playground""version": "1.0.0",
  "description": "Boilerplate for quick one-off TypeScript projects. Just
  "scripts"  "init": "tes": " tsconfig.json || (tsc --init -t ESNext -m ES
    "start": "": ""start""start"ncurrently \"npm run watch\" \"npm run ser
    "serve": "http-server"npm run watch\" \"npm run serve\""serve"uild":
  }": ""watch"": "Carlos Araya",
  "lice": ""build": "tsc -p ."
  },
```

```
  "author" "concurrently": "^3": ""license"ttp-server": "^0.11.1",
    ": "escript": "^2.8.1"
  }": ": "^3.5.1",
    "http-server": "^0.11.1",
    "typescript": "^2.8.1"
  }
}
```

To run the script open a terminal window, navigate to the directory where you dropped `package.json` and run:

```
npm run start
```

Why I like Typescript

As a developer, sometimes I'm lazy. Because I know what I want the code I'm writing to do, I don't pay much attention to what type I expect variables to be and dpn't document my assumptions beyond sparse comments in the script itself.

At least in the early stages of writing scripts, it helps me know what type each parameter should have and what I expect the return value to be.

It also helps with reaadability. If you make the type of parameters explicit, it'll help you and the person maintaining your code 6 months from now.

For example, the following

```
function divide(divisor: number, dividend: number) {
  return dividend / divisor;
}
```

In the example below we ensure that the content variable is a string before we convert it to upper case. In doing this we avoid implicit type conversions that can have unpredictable results.

```
function makeBig(content: string) {
  return content.toUpperCase();
}
```

If we need to provide support for browsers, we can transpile the code to Javascript (either ES5 or ES2015+) so it'll run natively in the browser.

So how does it work?

Boolean
: The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

Number
: As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

String
: Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type string to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

  You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

Array
: TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type.

  The second way uses a generic array type, Array

Tuple
: Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same.

When accessing an element with a known index, the correct type is retrieved.

Enum

A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members.

Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the any type.

The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect Object to play a similar role, as it does in other languages. But variables of type Object only allow you to assign any value to them - you can't call arbitrary methods on them, even ones that actually exist.

The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types

Void

void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value

Declaring variables of type void is not useful because you can only assign undefined or null to them.

Null and Undefined

In TypeScript, both undefined and null actually have their own types named undefined and null respectively. Much like void, they're not extremely useful on their own.

By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

However, when using the --strictNullChecks flag, null and undefined are only

assignable to void and their respective types. This helps avoid many common errors.

Never

The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn't assignable to never.

Object

object is a type that represents the non-primitive type, i.e. any thing that is not number, string, boolean, symbol, null, or undefined.

With object type, APIs like Object.create can be better represented.