TypeScript

CS-554 – WEB PROGRAMMING

What is TypeScript?

What is TypeScript?

TypeScript is an open source programming language developed by Microsoft that is a superset of JavaScript in order to provide a great deal of syntactical sugar.

- https://www.typescriptlang.org/

https://www.typescriptlang.org/play/index.html

What's the problem we're trying to solve?

Many programmers believe that adding structure and tooling are paramount to project success.

- Tools allow us to know something can fail before it does
- Structure provides patterns for us to follow to repeat duplicate work

In the wild-west that is known as JavaScript, there was historically a serious lack of structure and tooling until the last 5 years (approximately)

- One of the things that JS Devs most missed has been static type checking

How does TypeScript solve that problem?

TypeScript forces a build process step into your JavaScript, and therefore provides:

- Static type checking; never have to worry about passing a number when you should be passing a string again!
- Intellisense! By the addition of a type system, your IDEs can often more easily suggest what you are trying to write.
- Next-generation JS features today such as async / await support, transpiled down to today's JavaScript; similar to Babel

What are the downsides?

Additional build step

Another layer in your dev process that limits developer hiring pool.

Have to download typings for existing libraries to fully benefit from TS

- More dependencies, yay

How do we compile TypeScript?

TypeScript is compiled by installing the *typescript* module globally

- npm install -g typescript

This will install the TypeScript compiler which can run for the command line and be used to compile any project.

- Projects are often configured using a *tsconfig* file, which sets up basic rules

Local projects can also use a version of TypeScript defined locally

- This is allows you to set which version of TS to use; there are many!

How does it work?

Like most other build systems, you simply setup which files you want to target and run the compiler.

It will follow links to other TS files you reference, which will force those files to be built as well.

To resolve references to projects that were **not** created in TypeScript, you will need to download a *Type Definition* as well. This will allow your compiler to make sure you are using external libraries properly, as well.

- https://www.typescriptlang.org/docs/handbook/declaration-files/consumption.html

There are many definitions available for both server and frontend code, meaning you can easily write your backend and frontend TypeScript

- http://microsoft.github.io/TypeSearch/

TypeScript Usage

The majority of TypeScript is JavaScript

The majority of TypeScript is plain old vanilla JavaScript.

Any syntax that is valid in JavaScript will be valid in TypeScript, however you can write more advanced concepts in TypeScript that will be transpiled down into a particular version of JavaScript

- You can set this version in your tsconfig file
- Similar to babel in concept

Experimenting with TypeScript

To experiment with TypeScript, you can use the online TypeScript Playground maintained by Microsoft

- https://www.typescriptlang.org/play/index.html

This is fairly useful for a syntax checking and experimentation.

Integrating TypeScript

Integrating TypeScript into existing projects is extraordinarily easily

You can effectively start dropping in TypeScript one piece at a time and compiling smaller portions of your codebase.

Adding Types to methods

We can begin by adding TypeScript to existing methods in order to add static type checking to methods.

On the left is an example of a TypeScript file, while the right is the compiled JavaScript.

```typescript
function Greeter(greeting: string) {
    this.greeting = greeting;
}

Greeter.prototype.greet = function() {
    return "Hello, " + this.greeting;
}

let greeter = new Greeter("world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
};

document.body.appendChild(button);
```

```javascript
function Greeter(greeting) {
    this.greeting = greeting;
}
Greeter.prototype.greet = function () {
    return "Hello, " + this.greeting;
};
var greeter = new Greeter("world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};
document.body.appendChild(button);
```

Classes

Unlike vanilla JavaScript, we can easily build out our own classes in TypeScript that will get compiled down into objects.

This allows us to have strong type-checking for our custom data, as well as embrace more Object-Oriented Style Concepts.

In JavaScript, OOP is not necessarily always the answer, but composing logical data is generally a good thing.

```typescript
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}

document.body.appendChild(button);
```

```javascript
var Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());
var greeter = new Greeter("world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};
document.body.appendChild(button);

```

```
class Student {
    fullName: string;
    constructor(public firstName: string, public middleInitial: string, public lastName
: string) {
        this.fullName = firstName + " " + middleInitial + " " + lastName;
    }
}

interface Person {
    firstName: string;
    lastName: string;
}

function greeter(person : Person) {
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);
```

Interfaces

BECAUSE OF DUCK-TYPING, WE CAN DEFINE AND USE INTERFACES WE DO NOT EXPLICITLY INHERIT ON OUR CLASSES. IF IT LOOKS LIKE AN INTERFACE, IT ACTS LIKE AN INTERFACE.

Getters /Setters

It is easy to override getters and setters for your convenience, so that you do not have to continually call methods on your own. It allows for conveniently abstracting logic away.

```
1
2  class Employee {
3      private _fullName: string;
4
5      get fullName(): string {
6          return this._fullName;
7      }
8
9      set fullName(newName: string) {
10         if (newName) {
11             this._fullName = newName;
12         }
13         else {
14             throw ("Error: No name provided!");
15         }
16     }
17 }
18
19 let employee = new Employee();
20 employee.fullName = "Bob Smith";
21 if (employee.fullName) {
22     console.log(employee.fullName);
23 }
```

```
1  var Employee = /** @class */ (function () {
2      function Employee() {
3      }
4      Object.defineProperty(Employee.prototype, "fullName", {
5          get: function () {
6              return this._fullName;
7          },
8          set: function (newName) {
9              if (newName) {
10                 this._fullName = newName;
11             }
12             else {
13                 throw ("Error: No name provided!");
14             }
15         },
16         enumerable: true,
17         configurable: true
18     });
19     return Employee;
20 }());
21 var employee = new Employee();
22 employee.fullName = "Bob Smith";
23 if (employee.fullName) {
24     console.log(employee.fullName);
25 }
26
```

Enums

It is very strange that enums have not been added to JavaScript by this point.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];

alert(colorName); // Displays 'Green' as its value is 2 above
```

Generic Types

Generics are a way of allowing you to set up a type for a function to return, or a class to use, across your application. For example, **Array<string>** is actually a Generic Class, **Array**, with a typ parameter of **string**. As a result, all array methods will expect strings to be pushed, for filter to return a new array of strings, etc.

```
class Greeter<T> {
    greeting: T;
    constructor(message: T) {
        this.greeting = message;
    }
    greet() {
        return this.greeting;
    }
}

let greeter = new Greeter<string>("Hello, world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}

document.body.appendChild(button);
```

```
var Greeter = /** @class */ (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return this.greeting;
    };
    return Greeter;
}());
var greeter = new Greeter("Hello, world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};
document.body.appendChild(button);
```

String Literal Types

Similar to enums, you can also use strings that are bound to certain values.

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
        else {
            // error! should not pass null or undefined.
        }
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

Union Types

Union Types allow you to dictate that a parameter may be one of many types; such as expecting a **string** or a **number.**

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

Typescript with JSX

It is quite simple to have Typescript compile JSX with type checking.

- https://www.typescriptlang.org/docs/handbook/jsx.html
- https://www.typescriptlang.org/docs/handbook/react-&-webpack.html
- https://github.com/Microsoft/TypeScript-React-Starter#typescript-react-starter