Typescript - Making JS Less Terrible

What is Typescript

Javascript but with Types

And some other stuff

Getting Typescript

- First nodejs must be installed: https://nodejs.org/en/
- Then run command: npm install -g typescript
- To compile any typescript file run command: tsc filename.ts
- (Please note filename.ts is a placeholder name)
- Pretty much any code editor will have some typescript package although I recommend visual studio code(https://code.visualstudio.com/)

Why use Typescript

1. Allows for type checking
2. Good for large scale development compared to js
3. Ease to develop and debug for
4. Reducing error
5. Compiles to js so can be used on any site
6. Types can be inferred

Basic Types

- Booleans
- Numbers
- Strings
- Arrays denoted using [] or Array<T> where T is the type
- Null
- Undefined
- Any (pretty much what it says it is)
- Never (“never” worry about it , throw error or never terminate basically)
- Enums

Functions

- Defined using the keyword “function”
- Can specify the return type but this is optional
- Can write anonymous functions and assign them:

```
// Name because am unoriginal
function foo(x: number): number{
    return x;
}
```

```
let anon = (x:number, y: number) => { return x + y };
```

- Functions can also optional and default parameters

```
function foo(x: number, y:number = 5, z?: number): number{
    return x + y + (z || 0);
}
```

Functions

- And finally rest parameters, essentially this is for when you want to have a variable number of arguments for the function. Denoted using “…” followed by the parameter name and the array type

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Interfaces

- Interfaces contain a series of properties, each with their own name and type. Functions may also be properties.

```
interface rectangle {
    height: number;
    width: number;
};
```

- The properties can be instantiated in any order
-
```
let rect: rectangle = { width:0, height:0 };
```
- Optional properties in the interface are denoted using the “?”

```
interface rectangle {
    height?: number;
    width: number;
};
```

Interfaces

- Interfaces have inheritance! Using extends keyword
- Can have readonly properties of an interface
- Classes can implement interfaces which means they have to have the interfaces properties
- Indexable types

```
interface StringArray {
    [index: number]: string;
}
```

Classes

- Going to assume you know what a class is
- Syntax similar to Java/C#
- Can have private, public and protected, functions and attributes
- Protected means only a child of a class can use it, private only can be used with class, public you can probably guess
-

Classes(Examples)

```
class Parent {
    private someProp: boolean;
    protected constructor(value: boolean){
        this.someProp = value;
    }

    protected returnFive(){
        if (this.someProp){
            return 5;
        }
        else {
            return -1;
        }
    }
}

class Child extends Parent {
    constructor(){
        super(false);
    }
}

let p = new Parent(true);
let c = new Child();
```

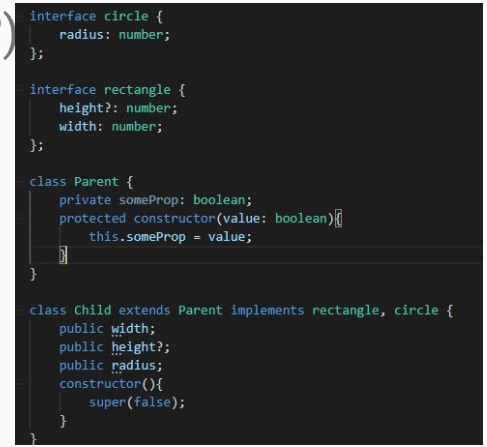
```
class Whatevs {
    private _sumVal: number;
    public constructor(x: number){
        this._sumVal = x * x;
    }

    public getSumVal(){
        return this._sumVal;
    }


}
```

Classes - Inheritance

- Classes can inherit properties from interfaces using the “implements” keyword, they are able to implement as many interfaces as you want.
- However to have classes be children of another class, use “extends”, however can only extend one other class,(or can you?)



```
interface circle {
    radius: number;
};

interface rectangle {
    height?: number;
    width: number;
};

class Parent {
    private someProp: boolean;
    protected constructor(value: boolean){
        this.someProp = value;
    }
}

class Child extends Parent implements rectangle, circle {
    public width;
    public height?;
    public radius;
    constructor(){
        super(false);
    }
}
```

Generics

- Allows you to make classes, interfaces and functions that can return any datatype but must have them specified. Defined using <T> in the function, class or interface, doesn't need to be T, can be any alphabet character.

```
interface someGeneric<T>{
    length: number;
    aVal: T;
    someReturn: (a: T, b: T) => number;
};

class UsingGenerics<T> {

    private _someVal: T;

    constructor(someVal: T){
        this._someVal = someVal;
    }

    public getSomeVal(): T{
        return this._someVal;
    }
```

Intersection Types

- An intersection type combines multiple types into one. This allows you to create new type which has the properties of both types. Can be really useful for inheritance.

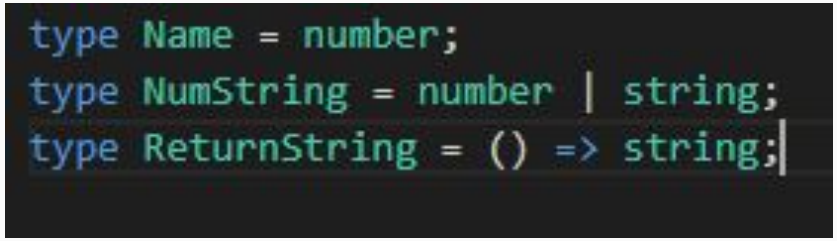
```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}
```

Union Types

- This is slightly counter intuitive as union types mean that function, variable, or property can be multiple types. This is done using the “|” character this can come in handy for functions.

```
function unionTypes(x: number | string) : number | string{
    console.log(x);
    return x;
}
```

Type Aliases

- Type aliases are just a way of giving a type another name. SImilar to interfaces but can be primitives, union types, function type etc.
- 
```
type Name = number;
type NumString = number | string;
type ReturnString = () => string;
```

Mixins - Buckle up kiddos

- Finally we get to the craziest thing in typescript mixins!!!
- So you want multiple inheritance for classes but typescript won’t let you do it
- To do this you state that it “implements” the classes it wants to inherit from. This means the methods of the
- Then you define a function which defines the function implementation of the child class using the parent classes’ implementation
- Simple right?

Mixins - Example

- So let's dissect this

```
class Disposable {
    isDisposed: boolean;
    dispose() {
        this.isDisposed = true;
    }
}
class Activatable {
    isActive: boolean;
    activate() {
        this.isActive = true;
    }
    deactivate() {
        this.isActive = false;
    }
}
class SmartObject implements Disposable, Activatable {
    constructor() {
        setInterval(() => console.log(this.isActive + " : " + this.isDisposed), 500);
    }
    interact() {
        this.activate();
    }
    // Disposable
    isDisposed: boolean = false;
    dispose: () => void;
    // Activatable
    isActive: boolean = false;
    activate: () => void;
    deactivate: () => void;
}
applyMixins(SmartObject, [Disposable, Activatable]);

function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
            derivedCtor.prototype[name] = baseCtor.prototype[name];
        });
    });
}
```

Mixins - Final Words

- Showed you these more as novelty
- You probably won’t need to ever use it.
- Or at least I pray you never need to use it
- Seen it used once and it wasn’t pretty

Rapping up

- This is only a “brief”(I know it doesn’t feel like that) look at Typescript there is much more you can do with it
- Typescript is open source so you can look at source code if you want to: https://github.com/Microsoft/TypeScript
- For more info about typescript go to: https://www.typescriptlang.org/docs/home.html
- Or if you have any Questions confront me in person if you want, I may run away though