Introduction to TypeScript – Day02

TypeScript is a superset of JavaScript, it is like JS on steriods. It provides syntactic sugar to basic JavaScript which in turn provides end to end safety.

TypeScript compiles down to JavaScript, which means that we can use it wherever you use JavaScript. JS can be used in the front end via a browser, in the back end via Node.js and Deno.

TypeScript uses type inference to give excellent type support without any additional download or IDE configuration.

In this bootcamp we will setup an environment for TS and then discuss some of it's capabilities. We will use several demos and illustrations throughout the bootcamp.

Part 1 – Classes

Continue using the same files from Part 6, just delete everything from the index.ts file. You could of course save the contents of that file if you find it useful, but here in Part 7 we start with a blank .ts file.

1. Here is a simple class in TS to get started with:

```
class Competition {
  constructor() {
  }
  addCompetitor() {
  }
  competitonDetails() {
  }
}
const competition = new Competition();
//competition.addCompetitor();
```

2. So far the class looks just like those in Java, C#, C++ and so on. Lets complete the *constructor* to accept the name of the competition. First add a *competitionName* property of the *string* type, then assign a value via the constructor

```
class Competition {
    competitionName : string;
    constructor(cName:string) {
        this.competitionName = cName;
    }
    addCompetitor() {

    }
    competitonDetails() {

    }
}
//const competition = new Competition();
//competition.addCompetitor();
```

Notice that if you add the property first, it shows an error until you assign that property's value via the constructor.

© Skillsoft

3. But there is a shortcut to this assignment, we can simply declare and initialize the property via the constructor in one statement:

```
class Competition {
//
    constructor(competitionName : string) {
        //
    }
    addCompetitor() {

    }
    competitonDetails() {

    }
}
const competition = new Competition("Weight Loss
Competition 2025");
//competition.addCompetitor();
```

4. We may still have a problem with accessing the property called *competitionName*. Try to complete the function `competitionDetails`():

```
        addCompetitor() {

        }
        competitonDetails() {
          return this.competitionName;
        }
    }
```

First, you must the *this* operator but then we get an error that *competitionName* does not exist.

5. As it turns out, if we use this type of property assignment, we must indicate in the constructor, what the scope of property it is. In other words add the appropriate *modifier*, just like with other languages, see code below:

```
    constructor(private competitionName : string) {
            //
        }
        addCompetitor() {
```

Note: the IDE will suggest a *Quick Fix*, this will also work, depending on your final goal.

6. Now we will add an array to hold competitors in this competition. In this case we will declare a property the traditional way and also make it private. In TS we have **private**, **public** and **protected**. Also, as in other languages, these modifiers extend to functions/methods.

```
class Competition {
    private competitors : string[] = [];
    constructor(competitionName : string) {
        //
    }
    addCompetitor() {
```

   Note: TS also has the *readonly* and the *?* (optional) modifiers.

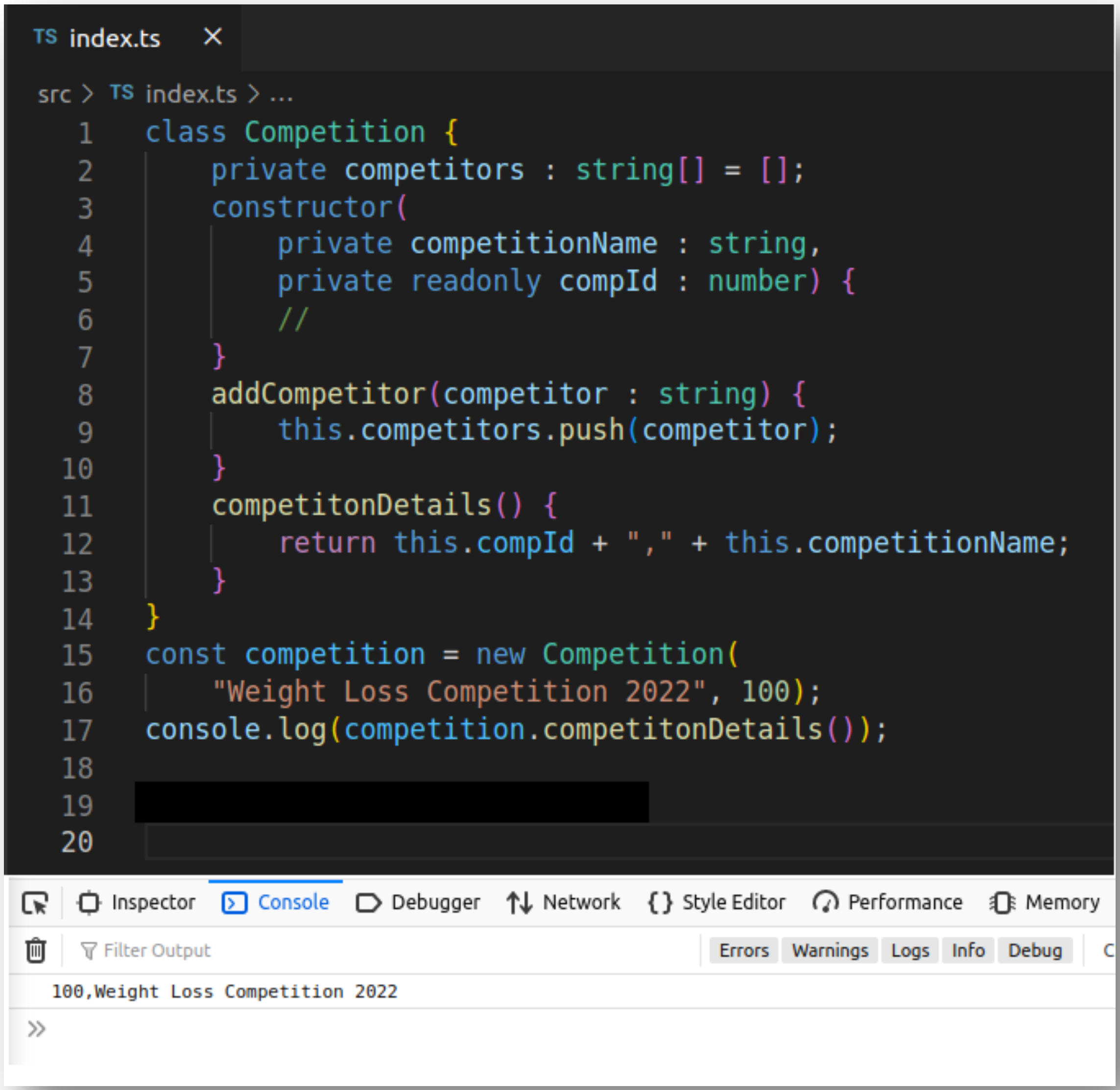
7. Now we can complete the *addCompetitor()* function:

```
class Competition {
    private competitors : string[] = [];
    constructor(private competitionName : string) {
        //
    }
    addCompetitor(competitor : string) {
        this.competitors.push(competitor);
    }
    competitonDetails() {
        return this.competitionName;
    }
```

8. TS also has the ability to make properties **read only**, so we can add a competition ID and change the competition details:

```
class Competition {
    private competitors : string[] = [];
    constructor(private competitionName : string,
private readonly compId : number) {
        //
    }
    addCompetitor(competitor : string) {
        this.competitors.push(competitor);
    }
    competitonDetails() {
        return this.compId + "," +
this.competitionName;
    }
}
```

```
    const competition = new Competition("Weight Loss
    Competition 2025", 100);
    console.log(competition.competitonDetails());
```



index.ts

src > index.ts > ...

```
class Competition {
    private competitors : string[] = [];
    constructor(
        private competitionName : string,
        private readonly compId : number) {
        //
    }
    addCompetitor(competitor : string) {
        this.competitors.push(competitor);
    }
    competitonDetails() {
        return this.compId + "," + this.competitionName;
    }
}
const competition = new Competition(
    "Weight Loss Competition 2022", 100);
console.log(competition.competitonDetails());

[redacted]

```

```
100,Weight Loss Competition 2022
```

9. At this point, below shows the code so far and the output as it appears in the Console window of my Firefox browser

10. We could also use *setters* and *getters* in TS, the following code shows how to retrieve the competition id and then print it out:

```
        }
        competitonDetails() {
            return this.compId + "," +
    this.competitionName;
        }
        get competitionID(){
            return this.compId;
        }
    }
    const competition = new Competition(
        "Weight Loss Competition 2022", 100);
    console.log(
        competition.competitonDetails() + ", ID: "
        + competition.competitionID
        );
```

Note that we do not **call** the competitionID as a function, so this wont work:
`competitionID()`

11. Setters are done in a similar way:

```
        private competitors : string[] = [];
        private admin : string = "";//add a new property
        constructor(
        …
        }
        //set the property added above
        set competitionAdmin(adminName: string){
            if(adminName != "Axle"){
                this.admin = adminName;
            }
        }
    }
```

Notice that we added some logic before accepting the new value for **`admin`**.

Part 2 – Interfaces

For this part you can save the existing code in index.ts into a separate file. For this section remove everything from index.ts, so lets start with a clean empty index.ts file.

1. Here is a basic interface for the competition we are running:

```
interface Competable {
    competitors : string[];
    admin : string;
    //
    addCompetitor(competitor : string) : void;
    competitonDetails() : string;
}
```

   Notice that we can still declare properties and functions, but the functions must show that they return something or nothing and they **cannot have any curly braces**. Also properties **cannot have any assignment**, not even null or undefined.

2. Now we can attempt to implement this interface for our weight loss competition but we immediately get an error. We have to satisfy all the properties and methods of the interface:

```
class wtLossCompetition implements Competable{

}
```

3. The following will be the basic class setup to satisfy the implementation. Notice that for now I just assigned an empty array and an empty string for admin:

```
class wtLossCompetition implements Competable{
    competitors: string[] = [];
    admin: string = "";
    addCompetitor(competitor: string): void {
    }
    competitonDetails(): string {
        return this.competitors.toString();
    }
}
```

4. Now in the implementation, we can add properties that are unique to the **weight loss competition** and methods:

```
class wtLossCompetition implements Competable{
    competitors: string[] = [];
    admin: string = "";
    backupAdmin : string = "";
    …
    setBackupAdmin(supportAdmin : string){
        this.backupAdmin = supportAdmin;
    }
}
```

5. Now that we have this template, we can add different competitions like **fantasy football**:

```
class FantasyFootbalCompetition implements Competable{
    competitors: string[] = [];
    admin: string = "";
    playerCap : number = 20;
    draftees: string[] = [];
    addCompetitor(competitor: string): void {

    }
    competitonDetails(): string {
        return this.competitors.toString();
    }
    draftPlayer(rookie : string){
        this.draftees.push(rookie);
    }
}
```

6. (Optional) If you wanted to designate a field in the interface as optional all you need to do is add a question mark after the name of the variable:

```
interface Competable {
    competitionName : string;
    competitors : string[];
    admin : string;
    backupAdmin? : string;
    //
    addCompetitor(competitor : string) : void;
    competitonDetails() : string;
}
```

Now, backupAdmin is optionally used when this interface is implemented.

7. Of course it is possible to have constructors in your concrete classes:

```
   class FantasyFootbalCompetition implements Competable{
       competitors: string[] = [];
       admin: string = "";
       playerCap : number = 20;
       draftees: string[] = [];
       constructor(cAdmin : string){
           this.admin = cAdmin;
       };
```

   It is possible to add a constructor to the interface itself but it is cumbersome.

Part 3 – Intersection, Union Types and Type Guards

Continue using the same files from Part 8 of Day 01, just delete everything from the index.ts file. You could of course save the contents of that file if you find it useful, but here on Day 02 we start with a blank .ts file.

1. In addition to Union types, **intersection** types also exist in TS:

```
type Competitor = {
    cName: string;
  }

type Admins = {
    adminID: number;
}

type adminCompetitor = Competitor & Admins;

const axle : adminCompetitor = {
    adminID : 123,
    cName : "Axle"
};
```

   The relationship between these two types should make sense to the developer for example Regular Employee, Part-Time Employee.

2. We could add date information to the Competitor type:

```
type Competitor = {
    cName: string;
    dateJoined : Date;
}
```

3. We would also have to adjust the *adminCompetitor* definition:

```
const axle : adminCompetitor = {
    adminID : 123,
    dateJoined : new Date("2022-06-30"),
    cName : "Axle"
};
```

4. Now we can do this:

```
console.log(axle.cName + " joined " +
axle.dateJoined.toLocaleDateString('en-US'));
```

5. Another situation can arise, we could have a union of two **object** types:

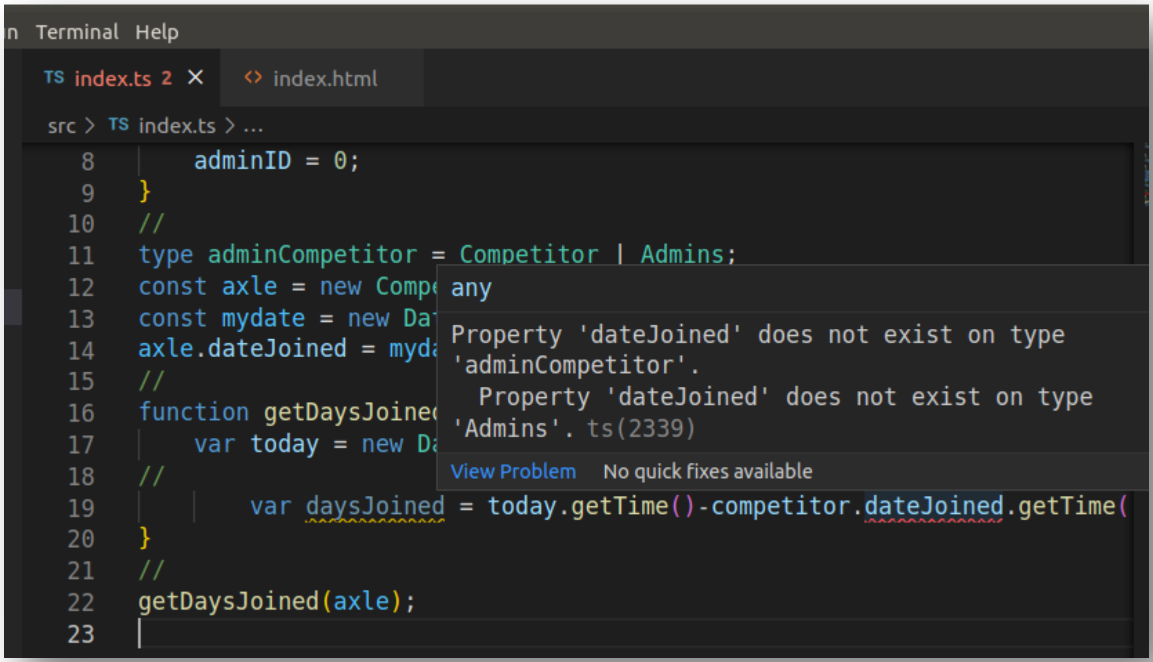
```
type adminCompetitor = Competitor | Admins;
```

6. Now, I could create a function like this to display the number of days the Competitor object has been involved in the competition:

```
function getDaysJoined(competitor : adminCompetitor){
    var today = new Date();
    var daysJoined = today.getTime()-
competitor.dateJoined.getTime();
}
```

As a side note, this example also demonstrates how to do date arithmetic in TS. The actual date methods you use will depend on your situation.

7. If you hover over *dateJoined*, you will notice that the IDE is complaining about not knowing about that property on *adminCompetitor*. This is because the TS engine cannot determine if the object is of type *Competitor* or *Admins*. We therefore have to use a **type guard** to force this identification.



```
    adminID = 0;
}
//
type adminCompetitor = Competitor | Admins;
const axle = new Compe
const mydate = new Da
axle.dateJoined = myda
//
function getDaysJoinec
    var today = new Da
//
        var daysJoined = today.getTime()-competitor.dateJoined.getTime(
}
//
getDaysJoined(axle);
```

```
any
Property 'dateJoined' does not exist on type
'adminCompetitor'.
  Property 'dateJoined' does not exist on type
'Admins'. ts(2339)
```

8. What we need to do is find out exactly which object type we have. The problem is that we cannot use *instanceof* or the *typeof* operators.

```
function getDaysJoined(competitor : adminCompetitor){
    var today = new Date();
//
    if(competitor instanceof Competitor){
        var daysJoined = today.getTime()-
competitor.dateJoined.getTime();
        console.log(daysJoined/(1000 * 60 * 60 * 24));
    }
}
```

The reason we cant use *typeof* is that in the end it is JS that is evaluating this code NOT TS. This means that JS can only compare the types it knows so object, string, Boolean etc. It does not know about Competitor so this comparison will not work.

9. The solution here is to check if the object we have, posseses something unique. As it turns out, dateJoined is unique to Competitor.

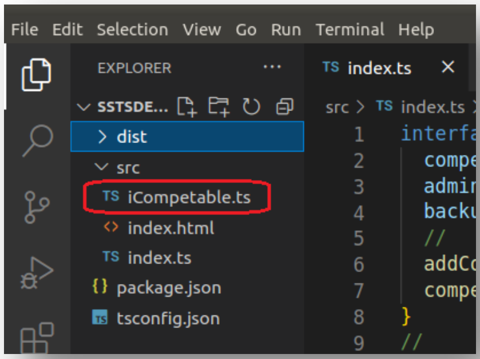
```
function getDaysJoined(competitor : adminCompetitor){
    var today = new Date();
    if("dateJoined" in competitor){
        var daysJoined = today.getTime()-
competitor.dateJoined.getTime();
        console.log(daysJoined/(1000 * 60 * 60 * 24));
    }
} }
```

Note: the final code for this section calls the function above and prints out the number of seconds since Axle joined the company.

Part 4 – Modularity

We will now distribute the code we have into different files and then re-import as necessary into the main file. Start with the code you had from Part8 of Day01. If you don't have that code, just unzip the zipped file provided and the code will be in the index.ts file in the src folder.

1. Create a new .ts file, in the src folder, for the interface and cut and paste all the code that it has currently. I called my file iCompetable.ts but it can be anything:



```
interfa
    compe
    admi
    backu
    //
    addC
    compe
}
//
```

2. Cut the interface code from where it is in the <u>index.ts</u> file and paste it all into the <u>iCompetable.ts</u> file:

```
    interface Competable {
        competitors : string[];
        admin : string;”
        backupAdmin? : string;
        //
        addCompetitor(competitor : string) : void;
        competitonDetails() : string;
    }
```

3. At this point if you compile everything should work but let us continue and do this the proper way. Before we continue repeat the same steps above, but for the wtLossCompetition class. So create a new ts file called wtLossCompetition.ts and insert into that file, the entire wtLossCompetition class. Also you may remove the FantasyFootbalCompetition class from the index.ts file.

4. Just so that we see something in the console window, create an instance of the wtLossCompetition class in the <u>index</u> file, and have it display some info:

```
    const wtloss = new wtLossCompetition();
```

5. At the moment the wtLossCompetition class does not have the ability to store a new player, so lets change that:

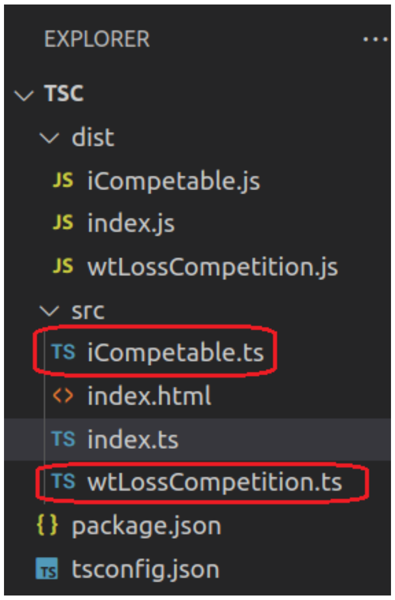
```
        admin: string = "";
        addCompetitor(competitor: string): void {
            this.competitors.push(competitor);
        }
        competitonDetails(): string {
```

6. With this setup we can add a new competitor in the index.ts file like this:

```
const wtloss = new wtLossCompetition();
wtloss.addCompetitor("Axle Barr");
console.log(wtloss.competitonDetails());
```

Note, at this point you may see an error message in the console window, it is complaining about the index.js file located n the dist folder. We will fix this shortly.

7. This is the file setup so far on my system:



```
EXPLORER
TSC
  dist
    iCompetable.js
    index.js
    wtLossCompetition.js
  src
    iCompetable.ts
    index.html
    index.ts
    wtLossCompetition.ts
  package.json
  tsconfig.json
```

8. Now add the export keyword in front of interface and the wtLossCompetition class in their respective files:

`export interface Competable { competitors : string[];`	`import {Competable} from "./iCompetable.js"; export class wtLossCompetition implements Competable{`

9. Over in index.ts you will notice that *Competable* is not found. Lets do the import now into that file as well as the `wtLossCompetition` file:

```
import { wtLossCompetition } from
"./wtLossCompetition.js";
//
class wtLossCompetition implements Competable{
  competitors: string[] = [];
```

Note: the import file is imported as a .js file NOT .ts. The project will compile with the either extension but upon execution it will fail in the browser. This is just an architectural decision, it can be changed later.

10. Also import Competable into the WtLossCompetition.ts file since it is used there:

```
import {Competable} from "./iCompetable.js";
export class WtLossCompetition implements Competable{
    competitors: string[] = [];
    admin: string = "";
```

11. Next step, add the *type* attribute with value of *module* to the index.html file:

```
        <link rel="shortcut icon" href="#" />
    <!-- <script>var exports = {};</script> -->
    <script type="module"
src="../dist/index.js"></script>
    <title>SS TypeScript Bootcamp</title>
</head>
```

12. Finally, change the tsconfig.json file to reflect that we are now using **ES6 modules**:

```
/* Modules */
      "module": "es2015",
/* Specify what module code is generated. */
      "rootDir": "./src",
/* Specify the root folder within your source files.
*/
      "outDir": "./dist",
```

13. Compile and now the program should work as before
    Here is the code for the three main classes at this point:

index.ts	iCompatable.ts	wtLossCompetition.ts

import { wtLossCompetition } from "./wtLossCompetition.js";
//
const wtloss = new wtLossCompetition();
wtloss.addCompetitor("Axle Barr");
	export interface Competable
	{
	  competitors : string[];
	  admin : string;
	  backupAdmin? : string;
	  //
	  addCompetitor(competitor : string) : void;
		import {Competable} from "./iCompetable.js";
		export class wtLossCompetition implements Competable{
		  backupAdmin : string = "";
		  competitors: string[] = [];
		  admin: string = "";
		  addCompetitor(competitor: string): void {
		    this.competitors.push(competitor);
		  }

console.log(
  wtloss.competitonDetails()
);
	competitonDetails() :
	string;
	}
		competitonDetails(): string {
		  return this.competitors.toString();
		}
		setBackupAdmin(supportAdmin : string){
		  this.backupAdmin = supportAdmin;
		} }

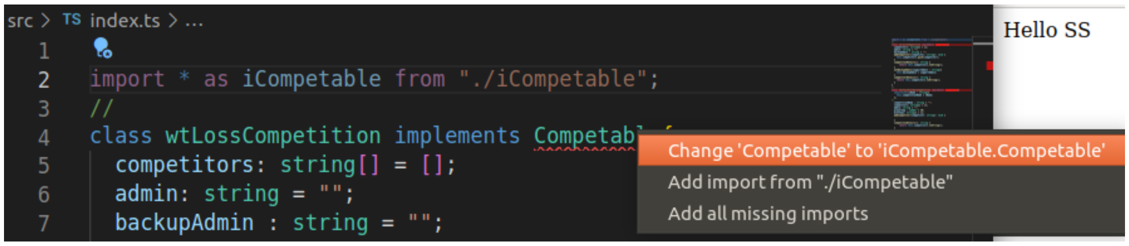
14. (Optional) An alternative way to import exportable objects, in the importing file use the * operator:

```
import * as iCompetable from "./iCompetable.js";
//
class wtLossCompetition implements  implements
Competable{
  competitors: string[] = [];
```

This is considered *grouping* and *aliasing*.

15. If you did this, you will have to change the implementation:
`class wtLossCompetition implements iCompetable.Competable{`
Notice the dot notation!

16. The IDE will help with this part:



```
import * as iCompetable from "./iCompetable";
//
class wtLossCompetition implements Competab
  competitors: string[] = [];
  admin: string = "";
  backupAdmin : string = "";
```

src > TS index.ts > ...
Hello SS
Change 'Competable' to 'iCompetable.Competable'
Add import from "./iCompetable"
Add all missing imports

Note: aliasing can be used on its own, without the grouping feature

17. (Optional) An alternative way to export exportable objects, is to use the default export feature. Note however, there can only be **one** default export per file:

```
export default interface Competable {
    competitors : string[];
    admin : string;
```

This is great if we are only exporting one thing.

18. (Optional) If you did export the default interface, the importation will now be different also:

```
import iComp from "./iCompetable.js";
```

© Skillsoft

```
//
class wtLossCompetition implements
iCompetable.Competable{
```

Notice that I could name the interface (or class or variable) anything, but of course where this *thing* is used also has to be changed, see below

19. (Optional) If you did export the default interface, and did the importation, you will now have to use that *alias* name for the import, so:

```
import iComp from "./iCompetable.js";
//
class wtLossCompetition implements iComp{
  competitors: string[] = [];
```

20. (Optional) If you look in the dist folder, you will now see two .js files

Part 5 – Generics

Converting the existing code to use generics presents a few challenges, lets see how to solve them. Continue using the files from the previous part, so from Part 2 above.

1. We will start with the interface since it is the most basic of all the components. The interface itself will be designated as generic but also we cannot now insist that the competitors array should be string, after al it is now generic

```
export default interface Competable<T> {
     competitors : T[];
     admin : string;
     backupAdmin? : string;
     //
     addCompetitor(competitor : T) : void;
     competitonDetails() : string;
   }
```

2. After this change, look at wtLossCompetition.ts, it shows an error when implementing *iComp*, we need to make this implementation generic also:

```
   import iComp from "./iCompetable.js";
   export class wtLossCompetition<T> implements iComp<T>{
       competitors: string[] = [];
```

3. Now *competitors* should show errors and that is because we made the array generic in the interface:

```
   export class wtLossCompetition<T> implements iComp<T>{
       competitors: T[] = [];
       admin: string = "";
       backupAdmin : string = "";
       addCompetitor(competitor: T): void {
         this.competitors.push(competitor);
       }
       competitonDetails(): string {
```

4. Also we may want to change the way competitionDetails work with the array of competitors. It should return an array of <T> instead of a specific string:

```
       }
       competitonDetails(): Array<T> {
           return this.competitors;
       }
       setBackupAdmin(supportAdmin : string){
```

5. We would also need to change iCompetable since competitionDetails depend on that interface. In other words, it should specify a return of a generic array rather than a specific string:

```
       addCompetitor(competitor : T) : void;
       competitonDetails() : Array<T>;
```

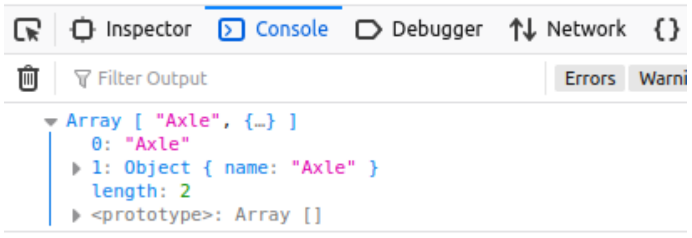
6. Assuming that we deleted the fantasy.ts file, compile and check the results. If you added a competitor using the *addCompetitor()* method, you will see the name show up in the developer's console of the browser. The competitors should now show up as an array of

names.

7. If it all works, lets us now try to add a competitor *object* in your index.ts file:

```
    const myCompetition = new wtLossCompetition();
    myCompetition.addCompetitor("Axle");
    myCompetition.addCompetitor({name:"Axle"});
    console.log(myCompetition.competitorDetails());
```

8. After compiling the result on my browser is shown below.



```
Array [ "Axle", {…} ]
  0: "Axle"
  1: Object { name: "Axle" }
  length: 2
  <prototype>: Array []
```

Part 6 – Decorators (Optional)

This topic still in **experimental** stage. We will attempt to render an HTML tag on the browser using a *decorator* function. For this part, you can continue using the files from Part 3. Also for this part you need to manually turn on experimental decorators in the tsconfig file:
`"experimentalDecorators": true,`

1. Lets start by decorating the class. Add this decorator just above the class, so it is a class decoroator:

```
    import {Competable} from "./iCompetable.js";
    @HiDOM
    export class wtLossCompetition<T> implements
    Competable<T>{
        backupAdmin : string = "";
        competitors: T[] = [];
```

The IDE will report an error but the error will go away once we define the accompanying function.

2. Now define a function with the same name as the decorator, below the class

```
            this.backupAdmin = supportAdmin;
        }
    }
    function HiDOM(){
        console.log("Hello");
    };
```

3. The function has to be coded with specific parameters. In this case the parameter represents the constructor of the class it is decorating, the *wtLossCompetition* class

```
    function HiDOM(target : Function){
        console.log("Hello");
    };
```

   If you now print `target` you will see the entire class in the console window

4. You can create Decorator Factories by returning a function from the decorator function itself. In this way you can pass parameters to the returned function:

```
    function HiDOM(target : Function){
        //console.log("Hello");
        return function(){

        }
    };
```

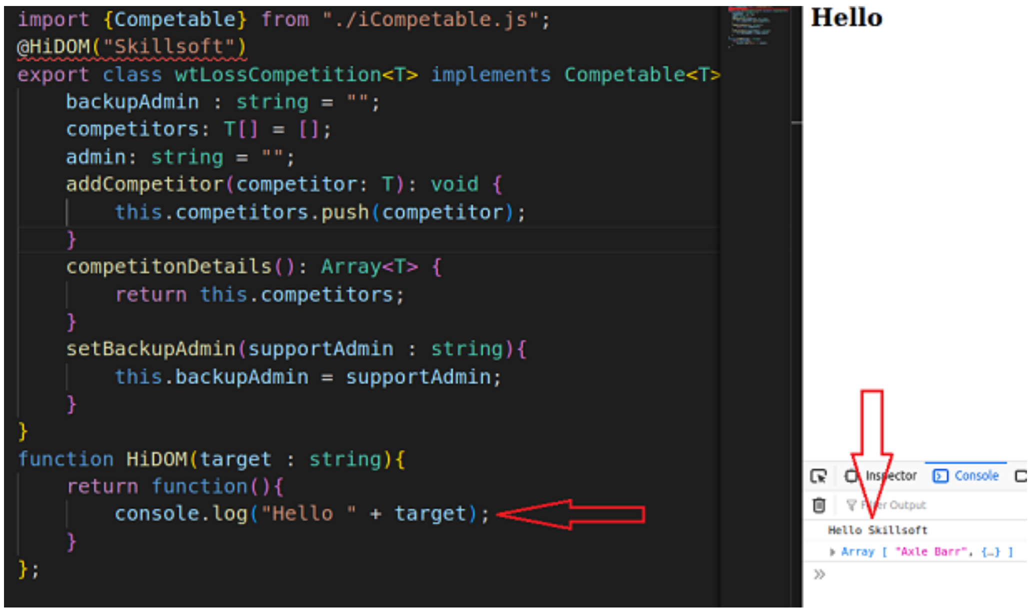
5. Continue building the inner function to use the cosole to log the target, change the target to a string:

```
    function HiDOM(target : string){
        return function(){
            console.log("Hello " + target);
        }
    };
```

6. Now change the function call to include a string parameter:

```
    import {Competable} from "./iCompetable.js";
    @HiDOM("Skillsoft")
    export class wtLossCompetition<T> implements Competable<T>{
        backupAdmin : string = ""
```

The @HiDOM line will still show an IDE error but if you compile and run, you will see the message being printed to the console window despite this error.



```
import {Competable} from "./iCompetable.js";
@HiDOM("Skillsoft")
export class wtLossCompetition<T> implements Competable<T>
    backupAdmin : string = "";
    competitors: T[] = [];
    admin: string = "";
    addCompetitor(competitor: T): void {
        this.competitors.push(competitor);
    }
    competitonDetails(): Array<T> {
        return this.competitors;
    }
    setBackupAdmin(supportAdmin : string){
        this.backupAdmin = supportAdmin;
    }
}
function HiDOM(target : string){
    return function(){
        console.log("Hello " + target);
    }
};
```

```
Hello
```

```
Hello Skillsoft
▸ Array [ "Axle Barr", {…} ]
»
```

This means that a parameter was passed from outer to inner functions

7. To fix the IDE error, under the function call, we need to give the inner function a parameter, it can be of any type. Also we can change the parameter of the outer function to more appropriate name:

```
    function HiDOM(msg : string){
        return function(constructor : any){
            console.log("Hello " + msg);
        }
    };
```

8. We could easily use this feature to pass a string to the DOM itself. First target an HTML element in the usual way. We would need an additional parameter to represent the part of the DOM we want to affect:

```
    function HiDOM(msg : string, el : string){
        return function(constructor : any){
            const pageTag = document.getElementById(el);
            console.log("Hello " + msg);
        }
```

9. Now create a new instance of the constructor:

```
        return function(constructor : any){
            const pageTag = document.getElementById(el);
            const myCompete = new constructor();
            console.log("Hello " + msg);
        }
```

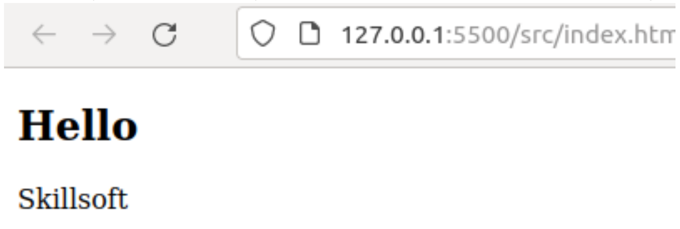
10. On the HTML file that you are using, make sure that you have an element with the name of *myDiv*, here I am using a div tag:

```
    <body>
        <h2>Hello</h2>
        <div id="myDiv"></div>
    </body>
    </html>
```

11. From where we made the function call, we need to tell the function which HTML element we want to use:

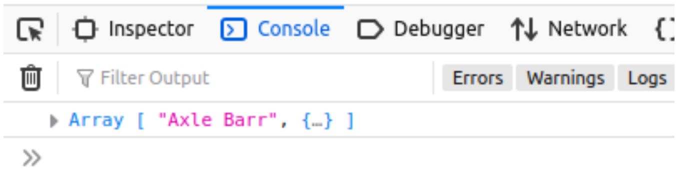
```
 import {Competable} from "./iCompetable.js";
@HiDOM("Skillsoft", "myDiv")
export class wtLossCompetition<T> implements Competable<T>{
   backupAdmin : string = "";
```

When you refresh, you should see the word *Skillsoft* appear in the browser window



Hello

Skillsoft



```
Array [ "Axle Barr", {…} ]
```

12. If this works, it means that we can use the technique shown here to build entire views like this:

```
import {Competable} from "./iCompetable.js";
@HiDOM("<h2>Hello from Skillsoft</h2>", "myDiv")
export class wtLossCompetition<T> implements
Competable<T>{
```

13. One last thing we do to make the code more robust is to check that the element was loaded and is available:

```
return function(constructor : any){
        const pageTag = document.getElementById(el);
        const myCompete = new constructor();
        if(pageTag){
            pageTag!.innerHTML = msg;
        }
```

14. Also, since we have an instance of the class itself, it means that we have access to all of the class properties and methods! Here I just show the current admin:

```
        return function(constructor : any){
            const pageTag = document.getElementById(el);
            const myCompete = new constructor();
            if(pageTag){
                pageTag!.innerHTML = msg + " Our admin is
   " + myCompete.admin;
            }
```

You can hardcode an admin in the class: admin: string = "Axle";

Appendix A – Webpack

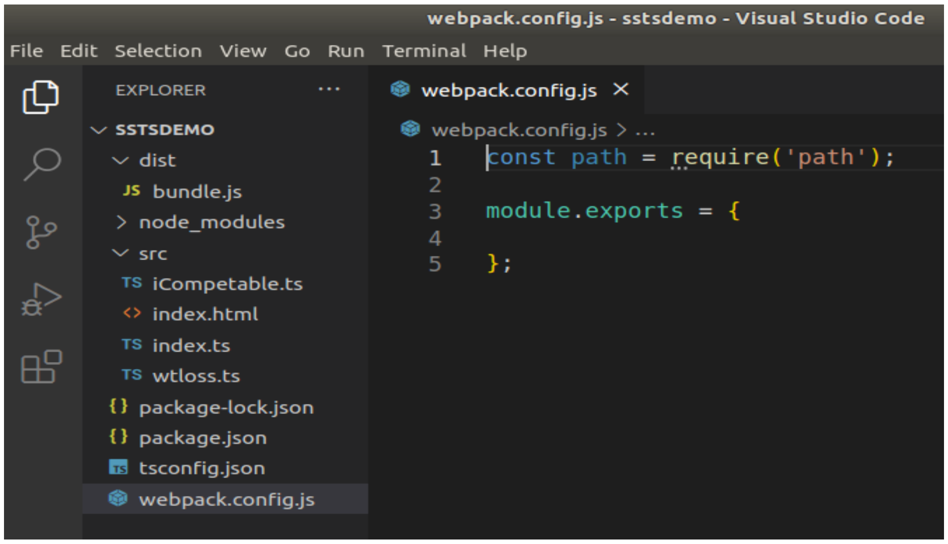
Webpack will bundle all our .ts and .js files into **just one** optimized and minimized output file. This will help with the number of http requests that the browser has to do in order to build a final view.

1. Before we do any project configuration, lets install Webpack from the terminal, so run this code from your terminal window: `npm install --save-dev webpack webpack-cli ts-loader`

2. We will start with the tsconfig.json file, where we will make sure that **es5** or above is set for target and module. Also remove *rootDir*, leave *outDir* and you can turn on *soureceMap* if you want debugging:

```
    "target": "es6",
    "module": "es2015",
    //"rootDir": "./src",
    "outDir": "./dist",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true,
    "noEmitOnError": true,
    "experimentalDecorators": true,
    "sourceMap": true
```

3. Add a new .js file into the root of your project directory. In my case my root directory is SSTSDEMO and I added a new file called webpack.config.js. Notice that there is already some code in the file. The first line will access the file system of your computer and third line is typical node code to export configuration details



```
const path = require('path');

module.exports = {

};
```

4. We will now configure an entry point and an output file for Webpack to use:

```
   module.exports = {
     entry: './src/index.ts',
     output: {
       filename: 'bundle.js',
       path: path.resolve(__dirname, 'dist')
     },
   //    devtool: 'inline-source-map',
   }
```

   The entry point tells WP where to start looking for files (to be configured soon). The output is of course where you want your output file to be stored, the filename here is bundle.js. The devtool config is optional if you want debugging support.

5. Next we configure rules for WP to follow when it starts the bundling process. First we will tell it which files need to be bundled:

```
     resolve: {
       extensions: ['.ts', '.js']
     }
```

   This tells WP to resolve or work with .ts and .js files. For this course at least all the code for Webpack configuration will be inside of the *module.exports* curly braces.

6. Next we need a module configuration. This section will contain rules for WebPack to follow. Usually the rules will go above *resolve*:

```
     module: {
       rules: [

       ]
     },
     resolve: {
       extensions: ['.ts', '.js']
     }
```

Notice that the rules is an array of objects we can pass to module

7. Here is the rest of the rules within module:

```
     module: {
       rules: [
         {
           test: /\.ts$/,
           use: 'ts-loader',
           exclude: /node_modules/
         }
       ]
     },
```

So we test for any file that ends with .ts and we use the *ts-loader* engine to turn these files into .js files and bundle them into one big file. We exclude anything in the node_modules folder, which is a **NodeJS** folder and is quite large.

8. Here is the entire config file:

```
   const path = require('path');
   module.exports = {
     entry: './src/index.ts',
     output: {
       filename: 'bundle.js',
       path: path.resolve(__dirname, 'dist')
     },
   //   devtool: 'inline-source-map',
     module: {
       rules: [
         {
           test: /\.ts$/,
           use: 'ts-loader',
           exclude: /node_modules/
         }
       ]
```

```
    },
    resolve: {
      extensions: ['.ts', '.js']
    }
  };
```

9. In the *package.json* file, we would need to add a command to run Webpack, although this can be done via the CLI, if installed on the OS itself:

```
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit
  1",
      "build":"webpack"
    },
    "keywords": [],
```

10. Change the `<script>` tags in index.html to reflect the bundle.js file we configured in step 3 above:

```
      <link rel="shortcut icon" href="#" />
      <script type="module"
  src="../dist/bundle.js"></script>
      <title>SS TypeScript Bootcamp</title>
```

11. Finally remove all files from the /dist folder and compile, it should all work as before but with just one bundled .js file.

12. To get Webpack to do its job, from the command line just type in or run:
    `npm run build`

    The /dist folder should now have the one bundle.js file and the browser should work as before with the previous code we had from Part 5

13. If you get an error about TypeScript not being installed, check your package.json file. If TypeScript is not there install it again usning npm. If all goes well you can test this new setup by creating an object and having it print something:

```
  const axle = <Employee>{
      empName : "Axle",
      payRate : 2,
      status : 'regular'
  }
  //
  calculatePay(axle);
```

14. Here is the entire index.ts file:

```
type Employee = {
    empName : string,
    payRate : Number,
    status : 'regular'
}
type Admin = {
    adminID : Number,
    salaryGrade : Number,
    status : 'management'
}
type PayrollEmployee = Employee | Admin;
function calculatePay(payrollEmployee:
PayrollEmployee){
    //if(payrollEmployee instanceof Employee)
    switch(payrollEmployee.status){
        case "management":
            console.log("Manager");
            break;
        case "regular":
            console.log("Regular Employee");

    }
}
const axle = <Employee>{
    empName : "Axle",
    payRate : 2,
    status : 'regular'
}
calculatePay(axle);
```

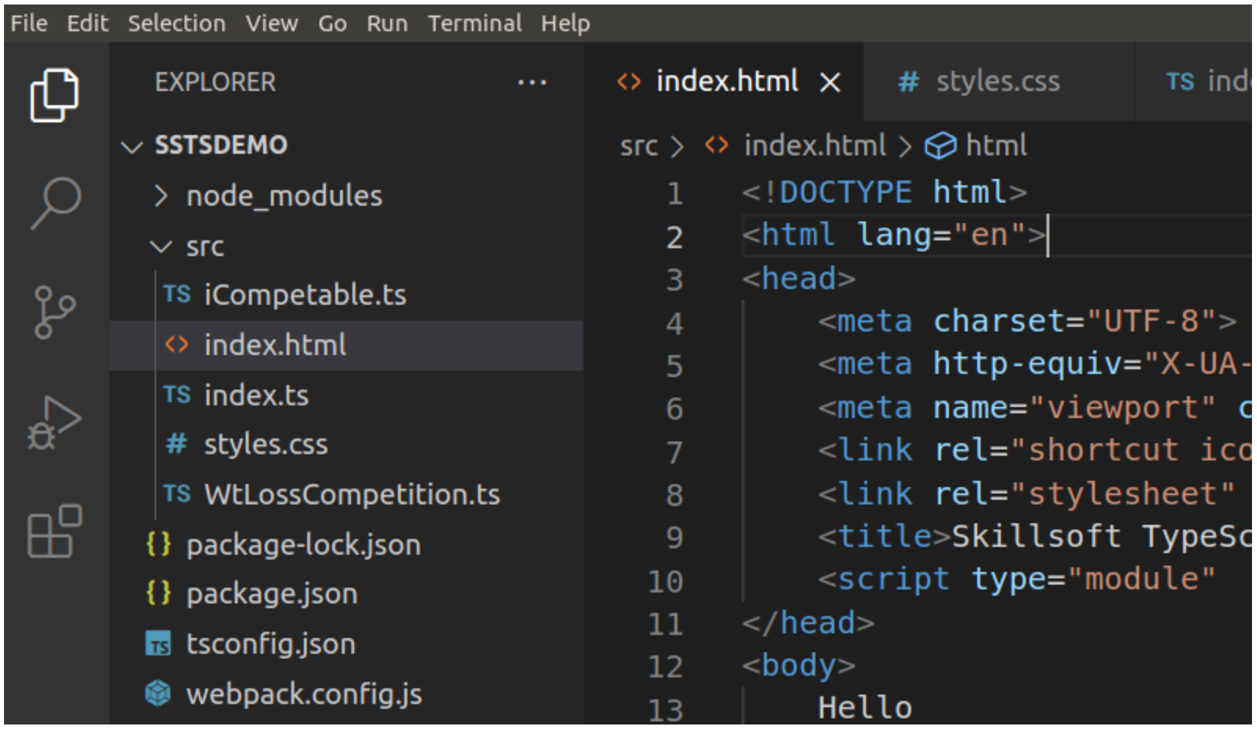
15. (Optional) If you wanted to see a more interesting example just add a .css file to the src folder. It can for example add a style to the “myDiv” div element.

```
div{
    background-color: lightgoldenrodyellow;
    padding: 5px 10px;
     margin: 0px;
    font-size: xx-large;
}
```

16. (Optional) Add this line to your HTML file in the <head> section:

```
      <link rel="shortcut icon" href="#">
       <link rel="stylesheet" type="text/css"
   href="./styles.css" />
       <title>Skillsoft TypeScript Demos</title>
       <script type="module"
   src="./../dist/bundle.js"></script>
```

17. Now add in the files that you had from Part 4. They should be iCompetable, and WtLossCompetition. These two files will go into the src folder along with the two index files:



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-
    <meta name="viewport" c
    <link rel="shortcut ico
    <link rel="stylesheet"
    <title>Skillsoft TypeSc
    <script type="module"
</head>
<body>
    Hello
```

18. Remove the dist folder if you have one and run the npm run build command again from the command line. This will produce a new dist folder with just one file in it. If you choose thi option do not run tsc anymore, just open the index.html file via the built-in web server we install on day1.

Appendix B – Union - Discriminated Types

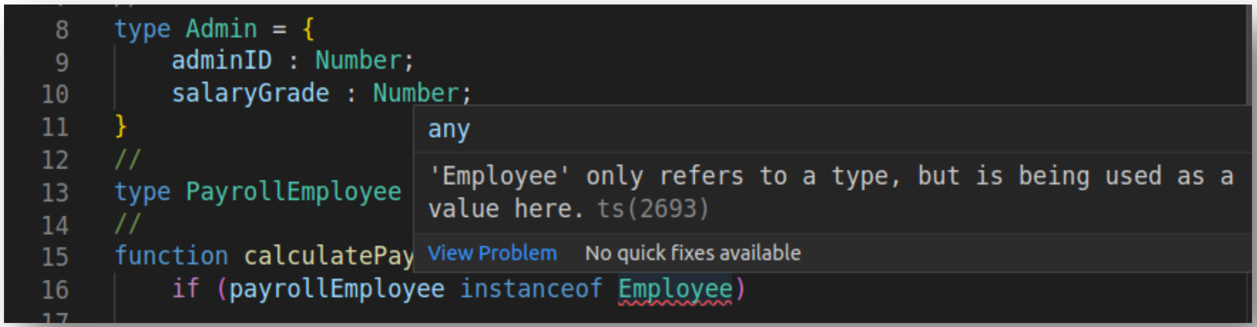
Sometimes you may have types that are related but need more fine-grain control over the derived types. For this part you just need an empty index.ts file and an index.html file, nothing else.

1. In this example we have two types, a union type and also a function that is supposed to calculate the final salary of these two related but different types:

```
type Employee = {
    empName : string;
    dateJoined : Date;
    payRate : Number;
}
//
type Admin = {
    adminID : Number;
    salaryGrade : Number;
}
//
type PayrollEmployee = Employee | Admin;
//
function calculatePay(payrollEmployee:
PayrollEmployee){

}
```

2. One technique we might use is the *instanceof* operator, but it does not work on *types*:



```
type Admin = {
    adminID : Number;
    salaryGrade : Number;
}
//
type PayrollEmployee
//
function calculatePay
    if (payrollEmployee instanceof Employee)
```

```
any
'Employee' only refers to a type, but is being used as a
value here. ts(2693)
```

From the image above Employee refers to a *type* but is being used as a *value*.

3. There exist a pattern that we can apply here. If we add a *property*, the same property to each of the types, we can then test on that *property*:

```
    type Employee = {
        status : 'regular';
        empName : string;
        dateJoined : Date;
        payRate : Number;
    }
    //
    type Admin = {
        status : 'management';
        adminID : Number;
        salaryGrade : Number;
    }
```

4. Now that we have this new property on each Type, we can use it in a Switch statement and we also get type support from the IDE:

```
    function calculatePay(payrollEmployee:
    PayrollEmployee){
        switch (payrollEmployee.status){

        }
    }
```

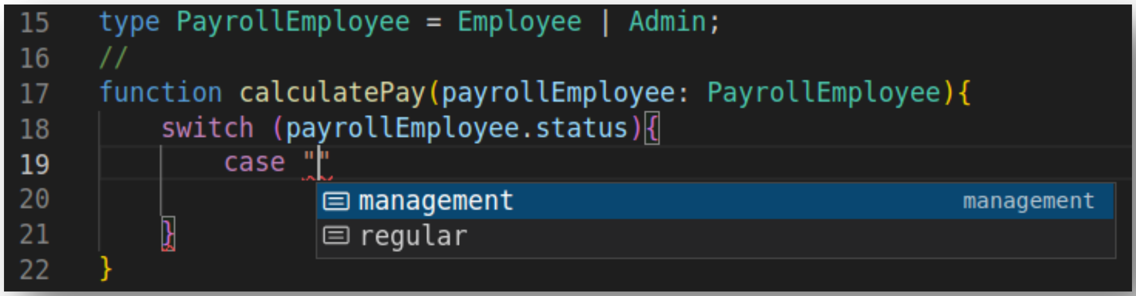
Notice that this code is being placed into the *calculatePay()* function

5. Now we can get type support when we test each case:

```
    function calculatePay(payrollEmployee:
    PayrollEmployee){
        switch (payrollEmployee.status){
            case "management":
                console.log("Manager");
                break;

        }
    }
```

6. Here is an image showing that we get two options for **case**, *management* or *regular*



```
type PayrollEmployee = Employee | Admin;
//
function calculatePay(payrollEmployee: PayrollEmployee){
    switch (payrollEmployee.status){
        case ""

    }
}
```

7. Here is the entire function

```
   function calculatePay(payrollEmployee:
   PayrollEmployee){
       switch (payrollEmployee.status){
           case "management":
               console.log("Manager");
               break;
           case 'regular':
               console.log("Regular");
       }
   }
```

Appendix C : Zipped Files

The zipped files represent the stat of the code after the instructions in the corresponding Word document. For example the file sstsdemo_d1_p2.zip is what the code looks like after all the instructions were followed from the TypeScript_Day01.docx document. In that document, the zipped file represents the code in Part 2 – HTML Setup.

Just unzip the files and follow the instructions on how to execute those files from the Word document. In most cases you would compile the ts files into js files using the tsc command. For the last zipped file on Day02, it is a little different.

If you wanted to use the files from the last example of Day02, you must first use a clean folder. Unzip the files and it should unzip into its own folder. Once you unzip the files, open your terminal window and point it to that folder. Run the command npm install. That command will create new files and folders. Only now should you open the folder in VS Code. Once you open in the IDE open a new terminal and run the npm run build command according to the instructions for Part6. Finally open the index.html file, located in the src folder, with the live server plugin for VS Code.

Appendix B : Topics

Part 1 – Classes

Part 2 – Interfaces

Part 3 –

Part 4 –

Part 5 –

Part 6 –