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

Part 2 – Interfaces

Part 3 – Intersection, Union Types and Type Guards

Part 4 – Modularity

Part 5 – Generics

Part 6 – Decorators (time permitting)

Part 1 – Classes

Continue using the same files from Part 6 on Day01, 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();
```

   This class has one constructor and two methods.

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.

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();
```

This is called *parameter properties*.

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, but see below.

The *private* keyword before *competitionName* has two functions in this scenario. It's not just marking the parameter as private - it's actually creating a private class property and automatically assigning the constructor parameter to it. This is a TypeScript-specific feature called a **parameter property** or **shorthand property declaration**.

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.

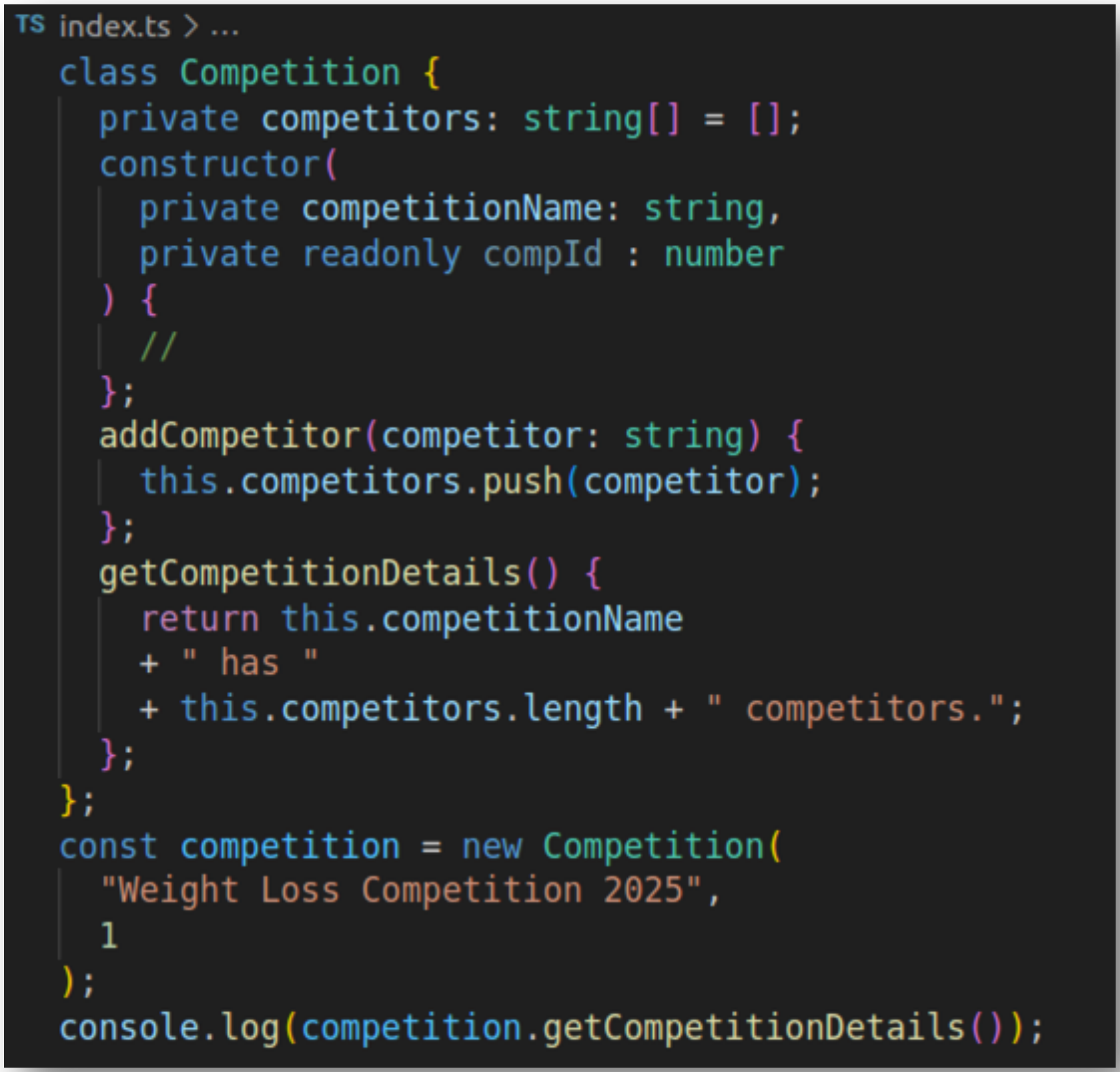
7. Now we can complete the *addCompetitor()* function:

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

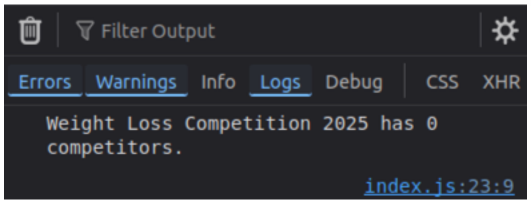
8. TS also has the ability to make properties and functions **read only**, so we can add a competition ID and change the competition details:

```
class Competition {
  private competitors: string[] = [];
  constructor(
    private competitionName: string,
    private readonly compId : number
  ) {
    //
  };
  addCompetitor(competitor: string) {
    this.competitors.push(competitor);
  };
  getCompetitionDetails() {
    return this.competitionName
    + " has "
    + this.competitors.length + " competitors.";
  };
};
const competition = new Competition("Weight Loss
Competition 2025", 1);
console.log(competition.getCompetitionDetails());
```

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



```
class Competition {
  private competitors: string[] = [];
  constructor(
    private competitionName: string,
    private readonly compId : number
  ) {
    //
  };
  addCompetitor(competitor: string) {
    this.competitors.push(competitor);
  };
  getCompetitionDetails() {
    return this.competitionName
    + " has "
    + this.competitors.length + " competitors.";
  };
};
const competition = new Competition(
  "Weight Loss Competition 2025",
  1
);
console.log(competition.getCompetitionDetails());
```



```
Weight Loss Competition 2025 has 0
competitors.
                        index.js:23:9
```

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:

```
        this.competitors.push(competitor);
      };
      getCompetitionId() {
        return this.compId;
      };
      getCompetitionDetails() {
        return this.competitionName
```

Note that you do not have to use a separate *get* keyword, but it wont be flagged if you did use that construction.

11. Setters are done in a similar way:

```
        private competitors : string[] = [];
        private admin : string = "";//add a new property
        constructor(
        …
        }
        //set the property added above
        setCompetitionAdmin(admin: string) {
          if(admin != "Axle"){
            this.admin = admin;
          } else {
            throw new Error("Admin cannot be Axle");
        };
```

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();
    }
}
```

   Although a constructor is not defined, every class has an empty constructor by default. In this case a constructor could be use to assign an empty string to admin.

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 = "";
    private leagueName: string = "";
    private seasonYear: number =
    new Date().getFullYear();

    addCompetitor(competitor: string): void {

    }
    competitonDetails(): string {
        return this.competitors.toString();
    }

    setSeasonYear(year: number): void {
      this.seasonYear = year;
    }

    getLeagueName(): string {
      return this.leagueName;
    }
  }
```

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 FantasyFootballCompetition implements Competable
   {
     competitors: string[] = [];
     admin: string = "";
     private leagueName: string = "";
     private seasonYear: number =
      new Date().getFullYear();

     constructor(leagueName: string, admin: string) {
       this.leagueName = leagueName;
       this.admin = admin;
     }
```

   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.

An intersection type crteates a single new type from two or more other types. The new type will have **all the properties and methods** of all the combined types. The resulting type includes all members from each type. Use intersection when you need an object to have characteristics from several different definitions.

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 adminCompetitor: AdminCompetitor = {
  cName: "Axle Barr",
  adminID: 12345
};
```

   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("2021-06-30"),
    cName : "Axle"
};
```

4. Now we can do this:

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

A better solution would be to create a function and display the necessary values:

```
const displayAdminCompetitor = (admin: AdminCompetitor): void => {
 console.log(`Admin Competitor Name: ${admin.cName}`);
 console.log(`Admin ID: ${admin.adminID}`);
 console.log(`Date Joined: ${admin.dateJoined.toLocaleDateString('en-US')}`);
};
```

A **union type** in TypeScript is where a variable (or parameter or return) can hold a value that can be **one of several specified types**. You define it using the vertical bar (|) basically "OR". Use a union when your variable wants to point to an object at some point in time. For example you may be dealing with employee ids which could be a string or a number. A union gives you the flexibility of handling either situation.

5. Here we could have a union of two **object** types:

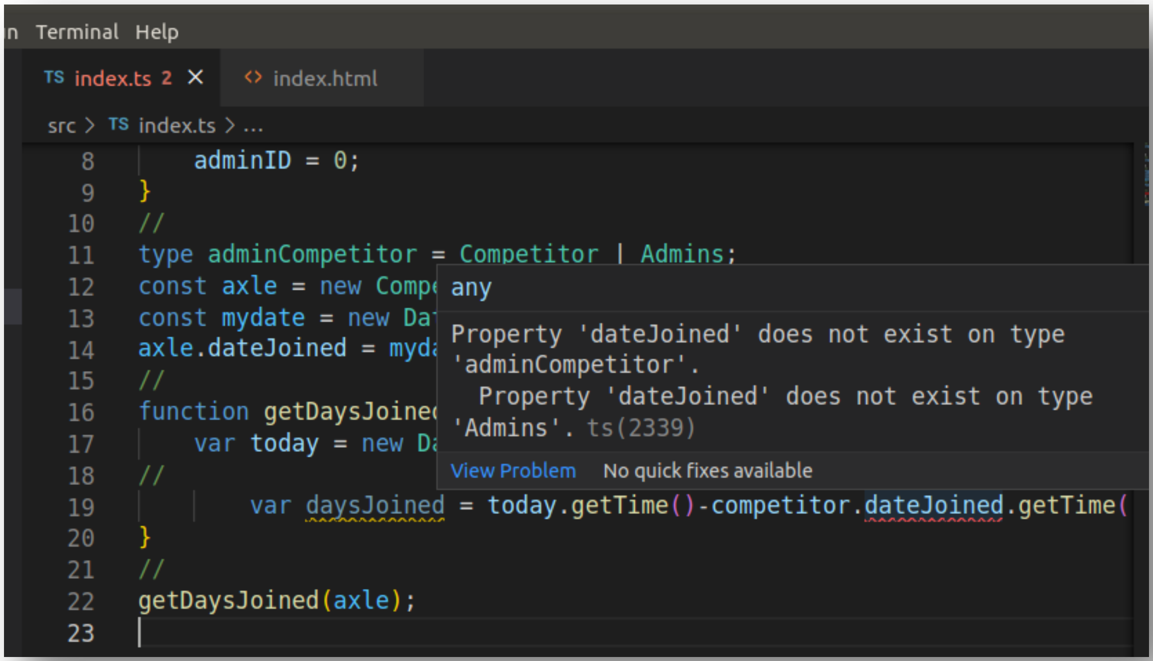
```
type adminCompetitor = Competitor | Admins;
```

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

```
const daysSinceJoined = (competitor: adminCompetitor):
number => {
  const today = new Date();
  const joinedDate = new Date(competitor.dateJoined);
  const timeDiff = today.getTime() - joinedDate.getTime();
  return Math.floor(timeDiff / (1000 * 3600 * 24));
};
```

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 getDaysJoine
    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 now is find out exactly which object type we have (type guard). The problem here 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 AdminCompetitor.

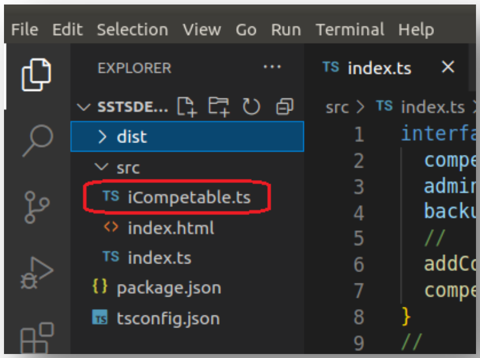
```
const daysSinceJoined = (competitor: adminCompetitor):
number => {
  const today = new Date();
  if (!('dateJoined' in competitor)) {
    return 0;
    // Regular competitors have no dateJoined property
  }
  const joinedDate = new Date(competitor.dateJoined);
```

Note: there are situations when the typeof, instanceof and user-defined type guards are more appropriate. This is an advanced topic.

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 Part2 of Day02. 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. If you are using the file mentioned, remove the FantasyFootballCompetition class. Lets just work with one class for now.

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



Note: as we are working in the browser (not in Node) all files must be relative to the browser's path.

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

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

3. At this point if you compile, you shoud have no errors, but let us continue and do this the proper way. You will notice that we now have two JS files in the dist folder.

4. Just so that we see something in the console window, create an instance of the wtLossCompetition class in the index 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:

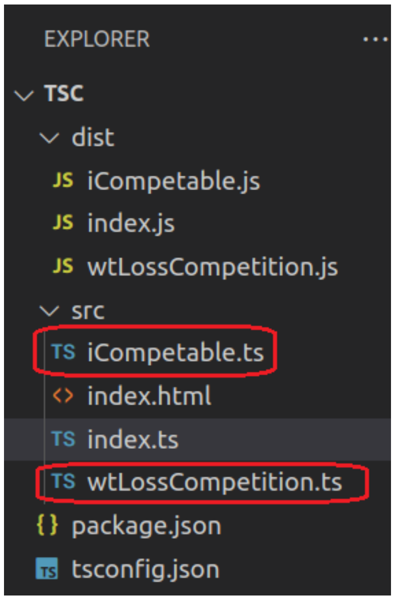
```
    admin: string = "";
    addCompetitor(competitor: string): void {
        this.competitors.push(competitor);
    }
    competitonDetails(): string {
```

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 then import the interface in the WtLossCompetition.ts file:

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

9. 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>
```

10. 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",
```

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

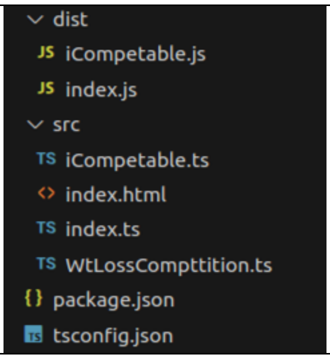
index.ts

```
import {Competable} from "./iCompetable.js";
class wtLossCompetition implements Competable{
  competitors: string[] = [];
  admin: string = "";
  backupAdmin : string = "";
  addCompetitor(competitor: string): void {
   this.competitors.push(competitor);
  };
  competitonDetails(): string {
   return this.competitors.toString();
  };
   setBackupAdmin(supportAdmin : string){
   this.backupAdmin = supportAdmin;
  };
};
const wtloss = new wtLossCompetition();
wtloss.addCompetitor("Axle Barr");
console.log(wtloss.competitonDetails());
```

iCompatable.ts

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

12. Lets try to do something similar with the class itself, so wtLossCompetition. First create a new TS file in the src folder called WtLossCompetition.ts:



```
dist
  iCompetable.js
  index.js
src
  iCompetable.ts
  index.html
  index.ts
  WtLossCompttition.ts
package.json
tsconfig.json
```

13. Just like before cut the entire class an insert it into the new file and yes, there will be lots of red lines everywhere. We will fix them one at a time.

14. Starting with the WtLossComptition.ts file, we need to import the Competable interface into this file. Since we no longer need it in the index.ts file, you can remove the import line from there:

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

This is great if we are only exporting one thing.

15. Now we need to import the WtLossCompetition class itself into index, but we first have to export the class. Lets use a different method:

```
    this.backupAdmin = supportAdmin;
  };
};
export { wtLossCompetition };
```

16. In the index.ts file, import the wtlosscompetition.js file NOT the ts file:

```
import { wtLossCompetition } from
"./WtLossCompttition.js";
const wtloss = new wtLossCompetition();
```

Note: it is difficult to work with module in the front end but bundlers like Webpack and Vite make this process a lot smoother. An advanced topic.

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 4 above. If you need a refresher on Generics, generally, check out Appendix D.

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 all the interface itself 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 *Competable*, we need to make this implementation generic also:

```
import iComp from "./iCompetable.js";
export class wtLossCompetition<T> implements
Competable<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){
```

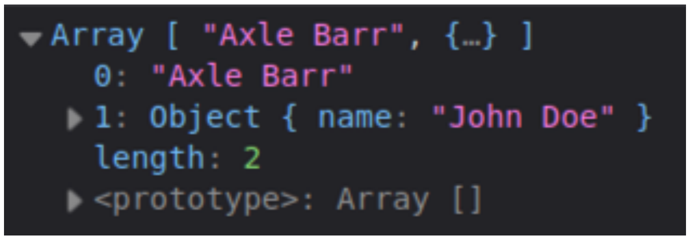
5. We would also need to change iCompetable since competitionDetails depend on that interface. In other words, it should specify a return of a generic array rather than a specific string:

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

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

```
    const wtloss = new wtLossCompetition();
    wtloss.addCompetitor("Axle Barr");
    wtloss.addCompetitor({name : "John Doe"});
    console.log(wtloss.competitonDetails());
```

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



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

8. Here are a few advanced topics for you to consider:
   a. Multiple Generic Types. For example we could have TCompetitor and Tadmin
   b. Types can be constrained so that they conform to a particular pattern, often called generic constraint:
   export interface Competable<T extends { id: string; name: string }> { competitors: T[];}
   In the example above, “extends” means assignable, NOT inheritance.

Part 6 – Decorators (Optional)

We will attempt to render an HTML tag on the browser using a *decorator* function…eventually. For this part, you can continue using the files from Part 5. Also for this part you need to manually turn on experimental decorators in the tsconfig file: `"experimentalDecorators": true`, if you are using a TS version less than 5.0.

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. Note, this function must be outside of the class definition.

4. You can create Decorator Factories by returning a new 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:

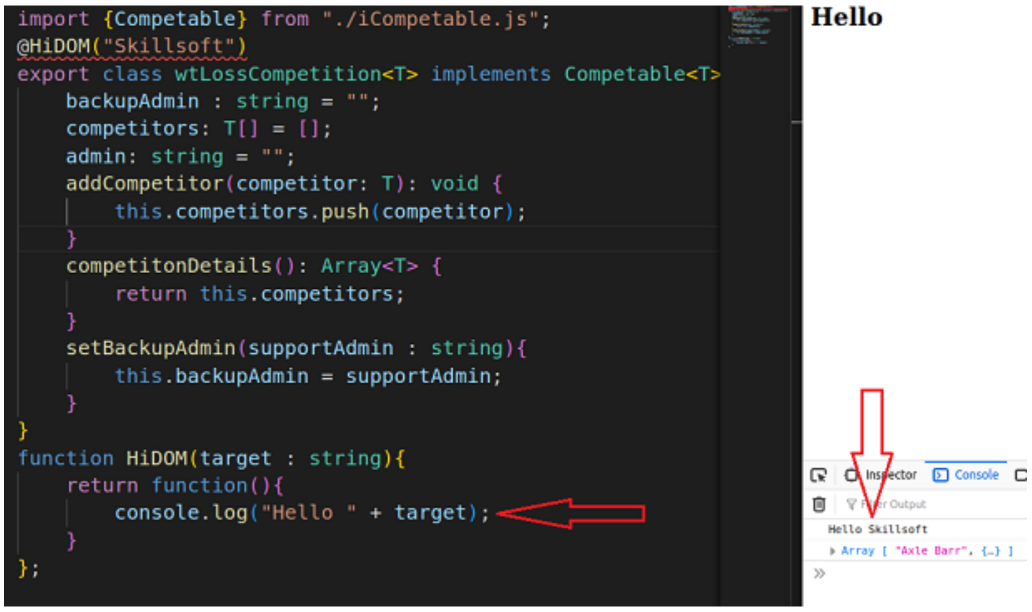
```
function HiDOM(target : string): Function {
    return function(){
        console.log("Hello " + target);
    }
};
```

Note, we are returning a function, so HiDOM() is a decorator factory.

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. 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(){
        const pageTag = document.getElementById(el);
        console.log("Hello " + msg);
    }
```

8. 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>
```

9. Now lets check and if an innerHTML property exists, we can use use it to pass our message into it:

```
function HiDOM(target : string, el : string) : Function {
  return function() {
    const pageTag = document.getElementById(el);
    console.log("Hello " + target);
    if (pageTag) {
      pageTag.innerHTML = "Hello " + target;
    } else {
      console.log("Element with id '" + el + "' not found");
    }
  };
```

   Refresh, you should see *Hello World* in the browser.

10. 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>{
```

11. Here is the final function body:

```
function HiDOM(target : string, el : string) :
Function {
  return function() {
    const pageTag = document.getElementById(el);
    if (pageTag) {
      pageTag.innerHTML = target;
    } else {
      console.log("Element with id '" + el + "' not
found");
    }
  };
};
```

Note: there is a flaw in this code, see below for the corrected code.

12 In TypeScript, class decorators expect to receive the constructor function as a parameter. This allows the decorator function to receive the **class** constructor, which is required for proper TypeScript class decorators. Here is the corrected code:

```
function HiDOM(msg : string, el : string) : Function {
  return function(constructor : any) {
    const pageTag = document.getElementById(el);
    const myCompete = new constructor();
    if (pageTag) {
      pageTag.innerHTML = "Hello " + el;
    } else {
      console.log("Element with id '" + el + "' not
found");
    }
    console.log("Hello " + msg);
  }
};
```

The function requires the constructor of type any in order to work properly. Then within the function apply the new operator to the constructor. This looks like it does nothing but it part of the specification.

13 Here is a an even better rendition of the function above:

```
function HiDOM(msg: string, el: string):
ClassDecorator {
  return function (constructor: Function) {
    document.addEventListener("DOMContentLoaded", ()
=> {
      const pageTag = document.getElementById(el);
      if (pageTag) {
        pageTag.innerHTML = msg;
      } else {
        console.warn(`Element with id '${el}' not
found`);
      }
      console.log(`HiDOM: ${msg}`);
    });
  };
}
```

Appendix A – Webpack

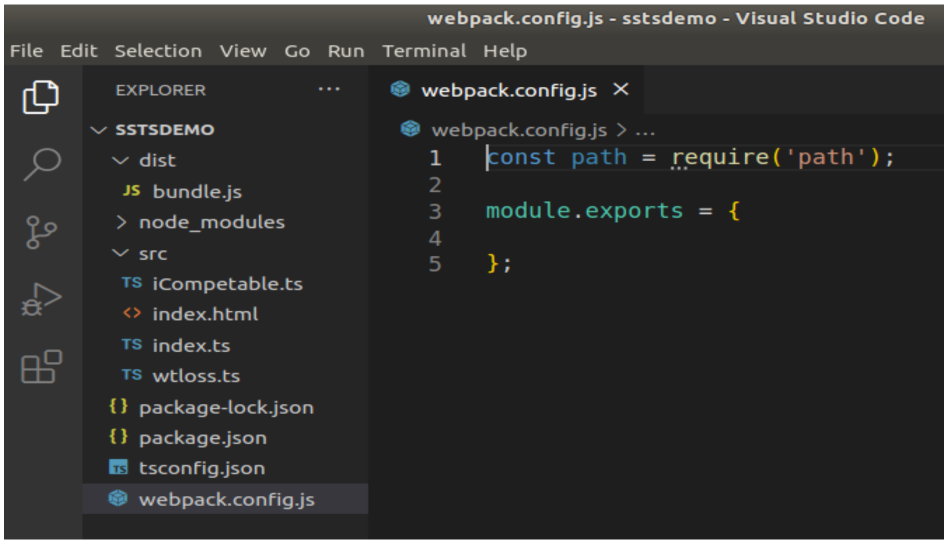
Webpack will bundle all our .ts and .js files into **just one** optimized and minimized output file. This will help with the number of http requests that the browser has to do in order to build a final view.

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);
```

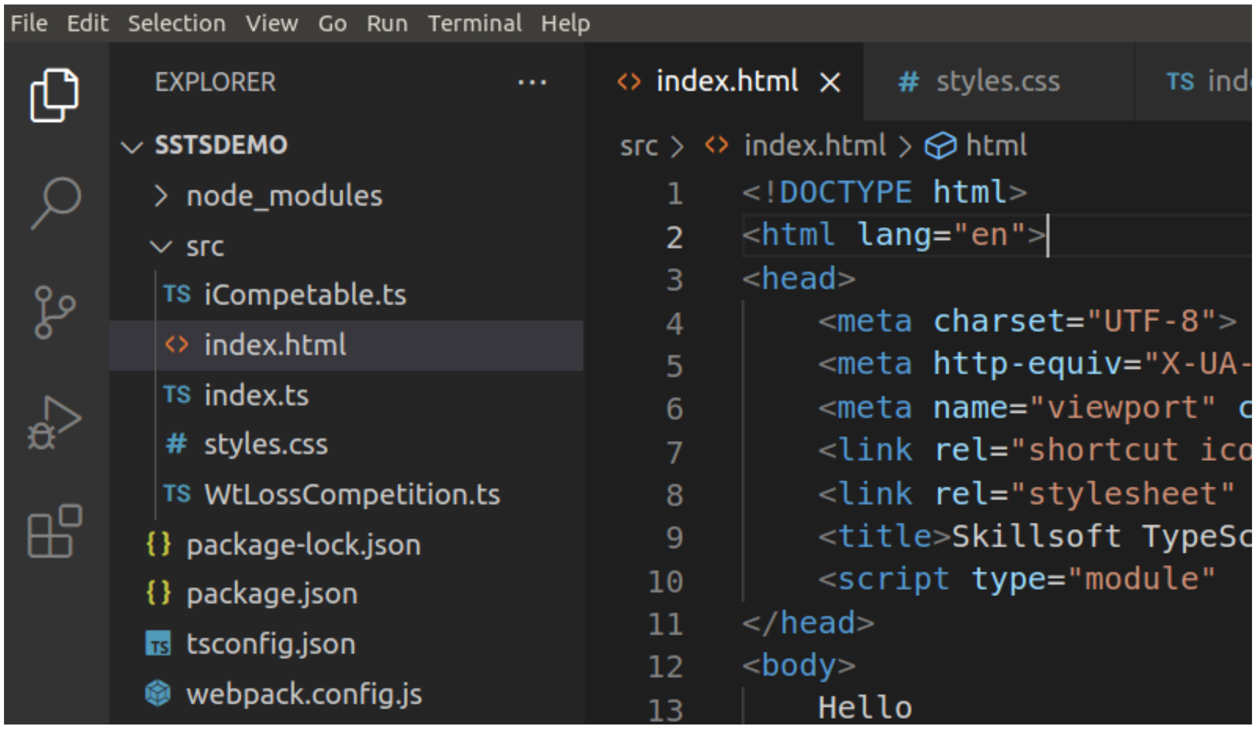
15. (Optional) If you wanted to see a more interesting example just add a .css file to the src folder. It can for example add a style to the “myDiv” div element.

```
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 : Generics in TS – A Primer

Imagine a "Smart" Container that has a label on it, but its blank. When you decide what you're putting into the box, you fill in the placeholder. For example we could write "FRUITS" on the placeholder. Now it's a "Smart Container of FRUITS." You can only put fruits in, and you're guaranteed to get fruits out. Later, you can use the same type of Smart Container but write "BOOKS" on the placeholder. Now it's a "Smart Container of BOOKS." You can only put books in the container now, no fruits, and you're guaranteed to get books out. The idea is that you make a decision at a point in time. Then at a different time, a different decision can be made.

The Problem Generics Solve (Without Generics)

Let's say you want to write a function that takes *anything* and simply returns that *same thing* (an "identity" function).

```
// without generics, the function is stuck, its not flexible
function identity(p) {
   return p;
}

const num = identity(123); // num is 'any' if TypeScript is used without specific types
const text = identity("hello"); // text is 'any'
```

The problem is that TypeScript loses the specific type information. It doesn't know 'num' is actually a number, or 'text' is a string. So it can't warn you if you try to do something wrong. For example if we write:

```
  num.toUpperCase();
```

You wont get an error from TypeScript here, but program would crash at runtime!

With generics:

```
function identity<T>(arg: T): T {

  return arg;

}
```

Here the 'T' is just a placeholder, it could have been ‘P’ or ‘Peter’. It is our type variable like the placeholder on the "Smart Container".

Now at the moment in time when this identity() function is called, we can decide the actual type that T represents, at that point in time.

For example:

const num = identity<number>(123); // Here, T becomes 'number'. `num` is inferred as 'number'.
const text = identity<string>("hello"); // Here, T becomes 'string'. `text` is inferred as 'string'.

Notice that <number> and <string> are called generic parameters and sometimes can be omitted.

For example:

const booleanValue = identity(true);

Here TS knows right awaw that the type we are working with is Boolean. However if you wanted to be complete you could write this statement like this:

const booleanValue = identity<boolean> (true);