Introduction to TypeScript

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.

Using Node.js as a platform, 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 – Environment and IDE

1. Install TypeScript compiler using this command:

npm install -g typescript

1. Test the install using this command:  
   tsc –version

Text

Description automatically generated

1. Graphical user interface, text, chat or text message

   Description automatically generatedIn your OS choose a place where you want to create a working directory, in my case I am using the Documents folder where I have a Demos folder and there I will create a new folder called sstsdemo:  
   Note: the command code . will invoke the **VSCode** editor at that location wherever it happens to be (the . is the period key on the keyboard)
2. Use the editor to create a new file called ss.ts
3. Once you have a .ts file, add the following code so that we can test that everything is working:  
   let greeting: string = 'Hello from Skillsoft!';

console.log(greeting);

export {};

1. If you enter the ls command at the terminal window, you will see your ss.ts file there. Now if we compile this file to JavaScript we need the **tsc** command, so enter this command:  
   tsc ss.ts
2. Now if you run the ls command again you will see the JavaScript file sitting there. We can compare both files in VSCode:

Text

Description automatically generated

Note: we cannot execute the file as yet, JavaScript runs in an environment like a browser or in NodeJS

1. Create a tsconfig.json file in the terminal window, by running the command   
   npx tsc -- init or just create one manually. We will use the command to create this config file and then remove the items we do not want. Note: you could use the Text

   Description automatically generatedterminal window in VSCode.
2. Lets set up two directories a src folder and a dist folder as required by the *tsconfig* file. Do this in the **root** of your project.
3. Around line 50 of the tsconfig.json file there is an *outDir* setting, remove the comments and add in an *out* directory:  
    "outDir": "./dist",
4. We will also configure the input folder, so around line 29 there should be a *rootDir* setting:

"rootDir": "./src",

Note, you may get an error from the IDE, ignore it for now

1. Here is a sample file:

|  |
| --- |
| {  "compilerOptions": {  "target": "es2016",  "module": "commonjs",  "esModuleInterop": true,  "forceConsistentCasingInFileNames": true,  "strict": true,  "skipLibCheck": true,  "outDir":"./dist",  "rootDir": "./src"  }  } |

Note: we will be adjusting the *commonjs* setting in a future section

1. A screenshot of a computer

   Description automatically generated with medium confidencethis is my setup showing the folder structure and the tsconfig.json file without all the commented out lines:
2. To do a test, move the ss.ts file into the src folder, also delete the ss.js file.
3. Run the command again, so run tsc once you delete the original ss.js file you will see that the deleted .js file is replaced in the dist folder once the command is run. NOTE: do not pass the ss.ts file to the command like this tsc src/ss.ts

This is my final setup on a Linux OS

Text

Description automatically generated

# Part 2 – HTML Setup

1. In the src folder add a new **html** file. You can do this from the IDE, just right click in the src folder and choose the *New File* option. Name the file index.html.
2. Put your cursor inside the index.html file and in the blank editor window start typing *html*, the IDE will respond with three options, choose *html:5* and an HTML template will appear in the file, save it. You can change the *title* if you want.
3. Add the following line just before the <title> tags:  
   <script src="index.js"></script>
4. Delete the ss.ts and ss.js files from their respective folders and create a new index.ts file in the src folder.
5. If you get an error in the *tsconfig* file, just close the project folder from VSCode and re-open the project folder. You must do this from the IDE, so go to *File -> Close Folder*. Once the IDE closes the project, you may open it again using the shortcut or the *File* menu.
6. Graphical user interface, text

   Description automatically generatedIn VSCode, click on the **Extensions** icon and in the search bar search for *Live Server*:
7. Choose the server by *Ritwick Dey* and install it using the blue install button. Click on the *Files* icon in the IDE once the server has been installed, it’s the icon.
8. Once you are absolutely sure that **Live Server** is installed, right click on the index.html file and choose **Open with Live Server**. Your default browser (Firefox in my case) should open showing the blank rendered contents of the index.html file we created earlier. Note, nothing will show right now in the main window, but you should be able to see the *title* of the page in the browser. We set the title in step 2 above. Also if you look at the bottom right of your IDE you should see some feedback from the server, it should tell what port the server is using. Remember, you must right click on the index file from within the IDE, **not** from your OS file system.
9. If you now position both the IDE and the browser next to each other you will see that the browser responds immediately to any change in the HTMl file. For example if we type *Hello* into the <body> tags, the browser responds, even without saving the HTML file.
10. Open the index.ts file in the IDE and add a console.log() line like this:

Text

Description automatically generatedconsole.log(“Hello”);

Do not save the file, but now in the browser hit the **F12** key to view the developers tools at the bottom of the browser. Make sure the *Console* tab is opened. There may be an error there but the console log message is not displayed.

1. Now back in the IDE, at the bottom left there is a gear icon, click it and choose the Settings option from the pop-up menu.
2. A screenshot of a computer

   Description automatically generatedIn the setting search bar, type in *auto save*, choose *afterDelay* and change the next setting to about 800, which is milliseconds

Close the window, the settings are saved.

1. Now lets setup a *Node.js* project, we will NOT be using Node, but this gives us a package.json file which is like a config file. So in the terminal run:  
   npm init -y
2. Since we setup the project to look in the dist folder for the index.js file, we need to add the browser path to reflect this, so add the <script> tags between the <head> tags in the browser setting, make sure your path is pointing to the right file:  
   <script src="../dist/index.js"></script>
3. Now run the tsc command from the terminal window and a new index.js file should be generated inside of the dist folder and the browser should refresh and now show *Hello* in the *Console* window in the developers area at the bottom.

Text

Description automatically generated

1. (Optional) The last thing to do in this section is to run the tsc command in *watch* mode, in this way anytime we make a change to the .ts file, the change will be affected immediately. To do this run the command: tsc -w
2. If you did the above, try changing some content that is being logged in the console window and see if the change happens right away. We configured the delay in step 13 above.

Text

Description automatically generatedText

Description automatically generated

# Part 3 – Working with Types

1. Start this section with this basic code of an employee object:

|  |
| --- |
| const employee = {  empName:"Axle",  dependents:2,  }  console.log(employee) |

Just replace the code you have in the index.ts file with what you see here.

1. If you hover over the object’s name, so *employee* you will see that TS has correctly inferred that this object is of type string and number:

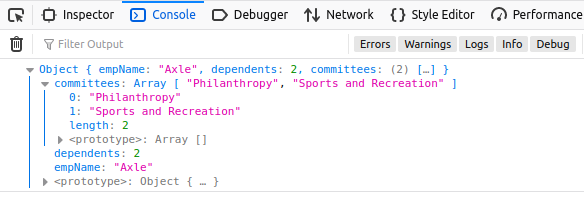
Graphical user interface, text

Description automatically generated

1. Lets add an array to the employee object:

|  |
| --- |
| const employee = {  empName:"Axle",  dependents:2,  committees:["Philanthropy", true]  };  console.log(employee); |

This is possible in TS, we can have an array of string and Boolean and if you hover over the *committees* key, you will see this inference being made by the TS compiler.

1. The problem with the above code is that we can now attempt to add a string into position 1 (zero based) of the *committees* array, and TS will just accept it:  
   employee.committees[1] = "Sports and Recreation";  
     
   

This may or may not be the desired output, that second element is reserved for a Boolean value NOT a string!

1. The solution is to enforce the type checking that TS gives us, it is a feature of TS. In other words, we need a *tuple* and this is the way to declare a tuple:

|  |
| --- |
| const employee : {  empName : string,  dependents : number,  committees : [string, boolean]  } = {  empName:"Axle",  dependents:2,  committees:["Philanthropy", true]  }  employee.committees[1] = "Sports and Recreation";//error  console.log(employee); |

Normally TS would infer the object, but in this case we want to define to TS what our object should look like. We also want to tell TS that the *committees* array, now **tuple**, is made up of a string and Boolean in that order. This means that if we try to insert a string into position 1, the IDE shows an error. Please note that the *push()* method of arrays and tuples will actually work here, it overrides this check, but that’s an advanced topid.

1. TS also supports the *enum* structure:

|  |
| --- |
| enum weekDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};  const employee : {  empName : string,  dependents : number,  committees : [string, boolean],  payDay : number;  } = {  empName:"Axle",  dependents:2,  committees:["Philanthropy", true],  payDay : weekDay.FRIDAY  } |

Notice that in the definition of employee, payday is a number but in the initialization it is a name or string. Once an enum is created it is given a numeric system behind the scenes. We can force a string, see below.

1. We can force the enum to work with strings by simply redeclaring each value:

|  |
| --- |
| enum weekDay {  MONDAY = "MONDAY",  TUESDAY = "TUESDAY",  WEDNESDAY = "WEDNESDAY",  THURSDAY = "THURSDAY",  FRIDAY = "FRIDAY"  };  const employee : {  empName : string,  dependents : number,  committees : [string, boolean],  payDay : string;  } = {  … |

# Part 4 – Literal Types and Aliases

Continue using the same files from Part 3, 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 4 we start with a blank .ts file.

1. If you hover the mouse over each of these statements, the first will show string but the second will show **Med**. The second is a *literal* type in TS.

|  |
| --- |
| let size = "Med";  const SIZE = "Med"; |

1. Literal types are excellent for use with functions. Here is a function that takes two values one string the other number, and it **returns** either a string or a numeric type:

|  |
| --- |
| function compareWords(  a : string, b : number) : "" | 0 {  if(…)  return "";  else  return 0;  } |

1. Literal types can be returned from functions.

|  |
| --- |
| function compareWords(  first : string, second : string) : "first is earlier" | "same" | "first is later" {  if(first < second)  return "first is earlier";  if(first > second) return "first is later";  return "same";  }  console.log(compareWords("Axle", "Axle")); |

This function will take two strings, compare them and return one of three and **only one** of three values. Note: this function can be re-written like this and can be found on the official docs for TS:  
function compare(a: string, b: string): -1 | 0 | 1 {

return a === b ? 0 : a > b ? 1 : -1;

}

1. To demonstrate another example using union types, enter this function into the IDE and run the program:

|  |
| --- |
| function irCalculator (p:number, r:number) {  return p \* r;  };  //  let investment = irCalculator(1000, 0.10);  console.log("Interest is " + investment); |

1. We could make the code from above more flexible. If we wanted to use the result in a calculation we could return a numeric value but we have the option of returning a string value, perhaps to add to some other string:

|  |
| --- |
| function irCalculator  (p:number,  r:number,  returnValue : "returnString" | "returnNumber") {  return p \* r;  … |

Here we are adding a third parameter which accepts how the user wants the result of this function, either as a *string* or as a *numeric* value

1. In this case we would have to add some more logic to the function

|  |
| --- |
| function irCalculator  (p:number, r:number, returnValue : "returnString" | "returnNumber") {  if(returnValue === "returnString")  return p \* r;  else  return +p \* +r;  }; |

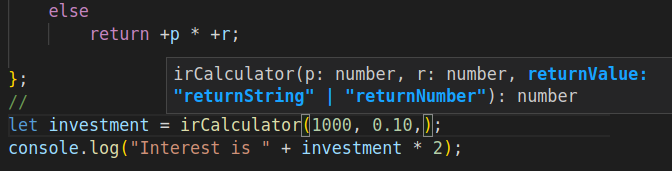
In the *if* clause, no need to explicitly return a string, but in the *else* clause we force the return type to be of type numeric. We can perform arithmetic operations on that result.

1. To call the function, you must now pass three parameters, and you can do math on the return:

|  |
| --- |
| let investment = irCalculator(1000, 0.10, "returnNumber");  console.log("Interest is " + (investment \* 2)); |

1. **Aliases** work just like types, we could declare one or more union primitive types using an alias, then replace the individual union with the alias:

|  |
| --- |
| type returnable = "returnString" | "returnNumber";  function irCalculator  (p:number, r:number, returnValue : returnable) {  if(returnValue === "returnString")  return p \* r;  … |



# Part 5 –Function Type and Callback Functions

Continue using the same files from Part 4, 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 5 we start with a blank .ts file.

1. Function literal and callback functions work the same as in JS. Lets first define what a function should look like, so what parameters it can accept and what type of return value it produces:

|  |
| --- |
| let interestCalc = function iCalc(x:number, y:number){  return x \* y;  }; |

Here we are saying that *interestCal* represents a function that takes two numeric types and returns a numeric type.

1. The above can be written like this in order to be more compact

|  |
| --- |
| let iCalc : (x:number, y:number) => number; |

Notice that the colon separates the function definition which in itself is written like an arrow function.

1. We could then have a function that matches the signature from #2

|  |
| --- |
| function irCalculator(p:number, r:number) {  return p \* r;  } |

The *irCalculator()* function takes two numbers and returns a number, just like *iCalc*.

1. It means then that we could assign the concrete function to our *iCalc* function type:

|  |
| --- |
| iCalc = irCalculator; |

1. If we now call *iCalc* and pass it the required parameters, we should see a numeric output

|  |
| --- |
| console.log( iCalc(1000, 0.10) ); |

1. In a similar way, I can now define a new function that takes a function as one of its parameters (callback functions)

|  |
| --- |
| function printInterest(p:number, r:number, **fIntCalc:Function**){  return(fIntCalc(p, r));  } |

Notice the third parameter here is a function called fIntCalc or interest rate calculator

1. We can now call this function, pass it the two values from #5 then as the third parameter pass *irCalculator* and then we can print the result of all these changes:

|  |
| --- |
| console.log(printInterest(1000, 0.10, **irCalculator**)); |

This should now pass *irCalculator()* from #3 above, into the *printInterest()* function, the calculation is done first then the value is returned to the *console.log()* function.

1. Here is the entire program to this point:

|  |
| --- |
| let iCalc : (x:number, y:number) => number;  //  function irCalculator(p:number, r:number) {  return p \* r;  }  //  iCalc = irCalculator;  //  function printInterest(p:number, r:number, fIntCalc:Function){  return(fIntCalc(p, r));  }  //  console.log(printInterest(1000, 0.10, irCalculator)); |

You could pass either irCalculator OR iCalc, they both point to the same address in memory:

# Part 6 – Chaining and Null Coalescing

Continue using the same files from Part 5, 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 6 we start with a blank .ts file.

1. If you visit the **jsonplaceholder** site (https://jsonplaceholder.typicode.com/posts) and try to access the posts data, you see data that is wrapped in this format:

|  |
| --- |
| userId : 1  id : 1  title : “”  body : “” |

1. We could now define a type based on this data:

|  |
| --- |
| type post = {  userId : number;  postId : number;  title : string;  body : string;  }; |

1. Lets now create an actual post for demonstration and log the details or part of the details

|  |
| --- |
| const myPost : post = {  userId : 100,  postId : 3842,  title : "TypeScript Rocks",  body : {topic:"Decorators", explanation:"Adds functionality to functions"}  };  //  console.log(myPost.body); |

Note: this could be in a loop

Also change the **post** -> **body** to { topic : string, explanation : string }

1. This is the index.ts file so far:

|  |
| --- |
| type post = {  userId : number;  postId : number;  title : string;  body : { topic : string, explanation : string };  };  const myPost : post = {  userId : 100,  postId : 3842,  title : "TypeScript Rocks",  body : {topic:"Decorators", explanation:"Adds functionality to functions"}  };  console.log(myPost.body); |

1. But there could be a problem, this is data that we do not control, what if there is no *topic* or *explanation* being returned? One JS solution is to do something like this:

|  |
| --- |
| if(myPost.body && myPost.body.topic)  console.log(myPost.body.topic); |

With this code, if *body* exists, then JS will continue on to check *topic*. However if *body* does NOT exist, topic is **never** checked. This code works in TS and in JS and is called short circuiting.

1. You can test the above code by passing an empty string in the body.topic variable:

|  |
| --- |
| title : string;  body : { topic : any, explanation : string };  …  } title : "TypeScript Rocks",  body : {  topic:"",  explanation:"Adds functionality to functions"  } |

Change the topic type to be *any* and add an empty string to *topic* in the body section. This will not print anything for *body.topic*.

1. But TS has a better, shorter solution. Its called **Optional Chaining**.

|  |
| --- |
| body : {topic:"Decorators", explanation:"Adds functionality to functions"}  };  //  if(myPost.body?.topic)  console.log(myPost.body.topic); |

Much shorter, but we still do not get anything being printed for *myPost.body.topic.*

1. Another situation may be that you want to store the data you are getting from an outside soure:

|  |
| --- |
| let posts : string[] = [];  posts.push(myPost.body.topic); console.log(posts); |

Here we create an array of posts and attempt to push each topic into the array. In this case we get an empty string.

1. Since we expect that the topic could be *null* or *undefined*. Well we could add a default value or a fail-safe value like this:

|  |
| --- |
| let posts : string[] = [];  posts.push(myPost.body.topic || "No Topic"); |

Here if we do not get a value for topic, we simply add a **default value** to the array at that location.

1. Lets do a test, passs an empty string:

|  |
| --- |
| title : "TypeScript Rocks",  body : {topic:"", explanation:"Adds functionality to functions"}  };  //  let posts : string[] = [];  posts.push(myPost.body.topic || "No Topic");  console.log(posts); |

It prints “No Topic” because topic is now treated as *falsy*. Of course this may be exactly what we want.

1. There are times however when we only want to eliminate **undefined** or **null values**. If that is the case we must use the **nullish coalescing operator** in TS

|  |
| --- |
| title : "TypeScript Rocks",  body : {topic:"", explanation:"Adds functionality to functions"}  };  //  let posts : string[] = [];  posts.push(myPost.body.topic ?? "No Topic");  console.log(posts); |

Now it will print an empty string in the space where this element is loaded into the array. In other words, it wont print “No Topic”.

1. If you now explicitly set topic as *null* or *undefined*, you get the “No Topic” default string being stored:

|  |
| --- |
| title : "TypeScript Rocks",  body : {topic:undefined, explanation:"Adds functionality to functions"}  };  //  let posts : string[] = [];  posts.push(myPost.body.topic ?? "No Topic");  console.log(posts); |

If you tried this, you would need to change the definition of *body* in *type post* to something like this: body : {topic:any, explanation:string };

1. Here is the entire code for this section:

|  |
| --- |
| type post = {  userId : number;  postId : number;  title : string;  body : any;  };  //  const myPost = {  userId : 100,  postId : 3842,  title : "TypeScript Rocks",  body : {  topic:"",  explanation:"Adds functionality to functions"  }  };  //  let posts : string[] = [];  posts.push(myPost.body.topic || "No Topic");  console.log(posts); |

# Part 7 – 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(); |

1. 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.

1. 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 2023");  //competition.addCompetitor(); |

1. 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.

1. 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.

1. 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() { |

1. 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;  } |

1. 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 2022", 100);  console.log(competition.competitonDetails()); |

1. Text

   Description automatically generatedAt this point, below shows the code so far and the output as it appears in the Console window of my Firefox browser
2. 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()**

1. 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 8 – 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.

1. 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{    } |

1. 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();  }  } |

1. 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;  }  } |

1. 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);  }  } |

1. (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.

1. 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.

Appendix A : Favicon Error

To get rid of the favicon error add the following to the <head> tag of the browser:  
<link rel="shortcut icon" href="#">

This will also work:  
<link rel="icon" href="data:,">

Appendix B : Topics

Part 1 – Environment and IDE

Part 2 – HTML Setup

Part 3 – Working with Types

Part 4 – Literal Types and Aliases

Part 5 –Function Type and Callback Functions

Part 6 – Chaining and Null Coalescing

Part 7 – Classes

Part 8 – Interfaces

Appendix C : Optional Config Settings

1. removeComments : true

This setting will allow you to enter *comments* in the .tsc file but once compiled into a .js file, the comments will be removed

1. noEmitOnError : true

In order to show what this setting does, lets introduce an error:

Text

Description automatically generated

The code has an error, we are trying to assign an unknown variable type to a number type. The IDE shows the error but the code still compiled, notice the red arrow in the browser. If we change or add the *noEmitOnError* and set it to *true*, this compilation will NOT happen. The default is *false*.

1. You can set the following three at all at once:  
   noUnusedLocals : false  
   noUnusedParameters: false

noImplicitReturns : false

With this in place if you have functions in your code and you create variables but don’t use them, the IDE will complain. Also if you declare functions with parameter(s) and don’t supply them it will complain. If you create functions that does now always return a value it will also complain. This happens when you have **if** statements but don’t provide a *false* part.

Appendix D : Uncaught Reference Error

If you get this error in the browser:  
*Uncaught ReferenceError: exports is not defined*

Make sure your *"module": "commonjs"* line in the tsconfig.js file is commented out. Also make sure your target is **es2016** or above.

Finally remove the line *export {}* from your index.ts file if you have it.

Appendix E : Function Parameter in JS

"use strict";

const y = 9;

const x = function (n1, n2) {

return n1 + n2;

};

//

function addem(z, y, x){

return x(z, y);

}

let result = addem(1,2,x)

console.log(result);