1. Installing + Setting up the editor.  
     
   npm install -g typescript  
   tsc .\usingts.ts  
   Extensions : ESLint : code quality check support.

Material icon theme.  
Path Intellisense : better support when we work with imports.  
Prettier - Code formatter

1. npm install --save-dev lite-server : server index.html file [something like live-server]
2. core types : javascript knows and typescript(lowercase) also supports:  
   number : +ve , -ve, floats  
   string : “abc”, `${template literals}`

boolean : true / false  
…objects and so on.

const add = (n1: number,n2: number) => {  
 return n1+n2;  
}  
const number1 = 5;  
const number2 = 2.8;  
add(number1,number2);

1. object types.  
     
   const person : {name:string;age:number} = {  
    name:"steve",  
    age:33  
   }  
   console.log(person.name)
2. Nested objects and types.  
     
   const person : {name:string;age:number;address:{ line1:string;line2:string;pin:number} } = {  
    name:"steve",  
    age:33  
   }
3. Array – can have arrays with strings and numbers mixed.

Types of the array can be flexible or strict.  
  
const person:{name:string;age:number;hobbies:string[]}= {

name:"steve",

age:33,

hobbies : ['sports','cooking']

}

for(const hoby of person.hobbies){

console.log(hoby.toUpperCase())

//typescript automatically detects that this is a string

}

person.hobbies.//map/pop/push/reduce... etc all will be listed automatically

1. Working with Tuples – only with TS  
   [1,2] – fixed length array + fixed type array  
     
   const person: { name:string; age:number; hobbies:string[]; role:[number,string] }= {   
   name:"steve",  
   age:33,  
   hobbies : ['sports','cooking'],  
   role:[2,'dev-eng'] // WHY ? here we wanted to have an array with exactly to elements  
   }

person.role.push(22);

person.role.push('admin'); // we only need two elements - but still TS does allow this - size can not controlled by typescript

// person.role = [1,'simple-role','another-fake-entry'] - when assigning it this way Ts complains about the third entry

person.role[1] = "test-eng"; // can switch values

// person.role[1] = 20; //this will be complained by typescript - the order of the types is strict.

**// So if you have scenario that there should be exactly only two elements in an array**

**// + And you know the type of each element in advance : TUPLE is the perfect solution**

1. Working with Enums (custom type in typescript): There might global constants – which are represented as numbers, but for which you want to assign a label.

**WITHOUT ENUMS : with constants**  
const ADMIN = 0;

const READ\_ONLY\_USER = 1;

const AUTHOR = 2;

const person = { name:"steve", role: READ\_ONLY\_USER }

if(person.role === READ\_ONLY\_USER){

console.log("The role is READ\_ONLY\_USER")

}

// The advantage of this is instead of using the numbers we use strings which are more readable

// The downside is ANY number can be stored + v have lot of constants and we have to manage them

role: 100  
  
**WITH ENUMS**enum Roles {

ADMIN,READ\_ONLY\_USER,AUTHOR

};

const person = { name:"steve", role: Roles.READ\_ONLY\_USER }

if(person.role === Roles.READ\_ONLY\_USER)

console.log("The role is READ\_ONLY\_USER")

1. **Union types(TS)**

const combine = (n1: number|string ,n2: number|string ) => {

if(typeof n1 === 'number' && typeof n2 === 'number')

{

return n1+n2;

}

else{

return n1.toString() + n2.toString();

}

}

console.log(combine(2,3))

console.log(combine('steve','jobs'))

//Thus we can use union types to be flexible regarding - what we do in a function.

1. **Literal types** : exact value it holds (based on core types) – used in conjunction with union types.  
     
   const combine = (n1: number|string ,n2: number|string, typeOfConversion : 'as-number' | 'as-string') => {

if(typeof n1 === 'number' && typeof n2 === 'number' || typeOfConversion === 'as-number')

{

return +n1 + +n2;

}

else{

return n1.toString() + n2.toString();

}

}

**console.log(combine('2','3','as-number')) // can’t use incorrectly.**

console.log(combine(2,'3','as-string'))

1. **Type Aliases / Custom Types.**

Why ? : No need to write “number|string” etc fully in multiple places.  
 : write code quicker with descriptive type aliases  
  
Example 1:   
**type combinable** = number | string;

**type conversionDescriptor** = 'as-number' | 'as-string';

const combine = (n1: **combinable** ,n2: **combinable**, typeOfConversion : **conversionDescriptor** ) => {

if(typeof n1 === 'number' && typeof n2 === 'number' || typeOfConversion === 'as-number')

{

return +n1 + +n2;

}

else{

return n1.toString() + n2.toString();

}

}

console.log(combine('2','3','as-number'))

console.log(combine(2,'3','as-string'))

Example 2 :   
**type User = {**

**name:string;**

**age:number**

**}**

const displayUser= (**emp : User**)=>{

console.log(emp.name + " is " + emp.age + " years old ")

}

displayUser(**{name:"steve",age:23}**)

1. Function return types and void(TS) + undefined.  
     
    const add = (n1:number,n2:number) : number=>{

return n1+n1;

}

//below does not return anything

const printResult = (n1:number) :**void**=>{

console.log(n1);

}

console.log(printResult(5));

//Above function returns undefined - this doesnot mean that we can return undefined explicitly

//and so the below call will fail.

// const printResult2 = (n1:number) : **undefined**=>{

// console.log(n1);

// }

//the point is you should void if a function returns nothing - this fn does not have a return statement.

const printResult3 = (n1:number) :**void**=>{

console.log(n1);

}

//then if you explicitly say that a fn returns undefined.

// - it means to typescript that - we have a return statement + we don't return a value.

const printResult4 = (n1:number) :**undefined**=>{

console.log(n1);

**return**;

}

//another valid case with return + void

const printResult5 = (n1:number) :**void**=>{

console.log(n1);

**return**;

}

//Note that : undefined is a valid type in javascript

const temp = **undefined**;

//the above statement will NOT throw any error.

1. Functions as types : describes the parameters and return values of the function  
   **Why ? :**   
     
   const add = (n1:number,n2:number) : number=>{

return n1+n2

}

const printResult = (n1:number) :void=>{

console.log('printing ' + n1)

}

let addRef:(a:number,b:number)=>number;

addRef= add;

//now typescript will complain

addRef = printResult;

//by adding "let addRef:Function;" below statements can be caught

//addRef = 5;

//But the point is we did not say which kind of function.

//when we say : addRef = printResult;

//we are storing the wrong function in there and typescript is not complaining about that

// we need too be presise about how the function should look like

console.log(addRef(5,2));

1. **Function types and callbacks.**//By mentioning void in the callback here - we are essentially saying we will ignore

// any result you might be returning in the callback you pass

// the below call back type does NOT force you to pass a 'callback that does not return anything'

// - but 'it says that it does not care whatever the callback you pass returns'

const addAndPrint = (n1:number,n2:number,cb:(n:number)=>void)=>{

cb(n1+n2);

}

addAndPrint(2,4,(nArg)=>{

console.log(nArg);

return "hurray";

//Typescript does not pick this error - even though void is mentioned in the parameter above

})

1. Unknown type.  
     
   let userInput : unknown ;

userInput = 44;

userInput = {

name:"11",

age:22

}

//so far : unknown is similar to any typescript type

let username:string = "Mac";

//username = userInput;

// the above will not be allowed because the type in userInput is not known.

if(typeof userInput === 'string'){

username = userInput;

}

1. **Never type** : another type functions can return.  
   Why ?  
     
   let userInput : unknown ;

userInput = 44;

userInput = {

name:"11",

age:22

}

//so far : unknown is similar to any typescript type

let username:string = "Mac";

//username = userInput;

// the above will not be allowed because the type in userInput is not known.

if(typeof userInput === 'string'){

username = userInput;

}

//This function never returns anything - not even undefined.

const generateError = (msg:string,errCode:number):never => {

throw {

message:msg,

errorCode:errCode

}

}

**----------------- TYPE SCRIPT COMPILER ------------------  
1.** Using watch mode. **>** tsc app.ts –w / --watch

**2.** If more than 1 ts files.

> Navigate to the folder where your typescript files are present.  
 > tsc –init  
 > tsconfig.json file is generated. Why ? It essentially tells typescript how it should compile the .ts files  
 > tsc  
 //the above command will convert all typescript files in the current folder to .js files.  
 // to run in watch mode for all files  
 > tsc –w / tsc –watch

3. How to include / exclude a file in the typescript compiler.  
 **exclude :** files you want to exclude from the ts compiler  
 **include :** Files you want to specifically include in the compilation process. Other files are excluded [basically we compile include minus exclude]  
 **files :** only individual files you want to compile, can not specify folders

{

"compilerOptions": {

/\* Basic Options \*/

/\* Strict Type-Checking Options \*/

/\* Module Resolution Options \*/

/\* Source Map Options \*/

/\* Experimental Options \*/

/\* Advanced Options \*/

},

"exclude": [ "node\_modules", "\*.analytics.ts"],

"include": ["\*.ts"],

"files": ["app.ts"]

}

4. -------------------- COMPILER OPTIONS ------------------------  
**"target":** "es5",   
/\* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019' or 'ESNEXT'. \*/  
For which target javascript version we want to compile the code.  
The compiler also compiles the code that runs in certain set of browsers.  
ES6 : ES2015   
Alternatively we can use other transpilers.

**module**   
Why ? This options allows us to specify which default objects and features typescript knows.  
For example : Typescript should complain for the below statement , but it recognizes document.  
*const button = document.querySelector('button');*  
And the same will not work in the nodejs environment.  
This is controlled by the **lib** option. When uncommented ts shows a lot errors

**allowJs**   
Why ? we can allow javascript files in the compilation process.  
 **checkJs**Typescript compiler only check but does not compile

**.d.ts** : This matters if we are shiiping our project as a library.

**sourceMap**Why ? helps us with debugging and development.

In the developer tools window we only see the javascript files, This options helps us see the ts files there also and thus in debugging them too.  
  
**outDir and rootDir**

We can tell the typescript compiler where to look for all the typescript files and where to put the generated js files.  
The folder structure in the rootDir will be replicated in the outDir.

Let us say if you have two ts directories rootDir1and rootDir2 **:** Typescript willcreate two such directories in the outDir unless **rootDir** is specifically mentioned,which means to ignore other directories where ts files are present.  
 **noEmit :** The typescript compiler checks the files and reports any potential errors, but does not generate the javascript files.

**noEmitOnError:** If any ts file fails to compile the js files wont be generated.

5. ------------------------STRICT OPTIONS--------------------------------  
 **noImplicitAny :** Thisensuresthe parameters type in mentioned, by default any will be taken and ts compiler will not allow this. This is not the case with variables ..for example you can say **let logged;** and later say **logged=true;**

Why ? functions are created first.  
 **strictNullChecks:** whenthis is false.The typescript compilerwill not complain in the below code.  
const button = document.querySelector('button'); [no need of the bang symbol]  
 **strictFunctionTypes** : Related to function types, not the types inside of functions. Bugs could be introduced if we work with classes and inheritance, we could introduce bugs.

**strictBindCallApply :** Usefulso that we don’t use bind call and apply accidentally in a way that does not work with our code.

**strictPropertyInitialization :** important when we work with classes.

**noImplicitThis :** has to do with the this keyword and TS basically tries to warn you when you use the this keyword in a place when it is not clear what it refers to.

5. -------------------- code quality options -------------------  
  
 noUnusedLocals : we don’t want unused local variable  
 noUnusedParameters : we don’t want to keep unused parameters.