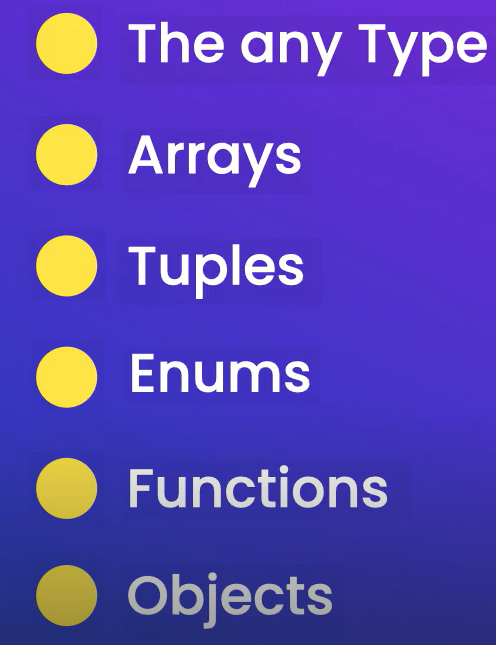


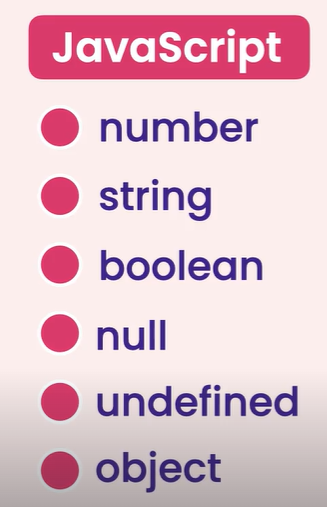
**Introduction**:

In this section, we will learn about:

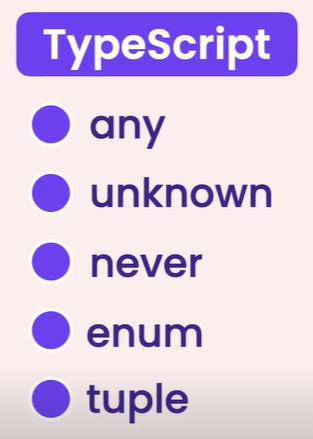


**Built-in Types**:

Like JavaScript has these built in types,



TypeScript extends this list and introduce types like,



Let’s see how can we play with primitive types in TS, by declaring a number using TS annotation.

let sales: number = 123456789;

Note: In TS if we have a large number we can separate digits using underscore \_ .

let sales: number = 123\_456\_789;

Now we can add a string and Boolean,

let course: string = "TypeScript";

let is\_published: boolean = true;

Let’s see something really cool, *In TS we don’t always have to annotate our variables because TS compiler can infer or detect the type of variables based on their value*.

So even if we remove the annotation,

 we can see that sales is a number.

We can do the same for the other two variables,

What if we declare a variable and don’t initialize to any value,

let level;

In this case TS assumes that this variable is of type *any*.



**The ‘any’ type**:

In TS we have a new type called *any* which can represent any kind of values. So if we declare a variable and don’t initialize it, the TS compiler assumes that this variable is of type *any*.

We can set it to a number and later on to a string.

let level;

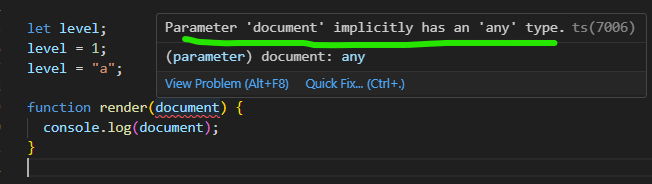
level = 1;

level = "a";

but this is against the whole idea of using TS, because we use it for type safety, so we get type checking. So if we use any type, we essentially lose that feature and the major benefit of using TS.

*As a best practice avoid using* ***any*** *type as much as possible*.

Let’s look at another example, say we have a function called render that takes a document and simply renders it on the console.



As we can see there is a compilation error which is saying parameter ‘document’ implicitly has an ‘any’ type.

*Implicitly means we haven’t explicitly or clearly set the type of this parameter, So compiler is inferring or guessing the type of this parameter*.

Let’s say this is part of a JS project which we are trying to convert to typescript and at this point, its impossible for us to explicitly annotate this with a particular type.

*Soution1*:

Annotate the parameter with any.

function render(document: any) {

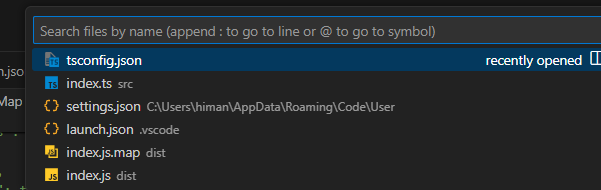
  console.log(document);

}

But what if we have tons of errors of this type, we don’t want to go to every function and explicitly annotate various parameters with *any* type.

*Solution2*: (*nuclear* option *not recommended*)

Use Ctrl+ P to open search option,



Go to tsconfig.json and notice in the type checking option,

    /\* Type Checking \*/

    "strict": true,                                      /\* Enable all strict type-checking options. \*/

*strict* is set to true. *This is equivalent of turning on some of the basic type checking features*.

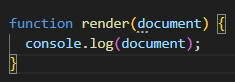
First setting we see is *noImplicitAny*, let’s uncomment it for now.

"noImplicitAny": true,                            /\* Enable error reporting for expressions and declarations with an implied 'any' type. \*/

If this feature is turned on, the compiler will complain about implicit *any* tags. Let’s turn this off.

 "noImplicitAny": false,

And in index.ts,

Error is gone now.

Note: Use this setting with caution only if you know what you are doing, otherwise there is no point of using TS.

So in the tsconfig.json. we will revert our code.

**Arrays**:

In JS we can declare arrays like this,

let numbers = [1, 2, 3];

Now the thing about JS arrays is that each element can be of different type.

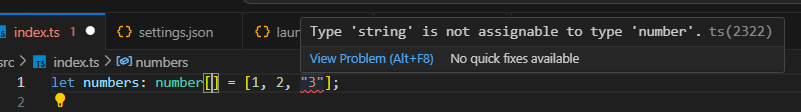
let numbers = [1, 2, "3"];

//*numbers and strings together-valid JS code*

Above code is valid JS code because JS arrays are dynamic so each element can be of different type.

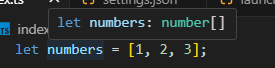
But what if we pass this array to a function that expects a list of numbers? Then third element is going to cause an issue.

This is where we use TS, so we can explicitly apply a type annotation here,

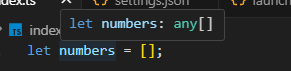


Saying numbers is a *number* array. So we can see the error immediately at compile time.

Note: After making all the elements in the array as numbers, we can remove type annotation and TS compiler can automatically guess the type of the array.



Let’s see what will happen if we keep an empty array.



Now the type of this variable is *any*, which is something we should avoid. In this type of array, we can keep a mix of different elements.

let numbers = [];

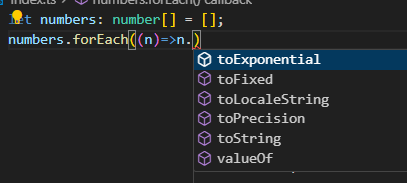
numbers[0] = 1;

numbers[1] = "b";

So if we want to use an empty array, we have to explicitly apply a type annotation here.

let numbers: number[] = [];

**Cool Trick**: Code completion / IntelliSense is another cool benefit of using TS. For example,



In this forEach method, since all the elements are numbers therefore when we put period after n, we can see all the available properties and methods on the number objects.

Since our editor knows the type of *n*, it offers code completion. Its very useful and offers great productivity boost.

**Tuples**:

TS has a new type called *tuple, which is a fixed length array where each element has a particular type. We often use them while working with a pair of values*.

For example, let’s say for each user, we want to represent two values, an ID and a name.

//1, 'Himanshu'

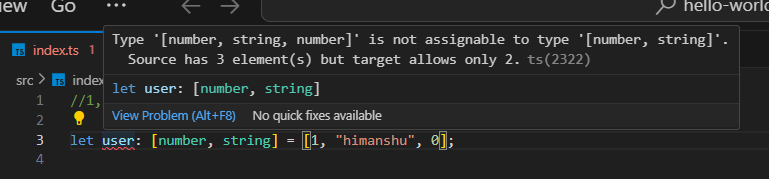
So we declare a variable and annotate it using a special syntax like this,

let user: [number, string] = [1, "himanshu"];

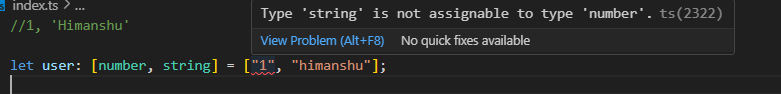
Here we tell the compiler that first element is going to be a number and second element is going to be a string and then we initialize our variables respectively.

So we have a fixed length array with exactly two elements, nothing more nothing less.

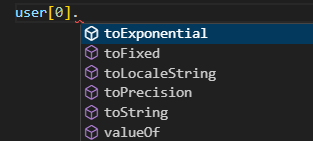
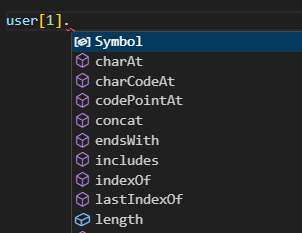
If we add a third element here, we get a compilation error,



According to this, First element has to be a number, if we change it to string



Just like before, here we also get IntelliSense / code completion,

The thing we need to know that tuples, internally are represented using plain JS arrays. So if we compile our code, we are just going to see a regular JS array.

So in the terminal, lets run tsc command, and in the index.js we can see,

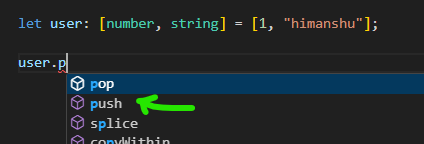
"use strict";

let user = [1, "himanshu"];

Just a regular JS array.

One of the GAP in TS:

If we see methods available in user array, there is a push method.



let user: [number, string] = [1, "himanshu"];

user.push(1);

There is no complaint from the compiler now, It should be.

Note: *As a best practice, restrict your tuples to only 2 values*, like key – value pairs, because anything more than that is going to make your code a bit harder to understand.

**Enums**:

TS has another built in type called *enum*, which *represents a list of related constants* (*like in C# or Java*).

Let’s say we want to represent size of T-Shirts, as constants.

One way to define these is like this,

const small = 1;

const medium = 2;

const large = 3;

Another way is to group these constants inside an *enum*,

So we use the enum keyword then give our enum a name (*Using PascalCase*).

enum Size

Followed with curly braces, where we keep our members inside,

enum Size {

  Small,

  Medium,

  Large,

}

*By default TS compiler assigns the first member value as 0, then 1, 2 and so on*…

We can explicitly set the value,

enum Size {

  Small = 1,

  Medium = 2,

  Large = 3,

}

And can even set values as strings,

enum Size {

  Small = "s",

  Medium = "m",

  Large = "l",

}

For now, we will rely on number values,

enum Size {

  Small = 1,

  Medium,

  Large,

} //medium and large will be 2 and 3 automatically

Now we have this new type, we can declare a variable like mySize of type Size and set it Size.Medium.

let mySize: Size = Size.Medium

Let’s see what happens when we console log mySize.

let mySize: Size = Size.Medium;

console.log(mySize);



We see the associate value with our enum member.

The generated JS code from this TS file,

"use strict";

var Size;

(function (Size) {

    Size[Size["Small"] = 1] = "Small";

    Size[Size["Medium"] = 2] = "Medium";

    Size[Size["Large"] = 3] = "Large";

})(Size || (Size = {}));

let mySize = Size.Medium;

console.log(mySize);

Its pretty verbose and lengthy.

To understand this code better, we have a trick,

*If we define our enum as a const, our compiler will generate a more optimized code*.

const enum Size {

  Small = 1,

  Medium,

  Large,

}

Our optimized code,

"use strict";

let mySize = 2;

console.log(mySize)

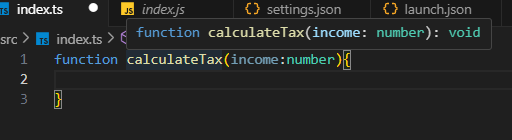
Now we don’t see all that code anymore, just one line for setting mySize to 2.

*Using an enum we can represent a list of related constants*.

**Functions**:

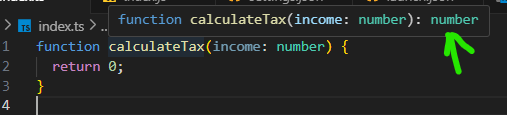
Let’s see how TS helps us prevent common problems when working with functions.

We start by defining a function called calculateTax, with a parameter called *income* as type number.



If we hover our mouse over function name, we can see the type of return value as void which means this function does not return a value.

If we return a value like a number,

, the type of return is a number.

So TS compiler has inferred the type of the return value for us.

*As a best practice, we should always properly annotate our functions. So all the parameters as well as the return type should be properly annotated. Especially if we are building an API for other people to use*.

Here to annotate the return type we add a colon and specify the return type.

function calculateTax(income: number): number 🡨 here{

  return 0;

}

Or

function calculateTax(income: number): void 🡨 here{

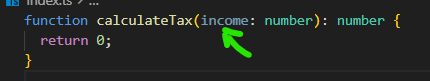
    return;

  }

This type of function setup has a very good benefit, if in case of number return type, if we forget to return a value or return a value of wrong type, we get a compilation error immediately.

As a best practice, always properly annotate your functions.

Moving on,



Notice the income parameter. It is an unused parameter.

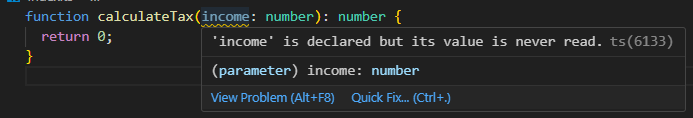
We have a compiler option to detect unused parameters. In the *tsconfig.json* file, under Type checking section.

We have an option called *noUnusedParameters* which is set to true.

 // "noUnusedParameters": true,                       /\* Raise an error when a function parameter isn't read. \*/

We have to explicitly turn this on, because it is not part of the strict setting.

"noUnusedParameters": true,



Now we can see a warning saying income is declared but never read.

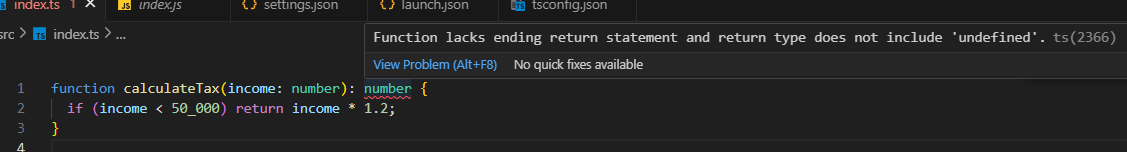
So let’s use this,

function calculateTax(income: number): number {

  if (income < 50\_000) return income \* 1.2;

}

But now we got another compilation error,



Function lacks ending return statement and return type does not include 'undefined'

It sounds a bit cryptic, *but it means that if this condition is true, we are going to return a number. Otherwise JS by default always returns undefined from our functions* and undefined is not a number.

So return else case as well,

function calculateTax(income: number): number {

  if (income < 50\_000) return income \* 1.2;

  return income \* 1.3;

}

We can use implicitly turn on this setting,

    "noImplicitReturns": true,                        /\* Enable error reporting for codepaths that do not explicitly return in a function. \*/

We have another useful setting for detecting unused variables.

function calculateTax(income: number): number {

  let x; //for example this x variable…

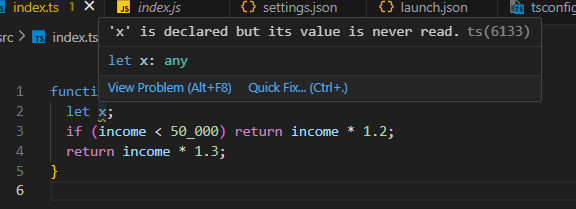
  if (income < 50\_000) return income \* 1.2;

  return income \* 1.3;

}

The setting to detect this is *noUnusedLocals*, let’s uncomment it.

"noUnusedLocals": true,                           /\* Enable error reporting when local variables aren't read. \*/



Let’s add one more parameter,

function calculateTax(income: number, taxYear: number): number {

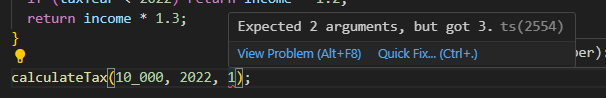
  if (taxYear < 2022) return income \* 1.2;

  return income \* 1.3;

}

calculateTax(10\_000, 2022); 🡪 correct as per TS

In case more arguments are provided,



In case we want to provide an argument optionally, we can do it by using a ? (question mark) after the parameter.

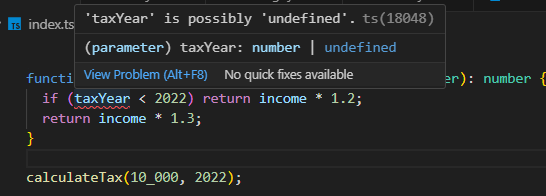
function calculateTax(income: number, taxYear?: number): number {

  if (taxYear < 2022) return income \* 1.2;

  return income \* 1.3;

}

But it will give a compilation error,



*taxYear* is possibly undefined. Because if we don’t provide this argument then it will give undefined.

*One way to solve this issue*:

Use JS trick here and give it a default value using || OR operator like this,

function calculateTax(income: number, taxYear?: number): number {

  if ((taxYear || 2022) < 2022) return income \* 1.2;

  return income \* 1.3;

}

*Better way*,

Give it a default value in the parameter itself.

function calculateTax(income: number, taxYear = 2022): number {

  if (taxYear < 2022) return income \* 1.2;

  return income \* 1.3;

}

Now we can call this function with or without taxYear argument. If we want to override this value, we can simply provide an argument.

function calculateTax(income: number, taxYear = 2022): number {

  if (taxYear < 2022) return income \* 1.2;

  return income \* 1.3;

}

calculateTax(10\_000, 2023); 🡪 it will override taxYear.

Properly Annotate functions, parameters and return type and enable these three compiler options.

    "noUnusedLocals": true,                           /\* Enable error reporting when local variables aren't read. \*/

    "noUnusedParameters": true,                       /\* Raise an error when a function parameter isn't read. \*/

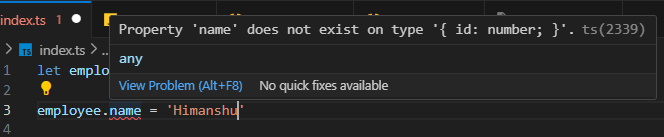
    "noImplicitReturns": true,                        /\* Enable error reporting for codepaths that do not explicitly return in a function. \*/

**Objects**:

Let’s declare an employee object with an id property.

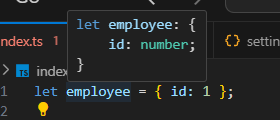
let employee = { id: 1 };

In JS objects are dynamic, so their shape can change over a lifetime of programs. Later we can give this object a new property like name.



But as we can see it’s not a valid TS.

So just like all the variables we have declared so far, *TS compiler has inferred the shape of this employee object*.

🡨 Shape of employee object.

We can also explicitly apply a type annotation here.

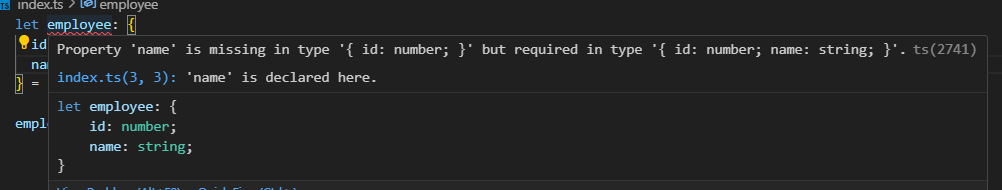
let employee: {

  id: number;

  name: string;

} = { id: 1 };

Previous error is gone, but we are getting a new error,



The reason we are seeing this, is because each employee must have these two properties, but while initializing this object, we haven’t supplied a name property.

Here we have two options,

1. Set the name property to an empty string.

let employee: {

  id: number;

  name: string;

} = { id: 1, name: "" };

1. Make name property optional by putting a ? in the property.

let employee: {

  id: number;

  name?: string;

} = { id: 1 };

Then we don’t need to supply name property while initializing an employee.

*Sometimes we need to make certain properties read only, so we don’t accidently change them later on*. With our current implementation, we can change the ID of an employee.

let employee: {

  id: number;

  name: string;

} = { id: 1, name: "Himanshu" };

employee.id = 0;

*This is where we can use readonly modifier*. So we apply it before the name of the property.

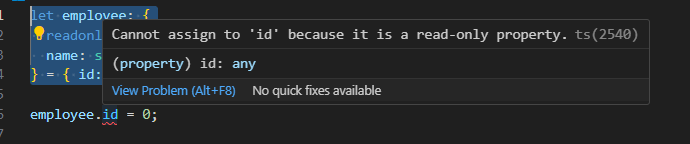
let employee: {

  readonly id: number;

  name: string;

} = { id: 1, name: "Himanshu" };

Now TS compiler gives the error.



Next we learn how to define a method in this object. Let’s say every employee function must have a *retire* method. So *in our type annotation, we need to define the signature of this method, i.e. how many parameters, type of each parameter, type of return value*.

let employee: {

  readonly id: number;

  name: string;

  retire: (date: Date) => void; 🡪 define a method with type annotations

} = {

  id: 1,

  name: "Himanshu",

  retire: (date: Date) => { 🡪 Call that method inside the object.

    console.log(date);

  },

};