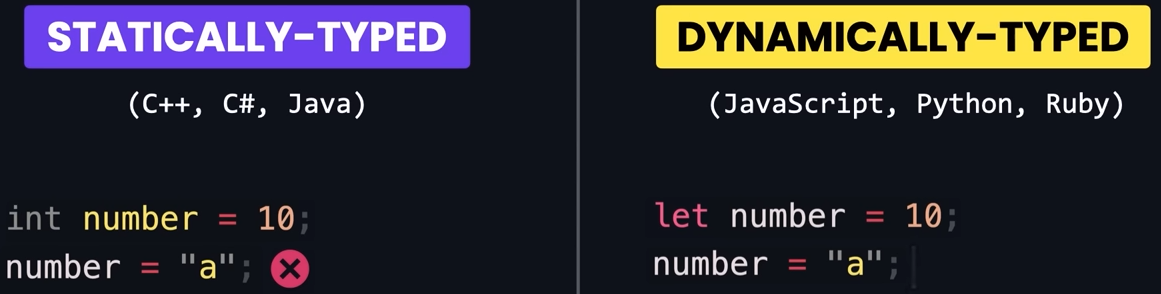
**Typescript – Introduction**

**What Is Typescript**

TypeScript is a syntactic superset of JavaScript which adds **static typing**. This basically means that TypeScript adds syntax on top of JavaScript, allowing developers to add types. Everything we do with JS, we can do with TS.



**Why should I use Typescript**

JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.

In JavaScript, function parameters and variables don't have any information! So, developers need to look at documentation, or guess based on the implementation.

TypeScript allows specifying the types of data being passed around within the code, and has the ability to report errors when the types don't match.

For example, TypeScript will report an error when passing a string into a function that expects a number. JavaScript will not.

**Setting up**

To install Typescript, run: **npm i -g typescript**

To check version, run: **tsc -v**

**Compiling**

To compile the code (specific TS file), run: **tsc [TS filename]**

To compile evert TS file, run: **tsc**

This will create a .js file with the equivalent JS code. By default, the JS compiler ES5, which is old JS.

|  |  |
| --- | --- |
| **Typescript** | **Javascript** |
| let age: number = 20; | var age = 20; |

**Configure the compiler**

To create a configuration file, run: **tsc –init**

This will create a configuration file called **tsconfig.json**

The settings that you’re going to use most of the time are:

* **target:** specifies the version of JS that the TS compiler is going to generate. If you press Ctrl + Space you can see all the valid values. Older version result is less concise code, but they are more compatible with old browsers.
* **rootDir:** specifies the route that contains our source files. So if you put **index.js** inside a **src** folder, the path would be **./src**
* **ourDir:** specifies the directory that will contain our JS files. Usually this is set to **./dist**
* **removeComments:** removes the comments in the JS compiled code.
* **noEmitOnError:** by default the code is compiled even if you have errors in your code. So this should be set to **true**

**Debug**

You can run the code line by line to see what is wrong with the code.

1. Go to the **tsconfig.json** file and enable **sourceMap**
2. Compile the code using **tsc** in the terminal. This will create .map JS file.
3. Click Run and Debug.
4. Click create a launch.json file.
5. From the dropdown, select Node.js.
6. Copy the this line of code inside the launch.json file.

            "preLaunchTask": "tsc: build -tsconfig.json"

1. Click Launch Program and click Step Over to run each line.

**Built-in Types**

|  |  |
| --- | --- |
| **Typescript** | **Javascript** |
| * Number * String * Boolean * Null * Undefined * object | * any * unkown * never * enum * tuple |

In TS, the common practice is to define the type of the variable:

let sales: number = 123\_456\_789;

let course: string = "Typescript";

let is\_published: boolean = true;

However, when we initialize a variable, TS compiler knows the type of the variable without specifying it (you will see if you hover on it).

let sales = 123\_456\_789;

let course = "Typescript";

let is\_published= true;

However, if you don’t initialize it, the variable is of type **any**.

**Type any**

You can set any variables to a number, and then to a string.

let level;

level = 1;

level = "a";

But this goes against the idea of using typescript, so you should avoid any type as much as possible. The example below gives the error “Parameter ‘document’ implicitly has an ‘any’ type.

function render(document) {

  console.log(document);

}

**Arrays**

In JS, you can have more than one type in the same array.

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

However, this will potentially cause issues. So, TS allows to define the type of the elements in the array:

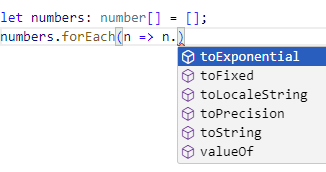
let numbers: number[] = [1, 2, 3];

in the case, if you remove the type annotation, TS compiler will still infer the type as number because all the types are numbers.

If the array is empty, it will be type any. So in this case, you would have to specify the type.

let numbers: number[] = [];

another cool benefit of using TS is that if you iterate over the array, as it knows what type to expect, it presents you all the built-in functions available for that type (in this case, number)



**Readonly**

The **readonly** keyword can prevent arrays from being changed.

const names: readonly string[] = ["Dylan"];  
names.push("Jack"); // Error: Property 'push' does not exist on type 'readonly string[]'.

**Tuple**

A **tuple** is a typed array with a **pre-defined length and types for each index**.

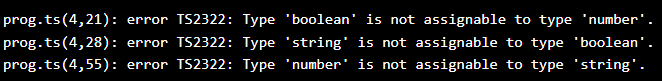
Tuples are great because they allow each element in the array to be a known type of value.

To define a tuple, specify the type of each element in the array:

// define our tuple  
let ourTuple: [number, boolean, string];  
  
// initialize correctly  
ourTuple = [5, false, 'Coding God was here'];

As you can see, we have a number, boolean and a string. But what happens if we try to set them in the wrong order:

// define our tuple  
let ourTuple: [number, boolean, string];  
  
// initialized incorrectly which throws an error  
ourTuple = [false, 'Coding God was mistaken', 5];



You often use tuples when working with a pair of values (best practice).

If you compile the code, you are just going to see a normal array.

**Enums**

An **enum** is a special "class" that represents a group of constants (unchangeable variables). For example, lets define the size of 3-tshirts as constants.

const small = 1;

const medium = 2;

const large = 3;

Another way to define this constants is inside a enum. For this we use Pascal naming convention.

//PascalCase

enum Size {

  Small = 0,

  Medium,

  Large,

}

By default, TS assigns the first value a value of 0. And the 1 and 2 for the next ones. But you can change these values to be whatever you want.

enum Size {

  Small = 1,

  Medium,

  Large,

}

You can also use string values, but in that case you have to specify the value for each member.

enum Size {

  Small = "s",

  Medium = "m",

  Large = "l",

}

Now that we have this new type, you can declare a variable of type Size and set it to Size.Medium.

enum Size {

  Small = 1,

  Medium,

  Large,

}

let mySize: Size = Size.Medium; // equal to 2

To simplify/optimize the JS generated code, use const before enum (see the difference in the JS code below).

const enum Size {

  Small = 1,

  Medium,

  Large,

}

|  |  |
| --- | --- |
| **Without const** | **With const** |
| "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; | let mySize = 2 /\* Size.Medium \*/; |

**Functions**

TS also prevents common problems when working with functions. In the case below, TS can infer the type of the function because it’s returning a number. However, it is good practice to annotate the type, for all the parameters as well and the function.

|  |  |
| --- | --- |
| **Bad Practice** | **Good Practice** |
| function calculateTax(income: number) {    return 0;  } | function calculateTax(income: number): number {    return 0;  } |

The function can be of type void if it’s not going to return a value.

**Unsused Parameters**

You want TS to warn you about unused parameters. To enable that, go to **tsconfig.json** and enable **noUnusedParameters.**

|  |  |
| --- | --- |
| **Bad Practice** | **Good Practice** |
| function calculateTax(income: number) {    return 0;  } | function calculateTax(income: number) {    return income \* 1.2;  } |

**Implicit returns**

You also want TS compiler to warn you if not every path in a function returns a value, like the function below. To do that, enable **noImplicitReturns.**

|  |  |
| --- | --- |
| **Bad Practice** | **Good Practice** |
| function calculateTax(income: number) {  if (income < 50000){      return income \* 1.2  }  //undefined  } | function calculateTax(income: number) {    if (income < 50000) {      return income \* 1.2;    }    return income \* 1.5;  } |

**Unused local variables**

You also want TS compiler to warn you any variable is not used, like the function below. To do that, enable **noUnusedLocals.**

|  |  |
| --- | --- |
| **Bad Practice** | **Good Practice** |
| function calculateTax(income: number) {    let x;    if (income < 50000) {      return income \* 1.2;    }    return income \* 1.5;  } | function calculateTax(income: number) {    if (income < 50000) {      return income \* 1.2;    }    return income \* 1.5;  } |

**Calling a function**

Consider the following function.

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

  if (taxYear < 2022) {

    return income \* 1.2;

  }

  return income \* 1.5;

}

If you want to call this function, you have to supply exactly 2 arguments. In Js, we can pass whatever number we want.

calculateTax(10000, 2022);

**Optional parameters**

You can make a parameter optional, if for example you don’t want to use it every time you call the function.

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

  if (taxYear < 2022) {

    return income \* 1.2;

  }

  return income \* 1.5;

}

calculateTax(10000);

If you don’t supply it, undefined will be used by default, and you cannot compare undefined with 2022. To solve this, you have 2 options:

* Use the OR operator ||

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

  if (taxYear || 2022 < 2022) {

    return income \* 1.2;

  }

  return income \* 1.5;

}

calculateTax(10000, 2022);

* Use the default values (best approach).

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

  if (taxYear || 2022 < 2022) {

    return income \* 1.2;

  }

  return income \* 1.5;

}

calculateTax(10000, 2022);

So, if you don’t supply a value for taxYear when you call the function, then it will have that value by default. If we pass, It will override it.

**Objects**

In JS, objects are dynamic, so the following code would add the property “name” to the employee object and it would be valid JS code.

let employee = { id: 1 };

employee.name = "Mosh";

In TS, this is not valid.

Just like other types, TS allows you to annotate the type of the variable with objects.

let employee: {

  id: number;

  name: string;

} = { id: 1 };

TS will still complain that we didn’t supply the name property when initialized the object. We either set it to and empty string or make it optional.

|  |  |
| --- | --- |
| **Empty String** | **Make it optional** |
| let employee: {    id: number;    name: string;  } = { id: 1, name: "" }; | let employee: {    id: number;    name?: string;  } = { id: 1 }; |

We can also make properties **read only**, so we don’t accidentally change them

let employee: {

  readonly id: number;

  name?: string;

} = { id: 1 };

employee.name = "Mosh";

**Methods**

When working with methods, in our type annotation we need to define:

* How many parameters
* Type of each parameter
* Type of return value

let employee: {

  readonly id: number;

  name: string;

  retire: (date: Date) => void;

} = {

  id: 1,

  name: "Mosh",

  retire: (date: Date) => {

    console.log(date);

  },

};

**Type Aliases**

The problem with the previous approach is, each time you want to create an object with the same structure, you have to repeat the same code for the type annotation.

Using Type aliases allows us to define a custom type, so the type is only defined in a single place.

type Employee = {

  readonly id: number;

  name: string;

  retire: (date: Date) => void;

};

let employee: Employee = {

  id: 1,

  name: "Mosh",

  retire: (date: Date) => {

    console.log(date);

  },

};

**Array of Objects**

Below is an example of how to make an array of the Person object:

type Person = {

  name: string;

  age: number;

};

let people: Person[];

**Type Interface**

Interface works exactly the same as the Alias, but with a slightly different sintax.

interface Person {

  name: string;

  age: number;

};

The way to extend the type is also different when comparing to Alias. In this example, Guy has its own types plus the Person types.

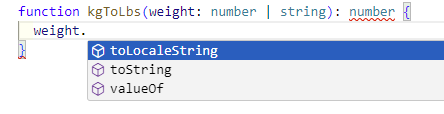
|  |  |
| --- | --- |
| **Alias** | **Interface** |
| type Person = {    name: string;    age: number;  }  type Guy = Person & {    profession: string  } | interface Person {    name: string;    age: number;  }  interface Guy extends Person {    profession: string;  } |

**Union Types**

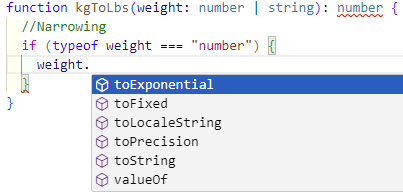
With Union types, we can give a variable or a function parameter more than 1 type. For example, let’s assume the parameter weight can be a number or a string.

function kgToLbs(weight: number | string) {}

if we don’t specify anything, when we type the variable and try to call a method, it will only show the methods that are common between number and string variables.



That’s when we use a technique called narrowing:



So, the correct way to write this function would be something like:

function kgToLbs(weight: number | string): number {

  //Narrowing

  if (typeof weight === "number") {

    return weight \* 2.2;

  } else {

    return parseInt(weight) \* 2.2;

  }

}

kgToLbs(10);

kgToLbs("10");

**Intersection types**

Intersection is another technique to combine types. For example, weight is a number and string at the same time. However, this is not a realistic example.

let weight: number & string;

For example, let’s define a type of object that can be dragged on the screen and one that can be resized.

type Draggable = {

  drag: () => void;

};

type Resizable = {

  resize: () => void;

};

We can then define a new type which combines them (intersection type).

type UIWidget = Draggable & Resizable;

Then when we define an object of type UIWidget, we have to implement the drag and resize methods.

let textBox: UIWidget = {

  drag: () => {},

  resize: () => {},

};

**Literal types**

Sometimes we want to limit the types assigned to a variable. This is when we use Literal Types. For example, quantity can take any number.

let quantity: number;

If we want to limit the numbers it can contain, we can specify a certain value, so quantity can only be set to that value.

let quantity: 50;

We can then use the union operator allow more values:

let quantity: 50 | 100;

Or we can use type alias to define a new type and then use that type in the quantity variable.

type Quantity = 50 | 100;

let quantity: Quantity;

**Nullable types**

By default, TS is very strict about using null and undefined values, once these values are common source of bugs in our application. Foe example, the code below would be valid in JS, however it would generate an error because you cannot call methods on null values.

function greet(name: string) {

  console.log(name.toUpperCase());

}

greet(null);

That’s why the TS compiler stops us from using these values.

What if you want to have the ability to have a null value? In this case, you should have a default behavior in case that happens.

function greet(name: string | null | undefined) {

  if (name) console.log(name.toUpperCase());

  else console.log("Hola!");

}

greet(null);

greet(undefined);

**Optional Chaining**

Consider the function below where the function returns a Customer type, or null if the it cannot find a customer.

type Customer = {

  birthday: Date;

};

function getCustomer(id: number): Customer | null {

  return id === 0 ? null : { birthday: new Date() };

}

let customer = getCustomer(0);

console.log(customer.birthday);

TS gives an error in this case because customer may be null, so we have to handle that case.

if (customer !== null) console.log(customer.birthday);

We can also check for undefined:

type Customer = {

  birthday: Date;

};

function getCustomer(id: number): Customer | null | undefined {

  return id === 0 ? null : { birthday: new Date() };

}

let customer = getCustomer(0);

if (customer !== null && customer !== undefined) console.log(customer.birthday);

**Optional Property Access Operator**

One simpler way to type the code above is by using the optional property access operator. So now, this code is only executed if customer is not null or undefined.

//Optional property access operator

console.log(customer?.birthday);

Now imagine you want to get the full year of the birthday. Let’s make the birthday property optional so TS allows it to be undefined. You will get the same error again because birthday can be undefined. Once again, we can use the optional property access operator.

type Customer = {

  birthday?: Date;

};

function getCustomer(id: number): Customer | null | undefined {

  return id === 0 ? null : { birthday: new Date() };

}

let customer = getCustomer(0);

//Optional property access operator

console.log(customer?.birthday?.getFullYear());

**Optional element access operator**

This is useful when you’re dealing with arrays. So before accessing an element of an array, we first check if it’s not null or undefined.

//Optional element access operator

customers?.[0]

**Optional call operator**

Let’s imagine we have a variable log that references a function of type any that takes a message and prints that message on the console.

let log: any = (message: string) => console.log(message)

Let’s say for some reason that function is null.

let log: any = null;

if you want to call the function a pass some value, the program is going to crash because log is null. So, we only want to call the function if it’s not null or undefined.

log?.("a")

**UseEffect**