**Typescript tutorials**

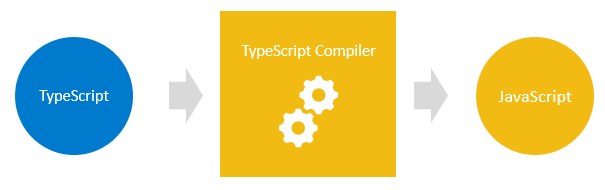
**Introduction to TypeScript**

TypeScript is a superset of [JavaScript](https://www.javascripttutorial.net/).

TypeScript builds on top of JavaScript. First, you write the TypeScript code. Then, you compile the TypeScript code into plain JavaScript code using a TypeScript compiler.

Once you have the plain JavaScript code, you can deploy it to any environment that JavaScript runs.

TypeScript files use the .ts extension rather than the .js extension of JavaScript files.



TypeScript uses the JavaScript syntaxes and adds additional syntaxes for supporting Types.

**TypeScript improves your productivity while helping avoid bugs**

For example, the following function adds two numbers x and y:

function add(x, y) {

return x + y;

}

When you use TypeScript to specify the type for the parameters like this explicitly:

function add(x: number, y: number) {

return x + y;

}

**TypeScript ENV Setup**

The following tools you need to set up to start with TypeScript:

* **Node.js** – Node.js is the environment in which you will run the TypeScript compiler. Note that you don’t need to know node.js.
* **TypeScript compiler** – a Node.js module that compiles TypeScript into JavaScript.
* **Visual Studio Code or VS Code**– a code editor supporting TypeScript. VS Code is highly recommended. However, you can use your favorite editor.
* Install TypeScript compiler
* To install the TypeScript compiler, you launch the Terminal on macOS or Linux and Command Prompt on Windows and type the following command:
* **npm install -g typescript**
* After the installation, you can type the following command to check the current version of the TypeScript compiler:
* tsc --v

**Hello Word Program**

create a new TypeScript file called **app.ts** The extension of a TypeScript file is **.ts**.

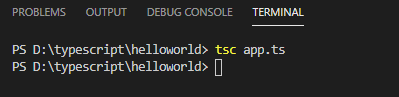
Fourth, type the following source code in the app.ts file:

let message: string = 'Hello, World!';

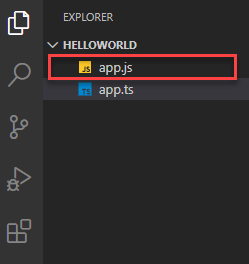
console.log(message);

type the following command on the Terminal to compile the app.ts file:

tsc app.ts



If everything is fine, you’ll see a new file called app.js is generated by the TypeScript compiler:



To run the app.js file in Node.js, you use the following command:

node app.js

**What is Type Annotation in TypeScript**

TypeScript uses type annotations to specify explicit types for identifiers such as variables, functions, objects, etc.

TypeScript uses the syntax **: type** after an identifier as the type annotation, which type can be any valid type.

Once an identifier is annotated with a type, it can be used as that type only. If the identifier is used as a different type, the TypeScript compiler will issue an error

**Type annotations in variables and constants**

The following syntax shows how to specify type annotations for variables and constants:

var name: type;

let name: type = value;

const name: type = value;

In this syntax, the type annotation comes after the variable or constant name and is preceded by a colon (:).

**The following example uses**[**number**](https://www.typescripttutorial.net/typescript-tutorial/typescript-number/)**annotation for a variable:**

let counter: number;

After this, you can only assign a number to the counter variable:

counter = 1;

**If you assign a string to the counter variable, you’ll get an error:**

let counter: number;

counter = 'Hello'; *// compile error*

**You can both use a type annotation for a variable and initialize it in a single statement like this:**

let counter: number = 1;

In this example, we use the number annotation for the counter variable and initialize it to one.

The following shows other examples of primitive type annotations:

let name: string = 'John';

let age: number = 25;

let active: boolean = true;

In this example, the name variable gets the [string](https://www.typescripttutorial.net/typescript-tutorial/typescript-string/) type, the age variable gets the number type, and the active variable gets the [boolean](https://www.typescripttutorial.net/typescript-tutorial/typescript-boolean/) type.

**Arrays Type**

To annotate an [array type](https://www.typescripttutorial.net/typescript-tutorial/typescript-array-type/) you use a specific type followed by a square bracket : type[] :

let arrayName: type[];

For example, the following declares an array of strings:

let names: string[] = ['John', 'Jane', 'Peter', 'David', 'Mary'];

**Objects Type**

To specify a type for an object, you use the object type annotation. For example:

let person: {

name: string;

age: number;

};

person = {

name: 'John',

age: 25,

}; *// valid*

In this example, the person object only accepts an object that has two properties: name with the string type and age with the number type.

**Function arguments & return types**

The following shows a function annotation with parameter type annotation and return type annotation:

let greeting : (name: string) => string;

In this example, you can assign any function that accepts a string and returns a string to the greeting variable:

greeting = function (name: string) {

return `Hi ${name}`;

};

The following causes an error because the function that is assigned to the greeting variable doesn’t match its [function type](https://www.typescripttutorial.net/typescript-tutorial/typescript-function-types/).

greeting = function () {

console.log('Hello');

};

Error:

Type '() => void' is not assignable to type '(name: string) => string'. Type 'void' is not assignable to type 'string'.

Summary

* Use type annotations with the syntax : [type] to explicitly specify a type for a variable, function, function return value, etc.

**TypeScript Type Inference**

Type inference describes where and how TypeScript infers types when you don’t explicitly [annotate](https://www.typescripttutorial.net/typescript-tutorial/typescript-type-annotations/) them.

Basic type inference

When you declare a variable, you can use a [type annotation](https://www.typescripttutorial.net/typescript-tutorial/typescript-type-annotations/) to explicitly specify a type for it. For example:

let counter: number;

However, if you initialize the counter variable with a number, TypeScript will infer the type the counter to be [number](https://www.typescripttutorial.net/typescript-tutorial/typescript-number/). For example:

let counter = 0;

It is equivalent to the following statement:

let counter: number = 0;

**TypeScript infers the following return type of the increment() function as number:**

function increment(counter: number) {

return counter++;

}

It is the same as:

function increment(counter: number) : number {

return counter++;

}

The best common type algorithm

Consider the following assignment:

let items = [1, 2, 3, null];

To infer the type of items variable, TypeScript needs to consider the type of each element in the array.

**Basic Types**

1. **TypeScript Number**

**Summary**: in this tutorial, you’ll learn about the TypeScript number data types.

All numbers in TypeScript are either floating-point values or big integers. The floating-point numbers have the type number while the big integers get the type bigint.

The number type

The following shows how to declare a variable that holds a floating-point value:

let price: number;

Or you can initialize the price variable to a number:

let price = 9.95;

As in JavaScript, TypeScript supports the number literals for decimal, hexadecimal, binary, and octal literals:

**Big Integers**

The [big integers](https://www.javascripttutorial.net/javascript-bigint/) represent the whole numbers larger than 253 – 1. A Big integer literal has the n character at the end of an integer literal like this:

let big: bigint = 9007199254740991n;

**TypeScript String**

**Summary**: in this tutorial, you’ll learn about the TypeScript string data type.

Like JavaScript, TypeScript uses double quotes (") or single quotes (') to surround string literals:

let firstName: string = 'John';

let title: string = "Web Developer";

TypeScript also supports template strings that use the backtick (`) to surround characters.

The template strings allow you to create multi-line strings and provide the string interpolation features.

The following example shows how to create multi-line string using the backtick (`):

let description = `This TypeScript string can

span multiple

lines

`;

String interpolations allow you to embed the variables into the string like this:

let firstName: string = `John`;

let title: string = `Web Developer`;

let profile: string = `I'm ${firstName}.

I'm a ${title}`;

console.log(profile);

Output:

I'm John.

I'm a Web Developer.

Summary

* In TypeScript, all strings get the string type.
* Like JavaScript, TypeScript uses double quotes ("), single quotes ('), and backtick (`) to surround string literals.

**TypeScript Boolean**

Introduction to the TypeScript **boolean**

The TypeScript boolean type has two values: true and false. The boolean type is one of the primitive types in TypeScript.

Declaring boolean variables

In TypeScript, you can declare a boolean variable using the boolean keyword. For example:

let pending: boolean;

pending = true;

*// after a while*

*// ..*

pending = false;

Boolean operator

To manipulate boolean values, you use the boolean operators. TypeScript supports common boolean operators:

| **Operator** | **Meaning** |
| --- | --- |
| && | Logical AND operator |
| || | Logical OR operator |
| ! | Logical NOT operator |

For example:

*// NOT operator*

const pending: boolean = true;

const notPending = !pending; *// false*

console.log(result); *// false*

const hasError: boolean = false;

const completed: boolean = true;

*// AND operator*

let result = completed && hasError;

console.log(result); *// false*

*// OR operator*

result = completed || hasError;

console.log(result); *// true*

Type annotations for boolean

As seen in previous examples, you can use the boolean keyword to annotate the types for the boolean variables:

let completed: boolean = true;

However, TypeScript often [infers types](https://www.typescripttutorial.net/typescript-tutorial/typescript-type-inference/) automatically, so type annotations may not be necessary:

let completed = true;

Like a variable, you can [annotate](https://www.typescripttutorial.net/typescript-tutorial/typescript-type-annotations/) boolean parameters or return the type of a function using the boolean keyword:

function changeStatus(status: boolean): boolean {

*//...*

}

**Boolean Type**

JavaScript has the Boolean type that refers to the non-primitive boxed object. The Boolean type has the letter B in uppercase, which is different from the boolean type.

It’s a good practice to avoid using the Boolean type.

**TypeScript object Type**

**Summary**: in this tutorial, you’ll learn about the TypeScript object type and how to write more accurate object type declarations.

The TypeScript object type represents all values that are not in primitive types.

The following are primitive types in TypeScript:

* [number](https://www.typescripttutorial.net/typescript-tutorial/typescript-number/)
* [bigint](https://www.javascripttutorial.net/javascript-bigint/)
* [string](https://www.typescripttutorial.net/typescript-tutorial/typescript-string/)
* [boolean](https://www.typescripttutorial.net/typescript-tutorial/typescript-boolean/)
* [null](https://www.javascripttutorial.net/object/javascript-null/)
* [undefined](https://www.javascripttutorial.net/javascript-undefined/)
* [symbol](https://www.javascripttutorial.net/symbol/)

The following shows how to declare a variable that holds an object:

let employee: object;

employee = {

firstName: 'John',

lastName: 'Doe',

age: 25,

jobTitle: 'Web Developer'

};

console.log(employee);

Output:

{

firstName: 'John',

lastName: 'Doe',

age: 25,

jobTitle: 'Web Developer'

}

If you reassign a primitive value to the employee object, you’ll get an error :

employee = "Jane";

Error:

error TS2322: Type '"Jane"' is not assignable to type 'object'.

**To explicitly specify properties of the employee object, you first use the following syntax to declare the employee object:**

let employee: {

firstName: string;

lastName: string;

age: number;

jobTitle: string;

};

**And then assign the employee object to a literal object with the described properties:**

employee = {

firstName: 'John',

lastName: 'Doe',

age: 25,

jobTitle: 'Web Developer'

};

**object vs. Object**

TypeScript has another type called Object with the letter O in uppercase. It’s important to understand the differences between them.

The **object** type represents all non-primitive values while the **Object** type describes the functionality of all objects.

For example, the **Object** type has the toString() and valueOf() methods that can be accessible by any object.

let vacant: {} = {};

console.log(vacant.toString());

Output:

[object Object]

**TypeScript Array Type**

**Summary**: in this tutorial, you’ll learn about the TypeScript array type and its basic operations.

**Introduction to TypeScript array type**

A TypeScript array is an ordered list of data. To declare an array that holds values of a specific type, you use the following syntax:

let arrayName: type[];

For example, the following declares an array of [strings](https://www.typescripttutorial.net/typescript-tutorial/typescript-string/):

let skills: string[] = [];

And you can add one or more strings to the array:

skills[0] = "Problem Solving";

skills[1] = "Programming";

or use the push() method:

skills.push('Software Design');

The following declares a variable and assigns an array of strings to it:

let skills = ['Problem Sovling','Software Design','Programming'];

In this example, TypeScript [infers](https://www.typescripttutorial.net/typescript-tutorial/typescript-type-inference/) the skills array as an array of strings. It is equivalent to the following:

let skills: string[];

skills = ['Problem Sovling','Software Design','Programming'];

After you define an array of a specific type, TypeScript will prevent you from adding incompatible values. For example, the following will cause an error:

skills.push(100);

… because we’re trying to add a number to the string array.

Error:

Argument of type 'number' is not assignable to parameter of type 'string'.

When you extract an element from the array, TypeScript [infers the type of the array element](https://www.typescripttutorial.net/typescript-tutorial/typescript-type-inference/). For example:

let skill = skills[0];

console.log(typeof(skill));

Output:

string

In this example, we extract the first element of the skills array and assign it to the skill variable.

Since an element in a string array is a string, TypeScript infers the type of the skill variable to string as shown in the output.

**TypeScript array properties and methods**

TypeScript arrays have the same [properties and methods as JavaScript arrays](https://www.javascripttutorial.net/javascript-array-methods/). For example, the following uses the length property to get the number of elements in an array:

let series = [1, 2, 3];

console.log(series.length); *// 3*

You can use all the useful array methods such as [forEach()](https://www.javascripttutorial.net/javascript-array-foreach/), [map()](https://www.javascripttutorial.net/javascript-array-map/), [reduce()](https://www.javascripttutorial.net/javascript-array-reduce/), and [filter()](https://www.javascripttutorial.net/javascript-array-filter/). For example:

let series = [1, 2, 3];

let doubleIt = series.map(e => e\* 2);

console.log(doubleIt);

Output:

[ 2, 4, 6 ]

**Storing values of mixed types**

The following illustrates how to define an array that holds both strings and numbers:

let scores = ['Programming', 5, 'Software Design', 4];

In this case, TypeScript infers the scores array as an array of string | number. It’s equivalent to the following:

let scores : (string | number)[];

scores = ['Programming', 5, 'Software Design', 4];

**TypeScript Tuple**

**Summary**: in this tutorial, you’ll learn about the TypeScript Tuple type and its usage.

Introduction to TypeScript Tuple type

A tuple works like an [array](https://www.typescripttutorial.net/typescript-tutorial/typescript-array-type/) with some additional considerations:

* The number of elements in the tuple is fixed.
* The types of elements are known, and need not be the same.

For example, you can use a tuple to represent a value as a pair of a string and a number:

let skill: [string, number];

skill = ['Programming', 5];

The order of values in a tuple is important. If you change the order of values of the skill tuple to [5, "Programming"], you’ll get an error:

let skill: [string, number];

skill = [5, 'Programming'];

Error:

error TS2322: Type 'string' is not assignable to type 'number'.

For this reason, it’s a good practice to use tuples with data that are related to each other in a specific order.

For example, you can use a tuple to define an RGB color that always comes in a three-number pattern:

(r,g,b)

For example:

let color: [number, number, number] = [255, 0, 0];

The color[0], color[1], and color[2] would be logically mapped to Red, Green and Blue color values.

**What is an enum**

An enum is a group of named constant values. Enum stands for enumerated type.

To define an enum, you follow these steps:

* First, use the enum keyword followed by the name of the enum.
* Then, define constant values for the enum.

**The following shows the syntax for defining an enum:**

enum name {constant1, constant2, ...};

In this syntax, the constant1, constant2, etc., are also known as the members of the enum.

TypeScript enum type example

The following example creates an enum that represents the months of the year:

enum Month {

Jan,

Feb,

Mar,

Apr,

May,

Jun,

Jul,

Aug,

Sep,

Oct,

Nov,

Dec

};

In this example, the enum name is Month and constant values are Jan, Feb, Mar, and so on.

The following declares a function that uses the Month enum as the type of the month parameter:

function isItSummer(month: Month) {

let isSummer: boolean;

switch (month) {

case Month.Jun:

case Month.Jul:

case Month.Aug:

isSummer = true;

break;

default:

isSummer = false;

break;

}

return isSummer;

}

And you can call it like so:

console.log(isItSummer(Month.Jun)); *// true*

This example uses constant values including Jan, Feb, Mar, … in the enum rather than magic values like 1, 2, 3,… This makes the code more obvious.

**TypeScript any Type**

**Summary**: in this tutorial, you will learn about the TypeScript any type and how to use it properly in your code.

**Introduction to TypeScript any type**

Sometimes, you may need to store a value in a variable. But you don’t know its type when writing the program. And the unknown value may come from a third-party API or user input.

In this case, you want to **opt out of the type checking** and allow the value to pass through the compile-time check.

For example:

let result: any;

result = 1;

console.log(result);

result = 'Hello';

console.log(result);

result = [1, 2, 3];

const total = result.reduce((a: number, b: number) => a + b, 0);

console.log(total);

The any type provides you with a way to work with the existing JavaScript codebase. It allows you to gradually opt in and opt out of type-checking during compilation. Therefore, you can use the any type for migrating a JavaScript project over to TypeScript.

TypeScript any vs. object

If you declare a variable with the [object](https://www.typescripttutorial.net/typescript-tutorial/typescript-object-type/) type, you can also assign any value to it. However, you cannot call a method on it even if the method exists. For example:

let result: any;

result = 10.123;

console.log(result.toFixed());

result.willExist(); *//*Code language: JavaScript (javascript)

In this example, the TypeScript compiler doesn’t issue any warning even the willExist() method doesn’t exist at compile time because the willExist() method might be available at runtime.

If you run the code, you’ll see the following error message on the console window:

TypeError: result.willExist is not a functionCode language: JavaScript (javascript)

However, if you change the type of the result variable to object, the TypeScript compiler will issue two errors:

let result: object;

result = 10.123;

result.toFixed();Code language: JavaScript (javascript)

Errors:

app.ts:2:1 - error TS2322: Type 'number' is not assignable to type 'object'.

2 result = 10.123;

**Summary**

* The TypeScript any type allows you to store a value of any type. It instructs the compiler to skip type-checking.
* Use the any type to store a value that you don’t know its type at the compile-time or when you migrate a JavaScript project over to a TypeScript project.

**TypeScript Unknown Type**

**Summary**: in this tutorial, you will learn about the TypeScript unknown type to enforce type checking of an unknown value.

**Introduction to the TypeScript unknown type**

In TypeScript, the unknown type can hold a value that is not known upfront but requires type checking.

To declare a variable of the unknown type, you use the following syntax:

let result: unknown;

Like the [any type](https://www.typescripttutorial.net/typescript-tutorial/typescript-any-type/), you can assign any value to a variable of the unknown type. For example:

let result: unknown;

result = 1;

result = 'hello';

result = false;

result = Symbol();

result = { name: 'John' };

result = [1, 2, 3];

Unlike the any type, TypeScript checks the type before performing operations on it. For example, you cannot call a method or apply an operator on a unknown value. If you attempt to do so, the TypeScript compiler will issue an error:

let result: unknown;

result = [1,2,3];

const total = result.reduce((a: number, b:number ) => a + b, 0);

console.log(total);

In this example, the result variable has the type of unknown. We assign an array the result value, but its type is still unknown. Therefore, we cannot call the reduce() method of an array on it.

To call the reduce() method on the result variable, you need to use the [type assertion](https://www.typescripttutorial.net/typescript-tutorial/type-assertions/) to explicitly tell the TypeScript compiler that the type of the result is [array](https://www.typescripttutorial.net/typescript-tutorial/typescript-array-type/). For example:

let result: unknown;

result = [1, 2, 3];

const total = (result as number[]).reduce((a: number, b: number) => a + b, 0);

console.log(total); *// 6*

In this example, we explicitly tell the TypeScript compiler that the type of the result is an array of numbers (result as number[]).

Therefore, we can call the reduce() method on the result array without any issues.

Unknown vs Any type

The following table highlights the key differences between the unknown and any types:

| **Feature** | **any** | **unknown** |
| --- | --- | --- |
| **Type Safety** | No type-safety | Enforces type safety |
| **Operations** | Operations can be performed without checks | Operations cannot be performed without type assertion (narrowing type) |
| **Use cases** | Useful for dynamic values but unsafe. | Useful for dynamic values and safe because it requires validation before use. |
| **Type Checking** | TypeScript compiler does not perform a type checking on an any variable. | TypeScript compiler enforces a type checking on an unknown variable. |
| **Common Scenarios** | Used for migrating JavaScript codebase to TypeScript. | Used when handling data from external sources (API calls, databases, ..) where type validation is necessary. |

TypeScript unknown examples

Let’s take some practical examples of using the Typescript unknown type.

1) Handling external data

When receiving data from an external API, you can use the unknown type to enforce validation before processing it.

The following example shows how to use the [fetch method](https://www.javascripttutorial.net/web-apis/javascript-fetch-api/) to call an API from the https://jsonplaceholder.typicode.com/posts endpoint:

const fetchData = async (url: string): Promise<unknown> => {

const response = await fetch(url);

return await response.json();

};

const showPosts = async () => {

const url = 'https://jsonplaceholder.typicode.com/posts';

try {

const posts = await fetchData(url); *// unknown type*

(

posts as { userId: number; id: number; title: string; body: string }[]

).map((post) => console.log(post.title));

} catch (err) {

console.log(err);

}

};

showPosts();

How it works.

First, define a function fetchData that calls API from a URL and returns JSON data. Since the shape of the returned data is not known, the function returns a Promise<unknown> value:

const fetchData = async (url: string): Promise<unknown> => {

const response = await fetch(url);

return await response.json();

};

Second, define the showPosts() function that uses the fetchData() function to call an API from the endpoint https://jsonplaceholder.typicode.com/posts:

const showPosts = async () => {

const url = 'https://jsonplaceholder.typicode.com/posts';

try {

const posts = await fetchData(url); *// unknown type*

(

posts as { userId: number; id: number; title: string; body: string }[]

).map((post) => console.log(post.title));

} catch (err) {

console.log(err);

}

};

In this example, the posts variable has a type of unknown.

Before accessing its title property, we use [type assertion](https://www.typescripttutorial.net/typescript-tutorial/type-assertions/) to instruct the TypeScript compiler to treat it as an array of post objects:

posts as { userId: number; id: number; title: string; body: string }[]Code language: TypeScript (typescript)

Third, call the showPosts() function:

showPosts();

2) Creating type-safe interfaces

The following example defines a function format that format a value before logging it to the console:

function format(value: unknown): void {

switch (typeof value) {

case 'string':

console.log('String:', value.toUpperCase());

break;

case 'number':

console.log('Number:', value.toFixed(2));

break;

default:

console.log('Other types:', value);

}

}

In this example, before accessing a method of the value, we validate its type to ensure that the operation is valid.

**Summary**

* The unknown type is like any type but more restrictive.
* Use the unknown type to handle data coming from external sources and requires validation before use.

**TypeScript void Type**

**Summary**: in this tutorial, you will learn about the TypeScript void type and how to use it as the return type of functions that do not return any value.

**Introduction to TypeScript void type**

The void type denotes the absence of having any type at all. Typically, you use the void type as the return type of functions that do not return a value. For example:

*function log(message): void {*

*console.log(messsage);*

*}*

It is a good practice to add the void type as the return type of a function or a method that doesn’t return any value. By doing this, you can gain the following benefits:

* Improve clarity of the code: you do not have to read the whole function body to see if it returns anything.
* Ensure type-safe: you will never assign the function with the void return type to a variable.

**Notice** that if you use the void type for a variable, you can only assign undefined to that variable. In this case, the void type is not useful. For example:

let useless: void = undefined;

useless = 1; *// error*

*If the --strictNullChecks flag is not specified, you can assign the useless to null.*

useless = null; *// OK if --strictNullChecks is not specified*

Summary

* Use the void type as the return type of functions that do not return any value.

**TypeScript never Type**

**Summary**: in this tutorial, you will learn about the TypeScript never type to represent a value that never occurs.

**Introduction to the TypeScript never type**

In TypeScript, a type is like a set of values. For example, the number type holds the numbers 1, 2, 3, etc. The string type holds the strings like 'Hi', 'Hello', etc. The null type holds a single value, which is null.

The never type is a type that holds **no value**. It is like an **empty set**.

Since a never type does not hold any value, you cannot assign a value to a variable with the never type.

For example, the following will result in an error:

let empty: never = 'hello';

The TypeScript compiler issues the following error:

Type 'string' is not assignable to type 'never'

So why do we need the never type in the first place?

Since the never type has zero value, you can use it to denote an **impossibility** in the type system.

For example, you may have an [intersection type](https://www.typescripttutorial.net/typescript-tutorial/typescript-intersection-types/) that can be both a string and a number at the same time, which is impossible:

type Alphanumeric = string & number; *// never*

Therefore, the TypeScript compiler infers the type of Alphanumeric as never.

This is because string and number are mutually exclusive. In other words, a value cannot be both a string and a number simultaneously.

Typically, you use the never type to represent the return type of a function that never returns the control to the caller. For example, a function that always throws an error:

function raiseError(message: string): never {

throw new Error(message);

}Code language: TypeScript (typescript)

Please do not confuse with functions that return [void](https://www.typescripttutorial.net/typescript-tutorial/typescript-void-type/) but still return the control to the caller.

If you have a function that contains an indefinite loop, its return type should be never. For example:

function forever(): never {

while (true) {}

}Code language: TypeScript (typescript)

In this example, the type of the return type of the forever() function is never.

The TypeScript never example

Let’s take an example of using the never type:

type Role = 'admin' | 'user';

const authorize = (role: Role): string => {

switch (role) {

case 'admin':

return 'You can do anything';

case 'user':

return 'You can do something';

default:

*// never reach here util we add a new role*

const \_unreachable: never = role;

throw new Error(`Invalid role: ${\_unreachable}`);

}

};

console.log(authorize('admin'));

Code language: JavaScript (javascript)

How it works.

Step 1. Define a type Role that can be either a string 'admin' or 'user':

type Role = 'admin' | 'user';

**TypeScript union Type**

**Summary**: in this tutorial, you will learn about the TypeScript union type that allows you to store a value of one or several types in a variable.

**Introduction to TypeScript union type**

Sometimes, you will run into a function that expects a parameter that is either a number or a string. For example:

function add(a: any, b: any) {

if (typeof a === 'number' && typeof b === 'number') {

return a + b;

}

if (typeof a === 'string' && typeof b === 'string') {

return a.concat(b);

}

throw new Error('Parameters must be numbers or strings');

}

In this example, the add() function will calculate the sum of its parameters if they are numbers.

If the parameters are strings, the add() function will concatenate them into a single string.

If the parameters are neither numbers nor strings, the add() function throws an error.

The problem with the parameters of the add() function is that its parameters have the [any](https://www.typescripttutorial.net/typescript-tutorial/typescript-any-type/) type. It means that you can call the function with arguments that are neither numbers nor strings, the TypeScript will be fine with it.

This code will be compiled successfully but cause an error at runtime:

add(true, false);

To resolve this, you can use the TypeScript union type. The union type allows you to combine multiple types into one type.

For example, the following variable is of type number or string:

let result: number | string;

result = 10; *// OK*

result = 'Hi'; *// also OK*

result = false; *// a boolean value, not OK*

A union type describes a value that can be one of several types, not just two. For example number | string | boolean is the type of a value that can be a number, a string, or a boolean.

Back to the add() function example, you can change the types of parameters from the any to a union like this:

function add(a: number | string, b: number | string) {

if (typeof a === 'number' && typeof b === 'number') {

return a + b;

}

if (typeof a === 'string' && typeof b === 'string') {

return a.concat(b);

}

throw new Error('Parameters must be numbers or strings');

}

We can specify the union type for the add function:

function add(a: number | string, b: number | string) : number | string {

if (typeof a === 'number' && typeof b === 'number') {

return a + b;

}

if (typeof a === 'string' && typeof b === 'string') {

return a.concat(b);

}

throw new Error('Parameters must be numbers or strings');

}

Later, you will learn about the [generic type](https://www.typescripttutorial.net/typescript-tutorial/typescript-generics/) to handle this more elegantly.

**Summary**

* A TypeScript union type allows you to store a value of one or several types in a variable.

**TypeScript String Literal Types**

**Summary**: in this tutorial, you will learn about the TypeScript string literal types that define a type that accepts a specified string literal.

**The string literal types allow you to define a type that accepts only one specified string literal.**

The following defines a string literal type that accepts a literal string 'click':

let click: 'click';

The click is a string literal type that accepts only the string-literal 'click'. If you assign the literal string 'click' to the click, it will be valid:

click = 'click'; *// valid*

However, when you assign another string literal to the click, the TypeScript compiler will issue an error. For example:

click = 'dblclick'; *// compiler error*

Error:

Type '"dblclick"' is not assignable to type '"click"'.

The string literal type is useful to limit a possible string value that a variable can store.

The string literal types can combine nicely with the [union types](https://www.typescripttutorial.net/typescript-tutorial/typescript-union-type/) to define a finite set of string literal values for a variable:

let mouseEvent: 'click' | 'dblclick' | 'mouseup' | 'mousedown';

mouseEvent = 'click'; *// valid*

mouseEvent = 'dblclick'; *// valid*

mouseEvent = 'mouseup'; *// valid*

mouseEvent = 'mousedown'; *// valid*

mouseEvent = 'mouseover'; *// compiler error*

If you use the string literal types in multiple places, they will be verbose.

To avoid this, you can use the type aliases. For example:

type MyMouseEvent = 'click' | 'dblclick' | 'mouseup' | 'mousedown';

let mouseEvent: MyMouseEvent;

mouseEvent = 'click'; *// valid*

mouseEvent = 'dblclick'; *// valid*

mouseEvent = 'mouseup'; *// valid*

mouseEvent = 'mousedown'; *// valid*

mouseEvent = 'mouseover'; *// compiler error*

**let anotherEvent: MyMouseEvent;**

**Summary**

* A TypeScript string literal type defines a type that accepts specified string literal.
* Use the string literal types with union types and type aliases to define types that accept a finite set of string literals.

**TypeScript Type Aliases**

**Summary**: in this tutorial, you will learn how to define new names for types using type aliases.

**Introduction to TypeScript type aliases**

In TypeScript, a type alias allows you to create a new name for an existing type.

Type aliases can be useful for:

* Simplifying complex types.
* Making code more readable.
* Creating reusable types that can be used in many places in the codebase.

To define a type alias, you use the type keyword followed by the alias name and the type it represents.

Here’s the syntax for defining a type alias

type alias = existingType;

The existing type can be any valid TypeScript type including primitive type, [object type](https://www.typescripttutorial.net/typescript-tutorial/typescript-object-type/), [union type](https://www.typescripttutorial.net/typescript-tutorial/typescript-union-type/), [intersection type](https://www.typescripttutorial.net/typescript-tutorial/typescript-intersection-types/), and [function type](https://www.typescripttutorial.net/typescript-tutorial/typescript-function-types/).

Type alias examples

1) Primitive types

The following example uses the type alias chars for the string type:

type Name: string;

let firstName: Name;

let lastName: Name;

In this example, we create the Name as a type alias for the string type and use it to declare two variables firstName and lastName.

2) Object types

The following example defines a type alias Person for an object that has two properties name and age:

type Person = {

name: string;

age: number;

};

let person: Person = {

name: 'John',

age: 25

};

3) Union Types

The following example shows how to define a type alias for the union type string | number:

type alphanumeric = string | number;

let input: alphanumeric;

input = 100; *// valid*

input = 'Hi'; *// valid*

input = false; *// Compiler error*

4) Intersection Types

The following example shows how to create a type alias for the intersection type Personal & Contact:

type Personal = {

name: string;

age: number;

};

type Contact = {

email: string;

phone: string;

};

type Candidate = Personal & Contact;

let candidate: Candidate = {

name: "Joe",

age: 25,

email: "joe@example.com",

phone: "(408)-123-4567"

};

**Summary**

* Use type aliases to define new names for existing types.

**TypeScript Casting**

**Casting with as**

A straightforward way to cast a variable is using the as keyword, which will directly change the type of the given variable.

Example

let x: unknown = 'hello';  
console.log((x as string).length);

**Casting with <>**

Using <> works the same as casting with as.

Example

let x: unknown = 'hello';  
console.log((<string>x).length);

**Force casting**

To override type errors that TypeScript may throw when casting, first cast to unknown, then to the target type.

Example

let x = 'hello';  
console.log(((x as unknown) as number).length); // x is not actually a number so this will return undefined

**How to convert a string to number in TypeScript?**

[Exactly like](https://stackoverflow.com/questions/1133770/convert-a-string-to-an-integer) [in JavaScript](https://stackoverflow.com/questions/17106681/parseint-vs-unary-plus-when-to-use-which/17106702#17106702), you can use the [parseInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt) or [parseFloat](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat) functions, or simply use the unary + operator:

var x = "32";

var y: number = +x;

**The TypeScript way to do this would be:**

Number('1234') // 1234

Number('9BX9') // NaN

**As shown by other answers here, there are multiple ways to do the conversion:**

Number('123');

+'123';

parseInt('123');

parseFloat('123.45')

**typeof Example :**

let str1:string = "102.2";

console.log(typeof str1);

let num = parseFloat(str1);

console.log(`${num}` + " is of type :" + typeof num);

let str2:string = "61";

console.log(typeof str2);

let num2 = parseInt(str2);

console.log(`${num2}` + " is of type :" + typeof num2);

**Output:**

string  
102.2 is of type :number  
string  
61 is of type :number