

---

# Guidebook

# The TypeScript Tutorial

Welcome to **aGuideHub.com**

---

## Index

<b>1.</b>	<b>Introduction To TypeScript</b>	<b>7</b>
1.1.	In this TypeScript tutorial, you'll learn the following:	7
<b>2.</b>	<b>Prerequisites</b>	<b>7</b>
2.1.	To follow the tutorial, you need to have the following:	7
<b>3.</b>	<b>Getting Started</b>	<b>8</b>
3.1.	Introduction to TypeScript	8
3.2.	Why TypeScript?	9
3.2.1.	1) TypeScript improves your productivity while helping avoid bugs	9
3.2.2.	2) TypeScript brings the future JavaScript to today	10
3.3.	TypeScript Environment Setup	10
3.3.1.	Install Node.js	11
3.3.2.	Install TypeScript compiler	11
3.4.	TypeScript Sample Program with node.js	12
3.5.	TypeScript Sample Program with vanilla JavaScript	13
<b>4.</b>	<b>Basic Types</b>	<b>15</b>
4.1.	TypeScript Types	15
4.1.1.	What is a type in TypeScript	16
4.1.2.	Types in TypeScript	16
4.1.3.	Purposes of types in TypeScript	17
4.1.4.	Examples of TypeScript type	17
4.1.5.	Summary	18
4.2.	Type Annotations	18
4.2.1.	What is Type Annotation in TypeScript?	19
4.2.2.	Type annotations in variables and constants	19
4.2.3.	Type annotation examples	20
4.2.4.	Summary	21
4.3.	TypeScript Type Inference	21
4.3.1.	Basic type inference	21
4.3.2.	The best common type algorithm	22
4.3.3.	Contextual typing	23
4.3.4.	Type inference vs. Type annotations	23
4.3.5.	Summary	24
4.4.	TypeScript Number	24
4.4.1.	The number type	24
4.4.2.	Decimal numbers	24
4.4.3.	Binary Numbers	24
4.4.4.	Octal Numbers	25
4.4.5.	Hexadecimal numbers	25
4.4.6.	Big Integers	25
4.4.7.	Summary	25
4.5.	TypeScript String	25
4.5.1.	Summary	26
4.6.	TypeScript Boolean	26
4.7.	TypeScript object Type	27

4.7.1.	Introduction to TypeScript object type	27
4.7.2.	object vs. Object	29
4.7.3.	The empty type {}	29
4.7.4.	Summary	30
4.8.	TypeScript Array Type	30
4.8.1.	Introduction to TypeScript array type	30
4.8.2.	TypeScript array properties and methods	31
4.8.3.	Storing values of mixed types	31
4.8.4.	Summary	32
4.9.	TypeScript Tuple	32
4.9.1.	Introduction to TypeScript Tuple type	32
4.9.2.	Optional Tuple Elements	33
4.9.3.	Summary	33
4.10.	TypeScript Enum	33
4.10.1.	What is an enum	33
4.10.2.	TypeScript enum type example	34
4.10.3.	How TypeScript enum works	34
4.10.4.	Specifying enum members' numbers	36
4.10.5.	When to use an enum	36
4.10.6.	Summary	37
4.11.	TypeScript any Type	37
4.11.1.	Introduction to TypeScript any type	38
4.11.2.	TypeScript any: implicit typing	38
4.11.3.	TypeScript any vs. object	39
4.11.4.	Summary	39
4.12.	TypeScript never Type	39
4.12.1.	Summary	40
4.13.	TypeScript union Type	41
4.13.1.	Introduction to TypeScript union type	41
4.13.2.	Summary	42
4.14.	TypeScript Type Aliases	42
4.14.1.	Introduction to TypeScript type aliases	42
4.14.2.	Summary	43
4.15.	TypeScript String Literal Types	43
4.15.1.	Summary	44
<b>5.</b>	<b>CONTROL FLOW STATEMENTS</b>	<b>44</b>
5.1.	TypeScript if else	44
5.1.1.	TypeScript if statement	44
5.1.2.	TypeScript if...else statement	45
5.1.3.	Ternary operator ?:	46
5.1.4.	TypeScript if...else if...else statement	46
5.1.5.	Summary	47
5.2.	TypeScript switch case	48
5.2.1.	Introduction to TypeScript switch case statement	48
5.2.2.	TypeScript switch case statement examples	49
5.2.3.	1) A simple TypeScript switch case example	49
5.2.4.	2) Grouping case example	49
5.3.	TypeScript for	51
5.3.1.	Introduction to the TypeScript for statement	51

5.3.2.	TypeScript for examples	52
5.3.3.	Summary	53
5.4.	TypeScript while	54
5.4.1.	Introduction to the TypeScript while statement	54
5.4.2.	TypeScript while statement examples	54
5.4.3.	Summary	56
5.5.	TypeScript do while	56
5.5.1.	Introduction to TypeScript do...while statement	56
5.5.2.	TypeScript do...while statement example	56
5.5.3.	Summary	57
5.6.	TypeScript break	57
5.6.1.	Using TypeScript break to terminate a loop	57
5.6.2.	Using the break statement to break a switch	58
5.6.3.	Summary	59
<b>6.</b>	<b>Functions</b>	<b>59</b>
6.1.	TypeScript Functions	59
6.1.1.	Introduction to TypeScript functions	59
6.1.2.	Summary	60
6.2.	TypeScript Function Types	60
6.2.1.	Introduction to TypeScript function types	61
6.2.2.	Inferring function types	62
6.3.	TypeScript Optional Parameters	62
6.3.1.	Summary	63
6.4.	TypeScript Default Parameters	63
6.4.1.	Introduction to TypeScript default parameters	63
6.4.2.	Default parameters and Optional parameters	64
6.4.3.	Summary	66
6.5.	TypeScript Rest Parameters	66
6.6.	TypeScript Function Overloadings	67
6.6.1.	Introduction to TypeScript function overloadings	67
6.6.2.	Function overloading with optional parameters	68
6.6.3.	Method overloading	69
6.6.4.	Summary	69
<b>7.</b>	<b>Class</b>	<b>70</b>
7.1.	TypeScript Class	70
7.1.1.	Introduction to the TypeScript Class	70
7.1.2.	Summary	72
7.2.	TypeScript Access Modifiers	72
7.2.1.	The private modifier	72
7.2.2.	The public modifier	73
7.2.3.	The protected modifier	74
7.2.4.	Summary	74
7.3.	TypeScript readonly	75
7.3.1.	Readonly vs. const	76
7.3.2.	Summary	76
7.4.	TypeScript Inheritance	76
7.4.1.	Introduction to the TypeScript inheritance	76
7.4.2.	Constructor	77
7.4.3.	Method overriding	78

7.4.4.	Summary	78
7.5.	TypeScript Static Methods and Properties	79
7.5.1.	Static properties	79
7.5.2.	Static methods	79
7.5.3.	Summary	80
7.6.	TypeScript Abstract Classes	80
7.6.1.	Introduction to TypeScript abstract classes	80
7.6.2.	Summary	82
<b>8.</b>	<b>Interface</b>	<b>83</b>
8.1.	TypeScript Interface	83
8.1.1.	Introduction to TypeScript interfaces	83
8.1.2.	Optional properties	85
8.1.3.	Readonly properties	85
8.1.4.	Function types	85
8.1.5.	Class Types	87
8.1.6.	Summary	87
8.2.	TypeScript Extend Interfaces	88
8.2.1.	Interfaces extending one interface	88
8.2.2.	Interfaces extending multiple interfaces	89
8.2.3.	Interfaces extending classes	89
8.2.4.	Summary	89
<b>9.</b>	<b>Advanced Types</b>	<b>90</b>
9.1.	TypeScript Intersection Types	90
9.1.1.	Introduction to TypeScript intersection types	90
9.1.2.	Type Order	92
9.1.3.	Summary	92
9.2.	TypeScript Type Guards	92
9.2.1.	typeof	92
9.2.2.	instanceof	94
9.2.3.	in	95
9.2.4.	User-defined Type Guards	96
9.2.5.	Summary	96
9.3.	Type Casting	97
9.3.1.	Type casting using the as keyword	97
9.3.2.	Type Casting using the <> operator	98
9.3.3.	Summary	98
9.4.	Type Assertions	98
9.4.1.	Introduction to Type Assertions in TypeScript	99
9.4.2.	The alternative Type Assertion syntax	99
9.4.3.	Summary	100
<b>10.</b>	<b>Generics</b>	<b>100</b>
10.1.	TypeScript Generics	100
10.1.1.	Introduction to TypeScript Generics	100
10.1.2.	Using the any type	101
10.1.3.	TypeScript generics come to rescue	102
10.1.4.	Calling a generic function	102
10.1.5.	Generic functions with multiple types	103
10.1.6.	Benefits of TypeScript generics	103
10.1.7.	Summary	103

---

10.2.	TypeScript Generic Constraints	104
10.2.1.	Introduction to generic constraints in TypeScript	104
10.2.2.	Using type parameters in generic constraints	105
10.2.3.	Summary	106
10.3.	TypeScript Generic Interfaces	106
10.3.1.	Introduction to TypeScript generic interfaces	106
10.3.2.	TypeScript generic interface examples	107
10.4.	TypeScript Generic Classes	108
10.4.1.	Introduction to TypeScript generic classes	108
10.4.2.	TypeScript generic classes example	109
<b>11.</b>	<b>Modules</b>	<b>111</b>
11.1.	TypeScript Modules	111
11.1.1.	Introduction to TypeScript modules	111
11.1.2.	Creating a new module	112
11.1.3.	Export statements	112
11.1.4.	Importing a new module	112
11.1.5.	Importing types	114
11.1.6.	Re-exports	114
11.1.7.	Default Exports	114
11.1.8.	Summary	115

---

## Introduction To TypeScript

TypeScript quickly gaining traction in the developer world cause typescript fixes many of the limitations of JavaScript while keeping all the benefits.

As we know TypeScript is a **Typed** version of JavaScript and that's the reason TypeScript helps you speed up the development by catching errors before you even run the JavaScript code.

TypeScript is an open-source programming language that builds on top of JavaScript. It works on any **browser**, any **OS**, and any **environment** that JavaScript runs.

### In this TypeScript tutorial, you'll learn the following:

- Why TypeScript offers a lot of benefits over vanilla JavaScript.
- Understand what TypeScript truly is and how it works under the hood.
- Use TypeScript and its rich features like Types, Classes, Interfaces, Modules, and much more.

## Prerequisites

### To follow the tutorial, you need to have the following:

- Basic knowledge of JavaScript.
- knowledge of OOP concepts.
- Knowledge of ECMAScript 2015 or ES6.

## Getting Started

### Introduction to TypeScript

TypeScript is a superset of JavaScript.

---

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.

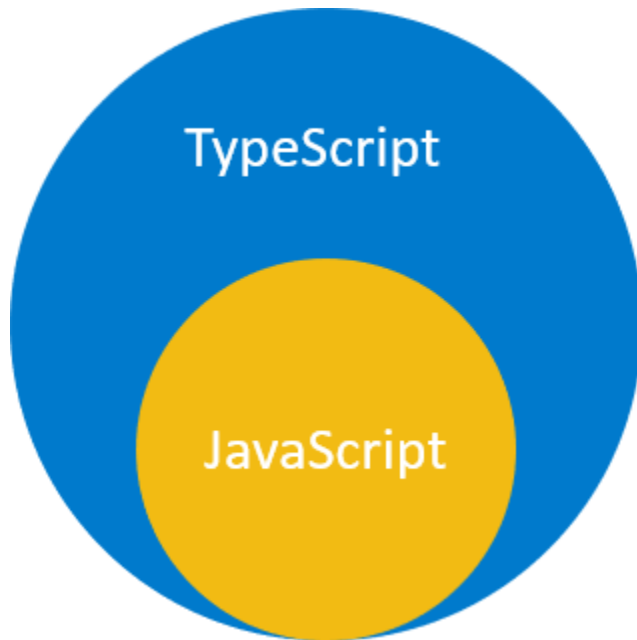


TypeScript uses the JavaScript syntaxes and adds additional syntaxes for supporting Types.

If you have a JavaScript program that doesn't have any syntax errors, it is also a TypeScript program. It means that all JavaScript programs are TypeScript programs. This is very helpful if you're migrating an existing JavaScript codebase to TypeScript.

The following diagram shows the relationship between TypeScript and JavaScript:





## Why TypeScript?

The main goals of TypeScript are:

- Introduce optional types to JavaScript.
- Implement planned features of future JavaScript, a.k.a. ECMAScript Next or ES Next to the current JavaScript.

### 1) TypeScript improves your productivity while helping avoid bugs

Types increase productivity by helping you avoid many mistakes. By using types, you can catch bugs at the compile time instead of having them occur at runtime.

The following function adds two numbers *x* and *y*:

```
function add(x, y) {  
  return x + y;  
}
```

If you get the values from HTML input elements and pass them into the function, you may get an unexpected result:

```
let result = add(input1.value, input2.value);  
console.log(result); // result of concatenating strings
```

For example, if users entered **10** and **20**, the **add()** function would return **1020**, instead of **30**.

---

The reason is that the `input1.value` and `input2.value` are **strings**, not **numbers**. When you use the operator `+` to add two **strings**, it concatenates them into a single **string**.

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

```
function add(x: number, y: number) {  
    return x + y;  
}
```

In this function, we added the number types to the parameters. The function `add()` will accept only numbers, not any other values.

When you invoke the function as follows:

```
let result = add(input1.value, input2.value);
```

the TypeScript compiler will issue an error if you compile the TypeScript code into JavaScript. Hence, you can prevent the error from happening at runtime.

## 2) TypeScript brings the future JavaScript to today

TypeScript supports the upcoming features planned in the ES Next for the current JavaScript engines. It means that you can use the new JavaScript features before web browsers (or other environments) fully support them.

Every year, TC39 releases several new features for ECMAScript, which is the standard of JavaScript. The feature proposals typically go through five stages:

- Stage 0: Strawperson
- Stage 1: Proposal
- Stage 2: Draft
- Stage 3: Candidate
- Stage 4: Finished

And TypeScript generally supports features that are in stage 3.

## TypeScript Environment Setup

In this tutorial, you'll learn how to set up a TypeScript development environment.

---

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

- Node.js – Node.js is the environment on 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. If you use JavaScript for node.js, you can install the ts-node module. It is a TypeScript execution and REPL for node.js
- Visual Studio Code or VS code – is a code editor that supports TypeScript. VS Code is highly recommended. However, you can use your favorite editor.

## Install Node.js

To install node.js, you follow these steps:

- Go to the [node.js download page](#).
- Download the node.js version that suits your platform i.e., Windows, macOS, or Linux.
- Execute the downloaded node.js package or execution file. The installation is quite straightforward.
- Verify the installation by opening the terminal on macOS and Linux or the command line on Windows and entering the command `node -v`. If you see the version that you downloaded, then you have successfully installed node.js on your computer.

## 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
```

It should return the version like this:

```
Version 4.0.2
```

Note that your version is probably newer than this version.

---

If you're on Windows and got the following error:

```
'tsc' is not recognized as an internal or external command,  
operable program, or batch file.
```

then you should add the following path `C:\Users\<user>\AppData\Roaming\npm` to the PATH variable. Notice that you should change the `<user>` to your windows user.

To install the ts-node module globally, you run the following command from the Terminal on macOS and Linux or Command Prompt on Windows:

```
npm install -g ts-node
```

## TypeScript Sample Program with node.js

First, create a new folder to store the code, e.g., `HelloWorld`.

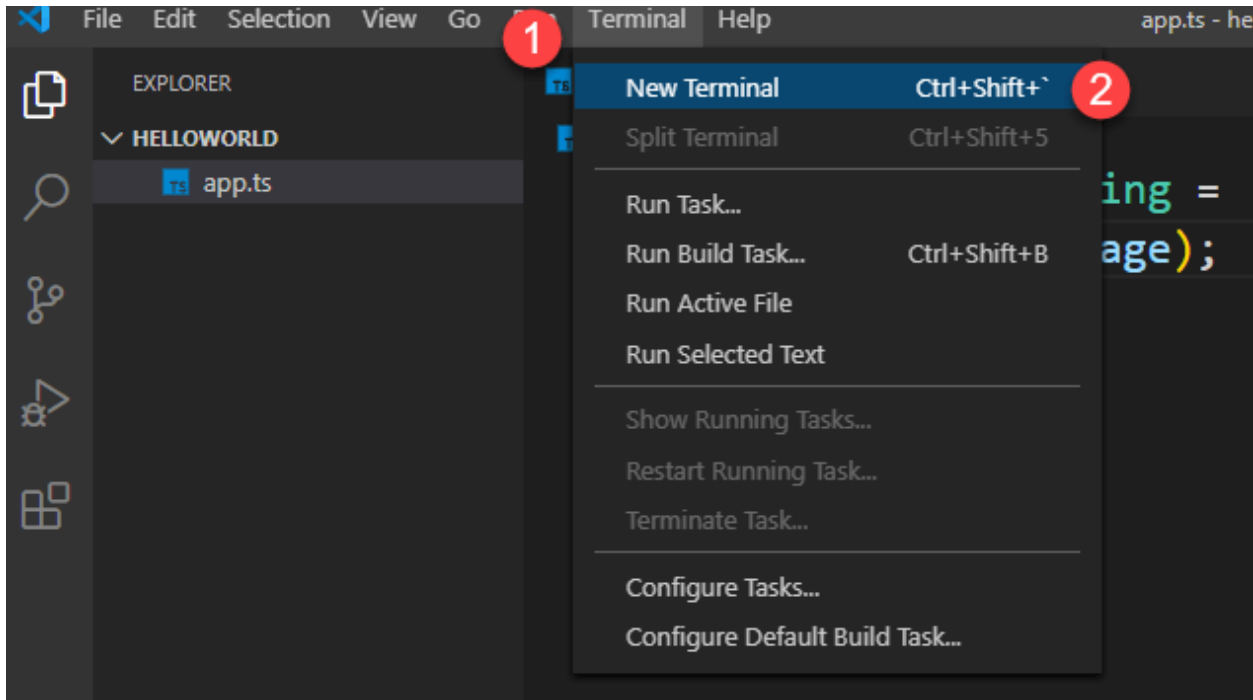
Second, launch VS Code and open that folder.

Third, 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:

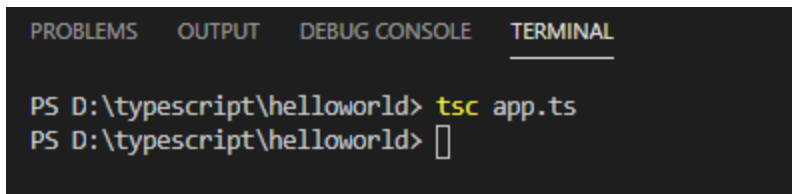
```
let message: string = 'Hello, World!';  
console.log(message);
```

Fifth, launch a new Terminal within the VS Code by using the keyboard shortcut **Ctrl+`** or follow the menu **Terminal > New Terminal**



Sixth, 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
```

If you installed the `ts-node` module mentioned in the [setting up the TypeScript development environment](#), you can use just one command to compile the TypeScript file and execute the output file in one shot:

```
ts-node app.ts
```

## TypeScript Sample Program with vanilla JavaScript

---

You follow these steps to create a webpage that shows the Hello, World! message on web browsers.

First, create a new file called **index.html** and include the **app.js** as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>TypeScript: Hello, World!</title>
</head>
<body>
  <script src="app.js"></script>
</body>
</html>
```

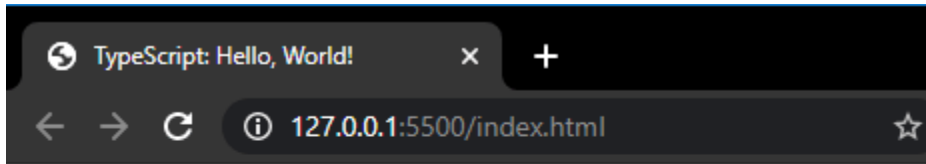
Second, change the **app.ts** code to the following:

```
let message: string = 'Hello, World!';
// create a new heading 1 element
let heading = document.createElement('h1');
heading.textContent = message;
// add the heading to the document
document.body.appendChild(heading);
```

Third, compile the app.ts file:

```
tsc app.ts
```

Fourth, open the **index.html** file in the browser.



# Hello, World!

To make the changes, you need to edit the **app.ts** file. For example:

```
let message: string = 'Hello, TypeScript!';

let heading = document.createElement('h1');
heading.textContent = message;

document.body.appendChild(heading);
```

And compile the **app.ts** file:

```
tsc app.ts
```

The TypeScript compiler will generate a new **app.js** file, and the Live Server will automatically reload it on the web browser.

Note that the **app.js** is the output file of the **app.ts** file, therefore, you should never directly change the code in this file, or you'll lose the changes once you recompile the **app.ts** file.

In this tutorial, you have learned how to create the first program in TypeScript called Hello, World! which works on **node.js** and **web browsers**.

## Basic Types

### TypeScript Types

in this tutorial, you'll learn about the TypeScript types and their purposes.

---

## What is a type in TypeScript

In TypeScript, a type is a convenient way to refer to the different **properties** and **functions** that a **value** has.

A value is anything that you can assign to a variable e.g., a number, a string, an array, an object, and a function.

See the following value:

```
'Hello'
```

When you look at this value, you can say that it's a string. And this value has properties and methods that a string has.

For example, the **'Hello'** value has a property called **length** that returns the number of characters:

```
console.log('Hello'.length); // 5
```

It also has many methods like **match()**, **indexOf()**, and **toLocaleUpperCase()**. For example:

```
console.log('Hello'.toLocaleUpperCase()); // HELLO
```

If you look at the value **'Hello'** and describe it by listing the properties and methods, it would be inconvenient.

A shorter way to refer to a value is to assign it a type. In this example, you say the **'Hello'** is a string. Then, you know that you can use the properties and methods of a string for the value **'Hello'**.

In conclusion, in TypeScript:

- a type is a label that describes the different properties and methods that a value has
- every value has a type.

## Types in TypeScript

TypeScript inherits the built-in types from JavaScript. TypeScript types are categorized into:



- 
- Primitive types
  - Object types

## Primitive types

The following illustrates the primitive types in TypeScript:

Name	Description
string	represents text data
number	represents numeric values
boolean	has true and false values
null	has one value: null
undefined	has one value: undefined. It is a default value of an uninitialized variable
symbol	that represents a unique constant value

## Object type

The object types are functions, arrays, classes, etc. Later, you'll learn how to create custom object types.

## Purposes of types in TypeScript

There are two main purposes of types in TypeScript:

- First, types are used by the TypeScript compiler to analyze your code for errors
- Second, types allow you to understand what values are associated with variables.

## Examples of TypeScript type

---

The following example uses the **querySelector()** method to select the **<h1>** element:

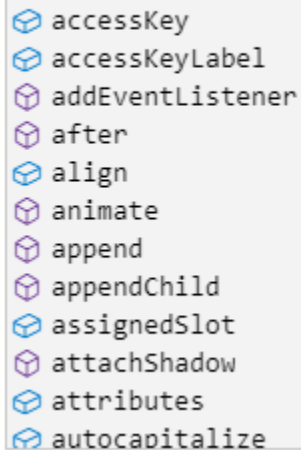
```
const heading = document.querySelector('h1');
```

The TypeScript compiler knows that the type of **heading** is **HTMLHeadingElement**:

```
const heading = document.querySelector('h1');  
const heading: HTMLHeadingElement | null
```

And it shows a list of methods of the **HTMLHeadingElement** type that the **heading** can access:

```
const heading = document.querySelector('h1');  
heading?.
```



- accessKey
- accessKeyLabel
- addEventListener
- after
- align
- animate
- append
- appendChild
- assignedSlot
- attachShadow
- attributes
- autocapitalize

If you try to access a property or method that doesn't exist, the TypeScript compiler will show an error. For example

```
const heading = document.querySelector('h1');  
heading.rotate()
```

## Summary

- Every value in TypeScript has a type.
- A type is a label that describes the properties and methods that a value has.
- TypeScript compiler uses types to analyze your code for hunting bugs and errors.

## Type Annotations

---

in this tutorial, you will learn about type annotations in TypeScript.

## What is Type Annotation in TypeScript?

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

TypeScript uses the **syntax**: type after an identifier as the type annotation, where the **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:

```
let variableName: type;  
let variableName: type = value;  
const constantName: 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** 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
```

Error:

```
Type '"Hello"' is not assignable to type 'number'.
```

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** type, the **age** variable gets the **number** type, and the active **variable** gets the **boolean** type.

## Type annotation examples

### Arrays

To annotate an 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

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 with its function type.

```
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

In this tutorial, you will learn about the Type inference in TypeScript.

Type inference describes where and how TypeScript infers types when you don't explicitly annotate them.

### Basic type inference

When you declare a variable, you can use a type annotation to explicitly specify a type for it. For example:

```
let counter: number;
```

However, if you initialize the **counter** variable to a number, TypeScript will infer the type the **counter** to be **number**. For example:

```
let counter = 0;
```

---

It is equivalent to the following statement:

```
let counter: number = 0;
```

Likewise, when you assign a function parameter a value, TypeScript **infers** the type of the parameter to the type of the default value. For example:

```
function setCounter(max=100) {  
    // ...  
}
```

In this example, TypeScript **infers** type of the max **parameter** to be **number**.

Similarly, 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**.

It uses the best common type algorithm to analyze each candidate type and select the type that is compatible with all other candidates.

In this case, TypeScript selects the number array type (**number[]**) as the best common type.

If you add a string to the items array, TypeScript will infer the type for the items as an array of numbers and strings: (**number | string**)[]

```
let items = [0, 1, null, 'Hi'];
```

When TypeScript cannot find the best common type, it returns the union array type. For example:

```
let arr = [new Date(), new RegExp('\d+')];
```

---

In this example, TypeScript infers the **type** for **arr** to be **(RegExp | Date)[]**.

## Contextual typing

TypeScript uses locations of variables to infer their types. This mechanism is known as contextual typing. For example:

```
document.addEventListener('click', function (event) {  
    console.log(event.button); //  
});
```

In this example, TypeScript knows that the event parameter is an instance of `MouseEvent` because of the click event.

However, when you change the click event to the scroll event, TypeScript will issue an error:

```
document.addEventListener('scroll', function (event) {  
    console.log(event.button); // compiler error  
});
```

Error:

```
Property 'button' does not exist on type 'Event'.(2339)
```

TypeScript knows that the event in this case, is an instance of `UIEvent`, not a `MouseEvent`. And `UIEvent` does not have the `button` property, therefore, TypeScript throws an error.

You will find contextual typing in many cases such as arguments to function calls, type assertions, members of objects and array literals, return statements, and right-hand sides of assignments.

## Type inference vs. Type annotations

The following show the difference between type inference and type annotations:

Type inference	Type annotations
TypeScript guesses the type	You explicitly tell TypeScript the type

So, when do you use type inference and type annotations?

In practice, you should always use the type inference as much as possible. And you use the type annotation in the following cases:

- When you declare a variable and assign it a value later.

- 
- When you want a variable that can't be inferred.
  - When a function returns **any** type and you need to clarify the value.

## Summary

- Type inference occurs when you initialize variables, set parameter default values, and determine function return types.
- TypeScript uses the best common type algorithm to select the best candidate types that are compatible with all variables.
- TypeScript also uses contextual typing to infer types of variables based on the locations of the variables.

## TypeScript Number

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:

### Decimal numbers

The following shows some decimal numbers:

```
let counter: number = 0;  
let x: number = 100,  
let y: number = 200;
```

### Binary Numbers



---

The binary number uses a leading zero followed by a lowercase or uppercase letter “B” e.g., 0b or 0B :

```
let bin = 0b100;
let anotherBin: number = 0B010;
```

Note that the digit after 0b or 0B must be 0 or 1.

## Octal Numbers

An octal number uses a leading zero followed the letter o (since ES2015) 0o. The digits after 0o are numbers in the range 0 through 7:

```
let octal: number = 0o10;
```

## Hexadecimal numbers

Hexadecimal numbers use a leading zero followed by a lowercase or uppercase letter X (0x or 0X). The digits after the 0x must be in the range (0123456789ABCDEF). For example:

```
let hexadecimal: number = 0XA;
```

JavaScript has the Number type (with the letter N in uppercase) that refers to the non-primitive boxed object. You should not use this Number type as much as possible in TypeScript.

## Big Integers

The big integers represent the whole numbers larger than  $2^{53} - 1$ . A Big integer literal has the n character at the end of an integer literal like this:

```
let big: bigint = 9007199254740991n;
```

## Summary

- All numbers in TypeScript are either floating-point values that get the number type or big integers that get the **bigint** type.
- Avoid using the Number type as much as possible.

## TypeScript String

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

In this tutorial, you will learn about the TypeScript boolean data type.

The TypeScript boolean type allows two values: true and false. It's one of the primitive types in TypeScript. For example:

```
let pending: boolean;
pending = true;
// after a while
// ..
pending = false;
```

---

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

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

### Introduction to TypeScript object type

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

The following are primitive types in TypeScript:

- number
- bigint
- string
- boolean
- null
- undefined
- 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'.
```

The employee object is an object type with a fixed list of properties. If you attempt to access a property that doesn't exist on the employee object, you'll get an error:

```
console.log(employee.hireDate);
```

Error:

```
error TS2339: Property 'hireDate' does not exist on type 'object'.
```

Note that the above statement works perfectly fine in JavaScript and returns undefined instead.

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 you assign the employee object to a literal object with the described properties:

```
employee = {
  firstName: 'John',
  lastName: 'Doe',
  age: 25,
  jobTitle: 'Web Developer'
};
```

Or you can combine both syntaxes in the same statement like this:

```
let employee: {
  firstName: string;
  lastName: string;
  age: number;
  jobTitle: string;
} = {
  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.

## The empty type {}

TypeScript has another type called empty type denoted by {}, which is quite similar to the object type.

The empty type {} describes an object that has no property on its own. If you try to access a property on a such an object, TypeScript will issue a compile-time error:

```
let vacant: {};
vacant.firstName = 'John';
```

Error:

```
error TS2339: Property 'firstName' does not exist on type '{}'.

```

But you can access all properties and methods declared on the Object type, which is available on the object via prototype chain:

```
let vacant: {} = {};
```

```
console.log(vacant.toString());
```

Output:

```
[object Object]
```

---

## Summary

- The TypeScript object type represents any value that is not a primitive value.
- The Object type, however, describes functionality that is available on all objects.
- The empty type {} refers to an object that has no property on its own.

## TypeScript Array Type

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:

```
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 Solving', 'Software Design', 'Programming'];
```

In this example, TypeScript infers the skills array as an array of strings. It is equivalent to the following:

```
let skills: string[];  
skills = ['Problem Solving', 'Software Design', 'Programming'];
```

Once you define an array of a specific type, TypeScript will prevent you from adding incompatible values to the array.

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 can do 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 the string as shown in the output.

## TypeScript array properties and methods

TypeScript arrays can access the properties and methods of a JavaScript. For example, the following uses the length property to get the number of element in an array:

```
let series = [1, 2, 3];  
console.log(series.length); // 3
```

And you can use all the useful array method such as `forEach()`, `map()`, `reduce()`, and `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 declare 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];
```

## Summary

- In TypeScript, an array is an ordered list of values. An array can store a mixed type of values.
- To declare an array of a specific type, you use the `let arr: type[]` syntax.

## TypeScript Tuple

in this tutorial, you'll learn about the TypeScript Tuple type and its usages.

### Introduction to TypeScript Tuple type

A tuple works like an array 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.

## Optional Tuple Elements

Since TypeScript 3.0, a tuple can have optional elements specified using the question mark (?) postfix.

For example, you can define an RGBA tuple with the optional alpha channel value:

```
let bgColor, headerColor: [number, number, number, number?];  
bgColor = [0, 255, 255, 0.5];  
headerColor = [0, 255, 255];
```

Note that the RGBA defines colors using the red, green, blue, and alpha models. The alpha specifies the opacity of the color.

## Summary

- A tuple is an array with a fixed number of elements whose types are known.

## TypeScript Enum

in this tutorial, you'll learn about the TypeScript enum type and how to use it effectively.

### 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.

## How TypeScript enum works

It is a good practice to use the constant values defined by enums in the code.

---

However, the following example passes a number instead of an enum to the `isItSummer()` function. And it works.

```
console.log(isItSummer(6)); // true
```

This example uses a number (6) instead of a constant defined by the `Month` enum. and it works.

Let's check the generated Javascript code of the `Month` enum:

```
var Month;
(function (Month) {
    Month[Month["Jan"] = 0] = "Jan";
    Month[Month["Feb"] = 1] = "Feb";
    Month[Month["Mar"] = 2] = "Mar";
    Month[Month["Apr"] = 3] = "Apr";
    Month[Month["May"] = 4] = "May";
    Month[Month["Jun"] = 5] = "Jun";
    Month[Month["Jul"] = 6] = "Jul";
    Month[Month["Aug"] = 7] = "Aug";
    Month[Month["Sep"] = 8] = "Sep";
    Month[Month["Oct"] = 9] = "Oct";
    Month[Month["Nov"] = 10] = "Nov";
    Month[Month["Dec"] = 11] = "Dec";
})(Month || (Month = {}));
```

And you can output the `Month` variable to the console:

```
{
  '0': 'Jan',
  '1': 'Feb',
  '2': 'Mar',
  '3': 'Apr',
  '4': 'May',
  '5': 'Jun',
  '6': 'Jul',
  '7': 'Aug',
  '8': 'Sep',
  '9': 'Oct',
  '10': 'Nov',
  '11': 'Dec',
  Jan: 0,
  Feb: 1,
  Mar: 2,
  Apr: 3,
  May: 4,
```

```
Jun: 5,  
Jul: 6,  
Aug: 7,  
Sep: 8,  
Oct: 9,  
Nov: 10,  
Dec: 11  
}
```

As you can see clearly from the output, a TypeScript enum is an object in JavaScript. This object has named properties declared in the enum. For example, Jan is 0 and Feb is 1.

The generated object also has number keys with string values representing the named constants.

That's why you can pass a number into the function that accepts an enum. In other words, an enum member is both a number and a defined constant.

## Specifying enum members' numbers

TypeScript defines the numeric value of an enum's member based on the order of that member that appears in the enum definition. For example, Jan takes 0, Feb gets 1, etc.

It's possible to explicitly specify numbers for the members of an enum like this:

```
enum Month {  
    Jan = 1,  
    Feb,  
    Mar,  
    Apr,  
    May,  
    Jun,  
    Jul,  
    Aug,  
    Sep,  
    Oct,  
    Nov,  
    Dec  
};
```

In this example, the Jan constant value takes 1 instead of 0. The Feb takes 2, and the Mar takes 3, etc.

## When to use an enum

You should use an enum when you:

- Have a small set of fixed values that are closely related
- And these values are known at compile time.

For example, you can use an enum for the approval status:

```
enum ApprovalStatus {  
    draft,  
    submitted,  
    approved,  
    rejected  
};
```

Then, you can use the ApprovalStatus enum like this:

```
const request = {  
    id: 1,  
    status: ApprovalStatus.approved,  
    description: 'Please approve this request'  
};
```

```
if(request.status === ApprovalStatus.approved) {  
    // send an email  
    console.log('Send email to the Applicant...');  
}
```

## Summary

- A TypeScript enum is a group of constant values.
- Under the hood, an enum is a JavaScript object with named properties declared in the enum definition.
- Do use an enum when you have a small set of fixed values that are closely related and known at compile time.

## TypeScript any Type

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 at the time of 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.

To do so, you use `any` type. Any type allows you to assign a value of any type to a variable:

```
// json may come from a third-party API
const json = `{"latitude": 10.11, "longitude":12.12}`;

// parse JSON to find location
const currentLocation = JSON.parse(json);
console.log(currentLocation);
```

Output:

```
{ latitude: 10.11, longitude: 12.12 }
```

In this example, the `currentLocation` variable is assigned to an object returned by the `JSON.parse()` function.

However, when you use the `currentLocation` to access object properties, TypeScript also won't carry any check:

```
console.log(currentLocation.x);
```

Output:

```
undefined
```

The TypeScript compiler doesn't complain or issue any error.

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`: implicit typing

If you declare a variable without specifying a type, TypeScript assumes that you use the `any` type. This feature is called type inference. Basically, TypeScript guesses the type of the variable. For example:

```
let result;
```

In this example, TypeScript infers the type for you. This practice is called implicit typing.

---

Note that to disable implicit typing to the any type, you change the noImplicitAny option in the tsconfig.json file to true. You'll learn more about the tsconfig.json in the later tutorial.

## TypeScript any vs. object

If you declare a variable with the object type, you can also assign it any value.

However, you cannot call a method on it even the method actually exists. For example:

```
let result: any;
result = 10.123;
console.log(result.toFixed());
result.willExist(); //
```

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.

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

```
let result: object;
result = 10.123;
result.toFixed();
```

Error:

```
error TS2339: Property 'toFixed' does not exist on type 'object'.
```

## Summary

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

## TypeScript never Type

In this tutorial, you will learn about the TypeScript never type that contains no value.

The never type is a type that contains no values. Because of this, you cannot assign any value to a variable with a never type.

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

---

```
function raiseError(message: string): never {
    throw new Error(message);
}
```

The return type of the following function is inferred to the never type:

```
function reject() {
    return raiseError('Rejected');
}
```

If you have a function expression that contains an indefinite loop, its return type is also the never type. For example:

```
let loop = function forever() {
    while (true) {
        console.log('Hello');
    }
}
```

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

If you see that the return type of a function is never, then you should ensure that it is not what you intended to do.

Variables can also acquire the never type when you narrow its type by a type guard that can never be true.

For example, without the never type, the following function causes an error because not all code paths return a value.

```
function fn(a: string | number): boolean {
    if (typeof a === "string") {
        return true;
    } else if (typeof a === "number") {
        return false;
    }
}
```

To make the code valid, you can return a function whose return type is the never type.

```
function fn(a: string | number): boolean {
    if (typeof a === "string") {
        return true;
    } else if (typeof a === "number") {
        return false;
    }
    // make the function valid
    return neverOccur();
}
```



```
let neverOccur = () => {  
  throw new Error('Never!');  
}
```

## Summary

- The never type contains no value.
- The never type represents the return type of a function that always throws an error or a function that contains an indefinite loop.

## TypeScript union Type

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.

In case 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 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 the 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');
}
```

## Summary

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

## TypeScript Type Aliases

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

### Introduction to TypeScript type aliases

Type aliases allows you to create a new name for an existing type. The following shows the syntax of the type alias:

```
type alias = existingType;
```

The existing type can be any valid TypeScript type.

The following example use the type alias `chars` for the string type:

```
type chars = string;
let message: chars; // same as string type
```

---

It's useful to create type aliases for union types. For example:

```
type alphanumeric = string | number;
let input: alphanumeric;
input = 100; // valid
input = 'Hi'; // valid
input = false; // Compiler error
```

## Summary

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

## TypeScript String Literal Types

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 in a variable.

The string literal types can combine nicely with the union types 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 very verbose.

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

```
type MouseEvent: 'click' | 'dblclick' | 'mouseup' | 'mousedown';
let mouseEvent: MouseEvent;
mouseEvent = 'click'; // valid
mouseEvent = 'dblclick'; // valid
mouseEvent = 'mouseup'; // valid
mouseEvent = 'mousedown'; // valid
mouseEvent = 'mouseover'; // compiler error

let anotherEvent: MouseEvent;
```

## 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.

## CONTROL FLOW STATEMENTS

### TypeScript if else

In this tutorial, you will learn about the TypeScript if...else statement.

#### TypeScript if statement

An if statement executes a statement based on a condition. If the condition is truthy, the if statement will execute the statements inside its body:

```
if(condition) {
    // if-statement
}
```

For example, the following statement illustrates how to use the if statement to increase the counter variable if its value is less than the value of the max constant:

```
const max = 100;
let counter = 0;

if (counter < max) {
    counter++;
}
```

```
console.log(counter); // 1
```

Output:

```
1
```

In this example, because the counter variable starts at zero, it is less than the max constant. The expression `counter < max` evaluates to true therefore the if statement executes the statement `counter++`.

Let's initialize the counter variable to 100:

```
const max = 100;
let counter = 100;

if (counter < max) {
  counter++;
}

console.log(counter); // 100
```

Output:

```
100
```

In this example, the expression `counter < max` evaluates to false. The if statement doesn't execute the statement `counter++`. Therefore, the output is 100.

## TypeScript if...else statement

If you want to execute other statements when the condition in the if statement evaluates to false, you can use the if...else statement:

```
if(condition) {
  // if-statements
} else {
  // else statements;
}
```

The following illustrates an example of using the if..else statement:

```
const max = 100;
let counter = 100;

if (counter < max) {
```

```
        counter++;
    } else {
        counter = 1;
    }

    console.log(counter);
```

Output:

```
1
```

In this example, the expression `counter < max` evaluates to false therefore the statement in the else branch executes that resets the counter variable to 1.

### Ternary operator ?:

In practice, if you have a simple condition, you can use the ternary operator ?: rather than the if...else statement to make code shorter like this:

```
const max = 100;
let counter = 100;

counter < max ? counter++ : counter = 1;

console.log(counter);
```

### TypeScript if...else if...else statement

When you want to execute code based on multiple conditions, you can use the if...else if...else statement.

The if...else if...else statement can have one or more else if branches but only one else branch.

For example:

```
let discount: number;
let itemCount = 11;

if (itemCount > 0 && itemCount <= 5) {
    discount = 5; // 5% discount
} else if (itemCount > 5 && itemCount <= 10) {
    discount = 10; // 10% discount
} else {
    discount = 15; // 15%
}
```

```
console.log(`You got ${discount}% discount. `)
```

Output:

```
0
```

This example used the if...elseif...else statement to determine the discount based on the number of items.

If the number of items from less than or equal 5, the discount is 5%. The statement in the if branch executes,

If the number of items is less than or equal to 10, the discount is 10%. The statement in the else if branch executes.

When the number of items is greater than 10, the discount is 15%. The statement in the else branch executes.

In this example, the assumption is that the number of items is always greater than zero. However, if the number of items is less than zero or greater than 10, the discount is 15%.

To make the code more robust, you can use another else if instead of the else branch like this:

```
let discount: number;
let itemCount = 11;

if (itemCount > 0 && itemCount <= 5) {
    discount = 5; // 5% discount
} else if (itemCount > 5 && itemCount <= 10) {
    discount = 10; // 10% discount
} else if (discount > 10) {
    discount = 15; // 15%
} else {
    throw new Error('The number of items cannot be negative!');
}

console.log(`You got ${discount}% discount. `)
```

In this example, only when the number of items is greater than 10, the discount is 15%. The statement in the second else if branch executes.

If the number of items is less than zero, the statement in the else branch executes.

## Summary

- Use if statement to execute code based on a condition.

- Use the else branch if you want to execute code when the condition is false. It's good practice to use the ternary operator `?:` instead of a simple if...else statement.
- Use if else if...else statement to execute code based on multiple conditions.

## TypeScript switch case

in this tutorial, you will about the TypeScript switch...case statement.

### Introduction to TypeScript switch case statement

The following shows the syntax of the switch...case statement:

```
switch ( expression ) {  
  case value1:  
    // statement 1  
    break;  
  case value2:  
    // statement 2  
    break;  
  case valueN:  
    // statement N  
    break;  
  default:  
    //  
    break;  
}
```

How it works:

First, the switch...case statement evaluates the expression.

Then, it searches for the first case clause whose expression evaluates to the same value as the value (value1, value2, ...valueN).

The switch...case statement will execute the statement in the first case clause whose value matches.

If no matching case clause is found, the switch...case statement looks for the optional default clause. If the default clause is available, it executes the statement in the default clause.



---

The break statement that associates with each case clause ensures that the control breaks out of the switch...case statement once the statements in the case clause complete.

If the matching case clause doesn't have the break statement, the program execution continues at the next statement in the switch...case statement.

By convention, the default clause is the last clause in the switch...case statement. However, it doesn't need to be so.

## TypeScript switch case statement examples

Let's take some examples of using the switch...case statement.

### 1) A simple TypeScript switch case example

The following example shows a simple switch...case example that shows a message based on the target Id:

```
let targetId = 'btnDelete';

switch (targetId) {
  case 'btnUpdate':
    console.log('Update');
    break;
  case 'btnDelete':
    console.log('Delete');
    break;
  case 'btnNew':
    console.log('New');
    break;
}
```

Output:

```
Delete
```

In this example, the targetId is set to btnDelete.

The switch...case statement compares the targetId with a list of values. Because the targetId matches the 'btnDelete' the statement in the corresponding case clause executes.

### 2) Grouping case example

If you have a code that is shared by multiple cases, you can group them. For example:

```
// change the month and year
let month = 2,
```

```
    year = 2020;

let day = 0;
switch (month) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    day = 31;
    break;
  case 4:
  case 6:
  case 9:
  case 11:
    day = 30;
    break;
  case 2:
    // leap year
    if (((year % 4 == 0) &&
        !(year % 100 == 0))
        || (year % 400 == 0))
      day = 29;
    else
      day = 28;
    break;
  default:
    throw Error('Invalid month');
}

console.log(`The month ${month} in ${year} has ${day} days`);
```

Output:

```
The month 2 in 2020 has 29 days
```

This example returns the days of a specific month and year.

---

If the month is 1,3, 5, 7, 8, 12, the number of days is 31. If the month is 4, 6, 9, or 11, the number of days is 30.

If the month is 2 and the year is a leap year, it returns 29 days, otherwise, it returns 28 days.

## TypeScript for

in this tutorial, you will learn about the TypeScript for loop statement that executes a piece of code repeatedly.

### Introduction to the TypeScript for statement

The following shows the syntax of the TypeScript for loop statement:

```
for(initialization; condition; expression) {  
    // statement  
}
```

The for loop statement creates a loop. It consists of three optional expressions separated by semicolons (;) and enclosed in parentheses:

- initialization: is an expression evaluated once before the loop begins. Typically, you use the initialization to initialize a loop counter.
- condition – is an expression that is evaluated at the end of each loop iteration. If the condition is true, the statements in the loop body execute.
- expression – is an expression that is evaluated before the condition is evaluated at the end of each loop iteration. Generally, you use the expression to update the loop counter.

All three expressions in the for loop statement are optional. It means that you can use the for loop statement like this:

```
for(;;) {  
    // do something  
}
```

In practice, you should use a for loop if you know how many times the loop should run. If you want to stop the loop based on a condition other than the number of times the loop executes, you should use a while loop.

TypeScript allows you to omit the loop body completely as follows:

```
for(initialization; condition; expression);
```

---

However, it is rarely used in practice because it makes the code more difficult to read and maintain.

## TypeScript for examples

Let's take some examples of using the TypeScript for loop statement.

### 1) Simple TypeScript for example

The following example uses the for loop statement to output 10 numbers from 0 to 9 to the console:

```
for (let i = 0; i < 10; i++) {  
    console.log(i);  
}
```

Output:

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

How it works:

- First, declare a variable `i` and initialize it to 0.
- Then check if `i` is less than 10. If it is, output `i` to the console and increment `i` by one.
- Finally, repeat the second step until `i` equals 10.

### 2) TypeScript for example: optional block

The following example shows the same output as the above example. However, the for doesn't have the initialization block:

```
let i = 0;  
for (; i < 10; i++) {
```

```
    console.log(i);  
}
```

Like the initialization block, you can omit the condition block.

However, you must escape the loop when a condition is met by using the `if` and `break` statements. Otherwise, you will create an infinite loop that causes the program to execute repeatedly until it is crashed.

```
for (let i = 0; ; i++) {  
    console.log(i);  
    if (i > 9) break;  
}
```

The following example illustrates a `for` loop that omits all three blocks:

```
let i = 0;  
for (; ;) {  
    console.log(i);  
    i++;  
    if (i > 9) break;  
}
```

Output:

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

How it works:

- First, declare a loop counter `i` and initialize it to 0 before entering the `for`.
- Then, in each loop iteration, output `i` to the console, increment it by one, and break out of the loop if `i` is greater than 9.

## Summary

- Use the TypeScript `for` statements when you want to execute a piece of code a number of times repeatedly.

---

## TypeScript while

in this tutorial, you will learn how to create a loop using the TypeScript while statement.

### Introduction to the TypeScript while statement

The while statement allows you to create a loop that executes a block of code as long as a condition is true.

The following shows the syntax of the TypeScript while statement:

```
while(condition) {  
    // do something  
}
```

The while statement evaluates the condition before each loop iteration.

If the condition evaluates to true, the while statement executes the code its in body surrounded by the curly braces ({}).

When the condition evaluates to false, the execution continues with the statement after the while statement.

Since the while statement evaluates the condition before its body is executed, a while loop is also called a pretest loop.

To break the loop immaturely based on another condition, you use the if and break statements:

```
while(condition) {  
    // do something  
    // ...  
  
    if(anotherCondition)  
        break;  
}
```

If you want to run a loop a number of times, you should use the TypeScript for statement.

### TypeScript while statement examples

Let's take some examples of using the TypeScript while statement.

#### TypeScript while: a simple example

The following example uses the while statement to output a number to the console as long as it is less than 5:

```
let counter = 0;

while (counter < 5) {
  console.log(counter);
  counter++;
}
```

Output:

```
0
1
2
3
4
```

How it works:

- First, declare a counter variable and initialize it to zero.
- Then, check if the counter is less than 5 before entering the loop. If it is, output the counter to the console and increments it by one.
- Finally, repeat the above step as long as counter is less than 5.

TypeScript while practical example

Let's say you have the following list element on an HTML document:

```
<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
</ul>
```

The following example shows how to use the while statement to remove all <li> element of the <ul> element:

```
let list = document.querySelector('#list');

while (list.firstChild) {
  list.removeChild(list.firstChild);
}
```

How it works:

- First, select the <ul> element by its id using the `querySelector()` method.

- 
- Then, check if firstChild of the list is available and remove it. Once the first child node is removed, the next child node is automatically promoted as the first child node. Therefore, the while statement removes all child nodes of the list element.

## Summary

- Use the TypeScript while statement to create a loop that will run as long as a condition is true.

## TypeScript do while

in this tutorial, you will learn how to use the do...while statement to create a loop that runs until a condition evaluates to false.

### Introduction to TypeScript do...while statement

The following shows the syntax of the do...while statement:

```
do {  
    // do something  
} while(condition);
```

The do...while statement executes statements in its body surrounded by the curly braces ({} ) until the condition is false.

The do...while statement always executes its loop body at least one.

Unlike the while statement, the do...while statement evaluates the condition after each loop iteration, therefore, it is called a post-test loop.

### TypeScript do...while statement example

The following example uses the do...while statement to output numbers from 0 to 9 to the console:

```
let i = 0;  
  
do {  
    console.log(i);  
    i++;  
} while (i < 10);
```

Output:



```
0
1
2
3
4
5
6
7
8
9
```

How it works:

- First, declare a variable `i` and initialize it to zero before entering the loop.
- Then, output `i` to the console, increment it by one, and check if it is less than 10. If it is, repeat the loop until `i` is greater than or equal to 10.

## Summary

- Use the `do...while` statement to create a loop that runs until a condition evaluates to false.

## TypeScript break

in this tutorial, you will learn about the TypeScript `break` statement to terminate a loop or a switch.

### Using TypeScript `break` to terminate a loop

The `break` statement allows you to terminate a loop and pass the program control over the next statement after the loop.

You can use the `break` statement inside the `for`, `while`, and `do...while` statements.

The following example shows how to use the `break` statement inside a `for` loop:

```
let products = [
  { name: 'phone', price: 700 },
  { name: 'tablet', price: 900 },
  { name: 'laptop', price: 1200 }
];

for (var i = 0; i < products.length; i++) {
  if (products[i].price == 900)
```

```
        break;
    }

    // show the products
    console.log(products[i]);
}
```

Output:

```
{ name: 'tablet', price: 900 }
```

How it works:

- First, initialize a list of products with name and price properties.
- Then, search for the product whose price is 900 and terminate the loop once the product is found by using the break statement.
- Finally, show the matching product to the console.

## Using the break statement to break a switch

The following example returns the discount of a specified product. It uses the break statement to break out of a switch:

```
let products = [
  { name: 'phone', price: 700 },
  { name: 'tablet', price: 900 },
  { name: 'laptop', price: 1200 }
];

let discount = 0;
let product = products[1];

switch (product.name) {
  case 'phone':
    discount = 5;
    break;
  case 'tablet':
    discount = 10;
    break;
  case 'laptop':
    discount = 15;
    break;
}

console.log(`There is a ${discount}% on ${product.name}.`);
```

---

Note that besides a loop or a switch, the break statement can be used to break out of a labeled statement. However, it is rarely used in practice so we don't cover it in this tutorial.

## Summary

- Use a break statement to terminate a loop or switch.

## Functions

### TypeScript Functions

In this tutorial, you will learn about the TypeScript functions and how to use type annotations to enforce the type checks for functions.

#### Introduction to TypeScript functions

TypeScript functions are the building blocks of readable, maintainable, and reusable code.

Like JavaScript, you use the function keyword to declare a function in TypeScript:

```
function name(parameter: type, parameter: type, ...): returnType {  
    // do something  
}
```

Unlike JavaScript, TypeScript allows you to use type annotations in parameters and return value of a function.

Let's see the following add() function example:

```
function add(a: number, b: number): number {  
    return a + b;  
}
```

In this example, the add() function accepts two parameters with the number type.

When you call the add() function, the TypeScript compiler will check each argument passed to the function to ensure that they are numbers.

In the add() function example, you can only pass numbers into it, not the values of other types.

The following code will result in an error because it passes two strings instead of two numbers into the add() function:

```
let sum = add('10', '20');
```

---

Error:

```
error TS2345: Argument of type '"10"' is not assignable to parameter of type 'number'
```

The types of the function parameters are also available within the function body for type checking.

The `: number` after the parentheses indicate the return type. The `add()` function returns a value of the number type in this case.

When a function has a return type, TypeScript compiler checks every return statement against the return type to ensure that the return value is compatible with it.

If a function does not return a value, you can use the `void` type as the return type. The `void` keyword indicates that the function doesn't return any value. For example:

```
function echo(message: string): void {  
    console.log(message.toUpperCase());  
}
```

The `void` prevents the code inside the function from returning a value and stops the calling code from assigning the result of the function to a variable.

When you do not annotate the return type, TypeScript will try to infer an appropriate type. For example:

```
function add(a: number, b: number) {  
    return a + b;  
}
```

In this example, the TypeScript compiler tries to infer the return type of the `add()` function to the number type, which is expected.

However, if a function has different branches that return different types, the TypeScript compiler may infer the union type or any type.

Therefore, it is important to add type annotations to a function as much as possible.

## Summary

- Use type annotations for function parameters and return type to keep the calling code inline and ensure the type checking within the function body.

## TypeScript Function Types

---

in this tutorial, you will learn about the TypeScript function types that allow you to define types for functions.

## Introduction to TypeScript function types

A function type has two parts: parameters and return type. When declaring a function type, you need to specify both parts with the following syntax:

```
(parameter: type, parameter: type, ...) => type
```

The following example shows how to declare a variable which has a function type that accepts two numbers and returns a number:

```
let add: (x: number, y: number) => number;
```

In this example:

- The function type accepts two arguments: x and y with the type number.
- The type of the return value is number that follows the fat arrow (=>) appeared between parameters and return type.

Note that the parameter names (x and y) are just for readability purposes. As long as the types of parameters match, it is a valid type for the function.

Once annotating a variable with a function type, you can assign the function with the same type to the variable.

TypeScript compiler will match the number of parameters with their types and the return type.

The following example shows how to assign a function to the add variable:

```
add = function (x: number, y: number) {  
    return x + y;  
};
```

Also, you can declare a variable and assign a function to a variable like this:

```
let add: (a: number, b: number) => number =  
    function (x: number, y: number) {  
        return x + y;  
    };
```

If you assign other functions whose type doesn't match to the add variable, TypeScript will issue an error:

```
add = function (x: string, y: string): number {  
    return x.concat(y).length;  
};
```

```
};
```

In this example, we reassigned a function, whose type doesn't match, to the add function variable.

## Inferring function types

TypeScript compiler can figure out the function type when you have the type on one side of the equation. This form of type inference is called contextual typing. For example:

```
let add = function (x: number, y: number): number {  
    return x + y;  
};  
  
let result = add(10, 20);
```

```
let add: (x: number, y: number) => number
```

In this example, the add function will take the type (x: number, y: number) => number.

By using the type inference, you can significantly reduce the amount of code with annotations.

## TypeScript Optional Parameters

In this tutorial, you will learn how to use the TypeScript optional parameters for functions.

In JavaScript, you can call a function without passing any arguments even though the function specifies parameters. Therefore, JavaScript supports the optional parameters by default.

In TypeScript, the compiler checks every function call and issues an error in the following cases:

- The number of arguments is different from the number of parameters specified in the function.
- Or the types of arguments are not compatible with the types of function parameters.

Because the compiler thoroughly checks the passing arguments, you need to annotate optional parameters to instruct the compiler not to issue an error when you omit the arguments.

To make a function parameter optional, you use the ? after the parameter name. For example:

```
function multiply(a: number, b: number, c?: number): number {  
  
    if (typeof c !== 'undefined') {  
        return a * b * c;  
    }  
}
```

```
    return a * b;
}
```

How it works:

- First, use the ? after the c parameter.
- Second, check if the argument is passed to the function by using the expression `typeof c !== 'undefined'`.

Note that if you use the expression `if(c)` to check if an argument is not initialized, you would find that the empty string or zero would be treated as undefined.

The optional parameters must appear after the required parameters in the parameter list.

For example, if you make the b parameter optional, and c parameter required the TypeScript compiler will issue an error:

```
function multiply(a: number, b?: number, c: number): number {
    if (typeof c !== 'undefined') {
        return a * b * c;
    }
    return a * b;
}
```

Error:

```
error TS1016: A required parameter cannot follow an optional
parameter.
```

## Summary

- Use the `parameter?:` type syntax to make a parameter optional.
- Use the expression `typeof(parameter) !== 'undefined'` to check if the parameter has been initialized.

## TypeScript Default Parameters

In this tutorial, you will learn about TypeScript default parameters.

### Introduction to TypeScript default parameters

JavaScript supported default parameters since ES2015 (or ES6) with the following syntax:

```
function name(parameter1=defaultValue1,...) {  
  // do something  
}
```

In this syntax, if you don't pass arguments or pass the undefined into the function when calling it, the function will take the default initialized values for the omitted parameters. For example:

```
function applyDiscount(price, discount = 0.05) {  
  return price * (1 - discount);  
}  
  
console.log(applyDiscount(100)); // 95
```

In this example, the applyDiscount() function has the discount parameter as a default parameter.

When you don't pass the discount argument into the applyDiscount() function, the function uses a default value which is 0.05.

Similar to JavaScript, you can use default parameters in TypeScript with the same syntax:

```
function name(parameter1:type=defaultvalue1, parameter2:type=defaultvalue2,...) {  
  //  
}
```

The following example uses default parameters for the applyDiscount() function:

```
function applyDiscount(price: number, discount: number = 0.05): number {  
  return price * (1 - discount);  
}  
  
console.log(applyDiscount(100)); // 95
```

Notice that you cannot include default parameters in function type definitions. The following code will result in an error:

```
let promotion: (price: number, discount: number = 0.05) => number;
```

Error:

```
error TS2371: A parameter initializer is only allowed in a function or constructor  
implementation.
```

## Default parameters and Optional parameters

Like optional parameters, default parameters are also optional. It means that you can omit the default parameters when calling the function.

In addition, both the default parameters and trailing default parameters share the same type. For example, the following function:



---

```
function applyDiscount(price: number, discount: number = 0.05): number {  
    // ...  
}
```

and

```
function applyDiscount(price: number, discount?: number): number {  
    // ...  
}
```

share the same type:

```
(price: number, discount?: number) => number
```

Optional parameters must come after the required parameters. However, default parameters don't need to appear after the required parameters.

When a default parameter appears before a required parameter, you need to explicitly pass undefined to get the default initialized value.

The following function returns the number of days in a specified month and year:

```
function getDay(year: number = new Date().getFullYear(), month: number): number {  
    let day = 0;  
    switch (month) {  
        case 1:  
        case 3:  
        case 5:  
        case 7:  
        case 8:  
        case 10:  
        case 12:  
            day = 31;  
            break;  
        case 4:  
        case 6:  
        case 9:  
        case 11:  
            day = 30;  
            break;  
        case 2:  
            // leap year  
            if ((year % 4 == 0) &&  
                !(year % 100 == 0) &&  
                || (year % 400 == 0))  
                day = 29;  
            else  
                day = 28;  
            break;  
        default:  
            throw Error('Invalid month');  
    }  
    return day;  
}
```

```
}  
  return day;  
}
```

In this example, the default value of the year is the current year if you don't pass an argument or pass the undefined value.

The following example uses the `getDay()` function to get the number of days in Feb 2019:

```
let day = getDay(2019, 2);  
console.log(day); // 28
```

To get the number of days in Feb of the current year, you need to pass undefined to the year parameter like this:

```
let day = getDay(undefined, 2);  
console.log(day);
```

## Summary

- Use default parameter syntax `parameter:=defaultValue` if you want to set the default initialized value for the parameter.
- Default parameters are optional.
- To use the default initialized value of a parameter, you omit the argument when calling the function or pass the undefined into the function.

## TypeScript Rest Parameters

In this tutorial, you will learn about the TypeScript rest parameters that allow you to represent an indefinite number of arguments as an array.

A rest parameter allows you a function to accept zero or more arguments of the specified type. In TypeScript, rest parameters follow these rules:

- A function has only one rest parameter.
- The rest parameter appears last in the parameter list.
- The type of the rest parameter is an array type.

To declare a rest parameter, you prefix the parameter name with three dots and use the array type as the type annotation:

```
function fn(...rest: type[]) {
```

```
//...  
}
```

The following example shows how to use the rest parameter:

```
function getTotal(...numbers: number[]): number {  
    let total = 0;  
    numbers.forEach((num) => total += num);  
    return total;  
}
```

In this example, the `getTotal()` calculates the total of numbers passed into it.

Since the `numbers` parameter is a rest parameter, you can pass one or more numbers to calculate the total:

```
console.log(getTotal()); // 0  
console.log(getTotal(10, 20)); // 30  
console.log(getTotal(10, 20, 30)); // 60
```

In this tutorial, you have learned about the TypeScript rest parameters that allow you to represent an indefinite number of arguments as an array.

## TypeScript Function Overloadings

in this tutorial, you will learn about function overloadings in TypeScript.

### Introduction to TypeScript function overloadings

In TypeScript, function overloadings allow you to establish the relationship between the parameter types and result types of a function.

Note that TypeScript function overloadings are different from the function overloadings supported by other statically-typed languages such as C# and Java.

Let's start with some simple functions:

```
function addNumbers(a: number, b: number): number {  
    return a + b;  
}  
  
function addStrings(a: string, b: string): string {  
    return a + b;  
}
```

In this example:

- The `addNumbers()` function returns the sum of two numbers.

- The `addStrings()` function returns the concatenation of two strings.

It's possible to use a union type to define a range of types for function parameters and results:

```
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 + b;  
}
```

However, the union type doesn't express the relationship between the parameter types and results accurately.

The `add()` function tells the compiler that it will accept either numbers or strings and return a number or string. It fails to describe that the function returns a number when the parameters are numbers and return a string if the parameters are strings.

To better describe the relationships between the types used by a function, TypeScript supports function overloads. For example:

```
function add(a: number, b: number): number;  
function add(a: string, b: string): string;  
function add(a: any, b: any): any {  
    return a + b;  
}
```

In this example, we added two overloads to the `add()` function. The first overload tells the compiler that when the arguments are numbers, the `add()` function should return a number. The second overload does the same but for a string.

Each function overload defines a combination of types supported by the `add()` function. It describes the mapping between the parameters and the result they return.

Now, when you call the `add()` function, the code editor suggests that there is an overload function available as shown in the following picture:

```
function add(a: number, b: number): number (+1 overload)  
let result = add(10, 20);
```

## Function overloading with optional parameters

---

When you overload a function, the number of required parameters must be the same. If an overload has more parameters than the other, you have to make the additional parameters optional. For example:

```
function sum(a: number, b: number): number;
function sum(a: number, b: number, c: number): number;
function sum(a: number, b: number, c?: number): number {
    if (c) return a + b + c;
    return a + b;
}
```

The `sum()` function accepts either two or three numbers. The third parameter is optional. If you don't make it optional, you will get an error.

## Method overloading

When a function is a property of a class, it is called a method. TypeScript also supports method overloading. For example:

```
class Counter {
    private current: number = 0;
    count(): number;
    count(target: number): number[];
    count(target?: number): number | number[] {
        if (target) {
            let values = [];
            for (let start = this.current; start <= target; start++) {
                values.push(start);
            }
            this.current = target;
            return values;
        }
        return ++this.current;
    }
}
```

The `count()` function can return a number or an array depending on the number of argument that you pass into it:

```
let counter = new Counter();

console.log(counter.count()); // return a number
console.log(counter.count(20)); // return an array
```

Output:

```
1
[
  1, 2, 3, 4, 5, 6, 7,
```

```
8, 9, 10, 11, 12, 13, 14,  
15, 16, 17, 18, 19, 20  
]
```

## Summary

- TypeScript function overloads allow you to describe the relationship between parameter types and results of a function.

## Class

### TypeScript Class

in this tutorial, you will learn about the TypeScript Class.

### Introduction to the TypeScript Class

JavaScript does not have a concept of class like other programming languages such as Java and C#. In ES5, you can use a constructor function and prototype inheritance to create a “class”.

For example, to create a Person class that has three properties ssn, first name, and last name, you use the following constructor function:

```
function Person(ssn, firstName, lastName) {  
  this.ssn = ssn;  
  this.firstName = firstName;  
  this.lastName = lastName;  
}
```

Next, you can define a prototype method to get the full name of the person by concatenating first name and last name like this:

```
Person.prototype.getFullName = function () {  
  return `${this.firstName} ${this.lastName}`;  
}
```

Then, you can use the Person “class” by creating a new object:

```
let person = new Person('171-28-0926', 'John', 'Doe');  
console.log(person.getFullName());
```

It would output the following to the console:

```
John Doe
```

---

ES6 allowed you to define a class which is simply syntactic sugar for creating constructor function and prototypal inheritance:

```
class Person {
  ssn;
  firstName;
  lastName;

  constructor(ssn, firstName, lastName) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

In the class syntax, the constructor is clearly defined and placed inside the class. The following adds the `getFullName()` method:

```
class Person {
  ssn;
  firstName;
  lastName;

  constructor(ssn, firstName, lastName) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  }
}
```

Using the `Person` class is the same as the `Person` constructor function:

```
let person = new Person('171-28-0926', 'John', 'Doe');
console.log(person.getFullName());
```

TypeScript class adds type annotations to the properties and methods of the class. The following shows the `Person` class in TypeScript:

```
class Person {
  ssn: string;
  firstName: string;
  lastName: string;

  constructor(ssn: string, firstName: string, lastName: string) {
    this.ssn = ssn;
    this.firstName = firstName;
  }
}
```

```
        this.lastName = lastName;
    }

    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

When you annotate types to properties, constructor, and method, TypeScript compiler will carry the corresponding type checks.

For example, you cannot initialize the `ssn` with a number. The following code will result in an error:

```
let person = new Person(171280926, 'John', 'Doe');
```

## Summary

- Use `class` keyword to define a class in TypeScript.
- TypeScript leverages the ES6 class syntax and adds type annotations to make the class more robust.

## TypeScript Access Modifiers

In this tutorial, you will learn about the access modifiers in TypeScript.

Access modifiers change the visibility of the properties and methods of a class. TypeScript provides three access modifiers:

- `private`
- `protected`
- `public`

Note that TypeScript controls the access logically during compilation time, not at runtime.

### The private modifier

The `private` modifier limits the visibility to the same-class only. When you add the `private` modifier to a property or method, you can access that property or method within the same class. Any attempt to access private properties or methods outside the class will result in an error at compile time.



---

The following example shows how to use the private modifier to the `ssn`, `firstName`, and `lastName` properties of the `Person` class:

```
class Person {
  private ssn: string;
  private firstName: string;
  private lastName: string;
  // ...
}
```

Once the private property is in place, you can access the `ssn` property in the constructor or methods of the `Person` class. For example:

```
class Person {
  private ssn: string;
  private firstName: string;
  private lastName: string;

  constructor(ssn: string, firstName: string, lastName: string) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }
}
```

The following attempts to access the `ssn` property outside the class:

```
let person = new Person('153-07-3130', 'John', 'Doe');
console.log(person.ssn); // compile error
```

## The public modifier

The public modifier allows class properties and methods to be accessible from all locations. If you don't specify any access modifier for properties and methods, they will take the public modifier by default.

For example, the `getFullName()` method of the `Person` class has the public modifier. The following explicitly adds the public modifier to the `getFullName()` method:

```
class Person {
  // ...
  public getFullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }
  // ...
}
```

---

It has the same effect as if the `public` keyword were omitted.

## The protected modifier

The protected modifier allows properties and methods of a class to be accessible within same class and within subclasses.

When a class (child class) inherits from another class (parent class), it is a subclass of the parent class.

The TypeScript compiler will issue an error if you attempt to access the protected properties or methods from anywhere else.

To add the protected modifier to a property or a method, you use the protected keyword. For example:

```
class Person {  
    protected ssn: string;  
    // other code  
}
```

The `ssn` property now is protected. It will be accessible within the `Person` class and in any class that inherits from the `Person` class. You'll learn more about inheritance [here](#).

The `Person` class declares the two private properties and one protected property. Its constructor initializes these properties to three arguments.

To make the code shorter, TypeScript allows you to both declare properties and initialize them in the constructor like this:

```
class Person {  
    constructor(protected ssn: string, private firstName: string, private lastName: string) {  
        this.ssn = ssn;  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
}
```

When you consider the visibility of properties and methods, it is a good practice to start with the least visible access modifier, which is `private`.

## Summary

- TypeScript provides three access modifiers to class properties and methods: private, protected, and public.
- The private modifier allows access within the same class.
- The protected modifier allows access within the same class and subclasses.
- The public modifier allows access from any location.

## TypeScript readonly

In this tutorial, you will learn how to use the TypeScript readonly access modifier to mark class properties as immutable property.

TypeScript provides the readonly modifier that allows you to mark the properties of a class immutable. The assignment to a readonly property can only occur in one of two places:

- In the property declaration.
- In the constructor of the same class.

To mark a property as immutable, you use the readonly keyword. The following shows how to declare a readonly property in the Person class:

```
class Person {
    readonly birthDate: Date;

    constructor(birthDate: Date) {
        this.birthDate = birthDate;
    }
}
```

In this example, the birthdate property is a readonly property that is initialized in the constructor of the Person class.

The following attempts to reassign the birthDate property that results in an error:

```
let person = new Person(new Date(1990, 12, 25));
person.birthDate = new Date(1991, 12, 25); // Compile error
```

Like other access modifiers, you can consolidate the declaration and initialization of a readonly property in the constructor like this:

```
class Person {
    constructor(readonly birthDate: Date) {
        this.birthDate = birthDate;
    }
}
```

```
}
```

## Readonly vs. const

The following shows the differences between readonly and const:

	readonly	const
Use for	Class properties	Variables
Initialization	In the declaration or in the constructor of the same class	In the declaration

## Summary

- Use the readonly access modifier to mark a class property as immutable.
- A readonly property must be initialized as a part of the declaration or in the constructor of the same class.

## TypeScript Inheritance

in this tutorial, you'll learn about the TypeScript inheritance concept and how to use it to reuse the functionality of another class.

### Introduction to the TypeScript inheritance

A class can reuse the properties and methods of another class. This is called inheritance in TypeScript.

The class which inherits properties and methods is called the child class. And the class whose properties and methods are inherited is known as the parent class. These names come from the nature that children inherit genes from their parents.

Inheritance allows you to reuse the functionality of an existing class without rewriting it.

JavaScript uses prototypal inheritance, not classical inheritance like Java or C#. ES6 introduces the class syntax that is simply the syntactic sugar of the prototypal inheritance. TypeScript supports inheritance like ES6.

Suppose you have the following Person class:

```
class Person {
```

```

    constructor(private firstName: string, private lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
    describe(): string {
        return `This is ${this.firstName} ${this.lastName}.`;
    }
}

```

To inherit a class, you use the extends keyword. For example, the following Employee class inherits the Person class:

```

class Employee extends Person {
    //..
}

```

In this example, the Employee is a child class and the Person is the parent class.

## Constructor

Because the Person class has a constructor that initializes the firstName and lastName properties, you need to initialize these properties in the constructor of the Employee class by calling its parent class' constructor.

To call the constructor of the parent class in the constructor of the child class, you use the super() syntax. For example:

```

class Employee extends Person {
    constructor(
        firstName: string,
        lastName: string,
        private jobTitle: string) {

        // call the constructor of the Person class:
        super(firstName, lastName);
    }
}

```

The following creates an instance of the Employee class:

```

let employee = new Employee('John', 'Doe', 'Front-end Developer');

```

Because the Employee class inherits properties and methods of the Person class, you can call the getFullName() and describe() methods on the employee object as follows:

```

let employee = new Employee('John', 'Doe', 'Web Developer');

```

```
console.log(employee.getFullName());
console.log(employee.describe());
```

Output:

```
John Doe
This is John Doe.
```

## Method overriding

When you call the `employee.describe()` method on the `employee` object, the `describe()` method of the `Person` class is executed that shows the message: `This is John Doe`.

If you want the `Employee` class has its own version of the `describe()` method, you can define it in the `Employee` class like this:

```
class Employee extends Person {
  constructor(
    firstName: string,
    lastName: string,
    private jobTitle: string) {

    super(firstName, lastName);
  }

  describe(): string {
    return super.describe() + `I'm a ${this.jobTitle}.`;
  }
}
```

In the `describe()` method, we called the `describe()` method of the parent class using the syntax `super.methodInParentClass()`.

If you call the `describe()` method on the `employee` object, the `describe()` method in the `Employee` class is invoked:

```
let employee = new Employee('John', 'Doe', 'Web Developer');
console.log(employee.describe());
```

Output:

```
This is John Doe.I'm a Web Developer.
```

## Summary

- Use the `extends` keyword to allow a class to inherit from another class.

- Use `super()` in the constructor of the child class to call the constructor of the parent class. Also, use the `super.methodInParentClass()` syntax to invoke the `methodInParentClass()` in the method of the child class.

## TypeScript Static Methods and Properties

in this tutorial, you will learn about the TypeScript static properties and methods.

### Static properties

Unlike an instance property, a static property is shared among all instances of a class.

To declare a static property, you use the `static` keyword. To access a static property, you use the `className.propertyName` syntax. For example:

```
class Employee {
  static headcount: number = 0;

  constructor(
    private firstName: string,
    private lastName: string,
    private jobTitle: string) {

    Employee.headcount++;
  }
}
```

In this example, the `headcount` is a static property that initialized to zero. Its value is increased by 1 whenever a new object is created.

The following creates two `Employee` objects and shows the value of the `headcount` property. It returns two as expected.

```
let john = new Employee('John', 'Doe', 'Front-end Developer');
let jane = new Employee('Jane', 'Doe', 'Back-end Developer');

console.log(Employee.headcount); // 2
```

### Static methods

Similar to the static property, a static method is also shared across instances of the class. To declare a static method, you use the `static` keyword before the method name. For example:

```
class Employee {
  private static headcount: number = 0;

  constructor(
    private firstName: string,
```

```
private lastName: string,
private jobTitle: string) {

    Employee.headcount++;
}

public static getHeadcount() {
    return Employee.headcount;
}
}
```

In this example:

- First, change the access modifier of the headcount static property from public to private so that its value cannot be changed outside of the class without creating a new Employee object.
- Second, add the getHeadcount() static method that returns the value of the headcount static property.

To call a static method, you use the className.staticMethod() syntax. For example:

```
let john = new Employee('John', 'Doe', 'Front-end Developer');
let jane = new Employee('Jane', 'Doe', 'Back-end Developer');

console.log(Employee.getHeadcount); // 2
```

In practice, you will find the library that contains many static properties and methods like the Math object. It has PI, E, ... static properties and abs(), round(), etc., static methods.

## Summary

- Static properties and methods are shared by all instances of a class.
- Use the static keyword before a property or a method to make it static.

## TypeScript Abstract Classes

in this tutorial, you will learn about TypeScript abstract classes.

### Introduction to TypeScript abstract classes



---

An abstract class is typically used to define common behaviors for derived classes to extend. Unlike a regular class, an abstract class cannot be instantiated directly.

To declare an abstract class, you use the `abstract` keyword:

```
abstract class Employee {  
    //...  
}
```

Typically, an abstract class contains one or more abstract methods.

An abstract method does not contain implementation. It only defines the signature of the method without including the method body. An abstract method must be implemented in the derived class.

The following shows the `Employee` abstract class that has the `getSalary()` abstract method:

```
abstract class Employee {  
    constructor(private firstName: string, private lastName: string) {  
    }  
    abstract getSalary(): number  
    get fullName(): string {  
        return `${this.firstName} ${this.lastName}`;  
    }  
    compensationStatement(): string {  
        return `${this.fullName} makes ${this.getSalary()} a month.`;  
    }  
}
```

In the `Employee` class:

- The constructor declares the `firstName` and `lastName` properties.
- The `getSalary()` method is an abstract method. The derived class will implement the logic based on the type of employee.
- The `getFullName()` and `compensationStatement()` methods contain detailed implementation. Note that the `compensationStatement()` method calls the `getSalary()` method.

Because the `Employee` class is abstract, you cannot create a new object from it. The following statement causes an error:

```
let employee = new Employee('John', 'Doe');
```

Error:

```
error TS2511: Cannot create an instance of an abstract class.
```

---

The following FullTimeEmployee class inherits from the Employee class:

```
class FullTimeEmployee extends Employee {
    constructor(firstName: string, lastName: string, private salary: number) {
        super(firstName, lastName);
    }
    getSalary(): number {
        return this.salary;
    }
}
```

In this FullTimeEmployee class, the salary is set in the constructor. Because the getSalary() is an abstract method of the Employee class, the FullTimeEmployee class needs to implement this method. In this example, it just returns the salary without any calculation.

The following shows the Contractor class that also inherits from the Employee class:

```
class Contractor extends Employee {
    constructor(firstName: string, lastName: string, private rate: number, private
hours: number) {
        super(firstName, lastName);
    }
    getSalary(): number {
        return this.rate * this.hours;
    }
}
```

In the Contractor class, the constructor initializes the rate and hours. The getSalary() method calculates the salary by multiplying the rate with the hours.

The following first creates a FullTimeEmployee object and a Contractor object and then shows the compensation statements to the console:

```
let john = new FullTimeEmployee('John', 'Doe', 12000);
let jane = new Contractor('Jane', 'Doe', 100, 160);

console.log(john.compensationStatement());
console.log(jane.compensationStatement());
```

Output:

```
John Doe makes 12000 a month.
Jane Doe makes 16000 a month.
```

It's a good practice to use abstract classes when you want to share code among some related classes.

## Summary

- 
- Abstract classes cannot be instantiated.
  - An Abstract class has at least one abstract method.
  - To use an abstract class, you need to inherit it and provide the implementation for the abstract methods.

## Interface

### TypeScript Interface

in this tutorial, you'll learn about TypeScript interfaces and how to use them to enforce type checking.

#### Introduction to TypeScript interfaces

TypeScript interfaces define the contracts within your code. They also provide explicit names for type checking.

Let's start with a simple example:

```
function getFullName(person: {  
  firstName: string;  
  lastName: string  
) {  
  return `${person.firstName} ${person.lastName}`;  
}  
  
let person = {  
  firstName: 'John',  
  lastName: 'Doe'  
};  
  
console.log(getFullName(person));
```

Output:

```
John Doe
```

In this example, the TypeScript compiler checks the argument that you pass into the `getFullName()` function.

If the argument has two properties whose types are string, then the TypeScript compiler passes the check. Otherwise, it'll issue an error.

---

As you can see clearly from the code, the type annotation of the function argument makes the code difficult to read.

To solve this, TypeScript introduces the concept of interfaces.

The following uses an interface called Person that has two string properties:

```
interface Person {  
    firstName: string;  
    lastName: string;  
}
```

By convention, the interface names are in the camel case. They use a single capitalized letter to separate words in their names. For example, Person, UserProfile, and FullName.

After defining the Person interface, you can use it as a type. And you can annotate the function parameter with the interface name:

```
function getFullName(person: Person) {  
    return `${person.firstName} ${person.lastName}`;  
}  
  
let john = {  
    firstName: 'John',  
    lastName: 'Doe'  
};  
  
console.log(getFullName(john));
```

The code now is easier to read than before.

The getFullName() function will accept any argument that has two string properties. And it doesn't have to have exactly two string properties. See the following example:

The following code declares an object that has four properties:

```
let jane = {  
    firstName: 'Jane',  
    middleName: 'K.',  
    lastName: 'Doe',  
    age: 22  
};
```

Since the jane object has two string properties firstName and lastName, you can pass it into the getFullName() function as follows:

```
let fullName = getFullName(jane);  
console.log(fullName); // Jane Doe
```

---

## Optional properties

An interface may have optional properties. To declare an optional property, you use the question mark (?) at the end of the property name in the declaration, like this:

```
interface Person {
  firstName: string;
  middleName?: string;
  lastName: string;
}
```

In this example, the Person interface has two required properties and one optional property.

And the following shows how to use the Person interface in the getFullName() function:

```
function getFullName(person: Person) {
  if (person.middleName) {
    return `${person.firstName} ${person.middleName} ${person.lastName}`;
  }
  return `${person.firstName} ${person.lastName}`;
}
```

## Readonly properties

If properties should be modifiable only when the object first created, you can use the readonly keyword before the name of the property:

```
interface Person {
  readonly ssn: string;
  firstName: string;
  lastName: string;
}

let person: Person;
person = {
  ssn: '171-28-0926',
  firstName: 'John',
  lastName: 'Doe'
}
```

In this example, the ssn property cannot be changed:

```
person.ssn = '171-28-0000';
```

Error:

```
error TS2540: Cannot assign to 'ssn' because it is a read-only property.
```

## Function types

---

In addition to describing an object with properties, interfaces also allow you to describe function types.

To describe a function type, you assign the interface to the function signature that contains the parameter list with types and returned types. For example:

```
interface StringFormat {  
    (str: string, isUpper: boolean): string  
}
```

Now, you can use this function type interface.

The following illustrates how to declare a variable of a function type and assign it a function value of the same type:

```
let format: StringFormat;  
  
format = function (str: string, isUpper: boolean) {  
    return isUpper ? str.toLocaleUpperCase() : str.toLocaleLowerCase();  
};  
  
console.log(format('hi', true));
```

Output:

```
HI
```

Note that the parameter names don't need to match the function signature. The following example is equivalent to the above example:

```
let format: StringFormat;  
  
format = function (src: string, upper: boolean) {  
    return upper ? src.toLocaleUpperCase() : src.toLocaleLowerCase();  
};  
  
console.log(format('hi', true));
```

The StringFormat interface ensures that all the callers of the function that implements it pass in the required arguments: a string and a boolean.

The following code also works perfectly fine even though the lowerCase is assigned to a function that doesn't have the second argument:

```
let lowerCase: StringFormat;  
lowerCase = function (str: string) {  
    return str.toLowerCase();  
}
```

```
console.log(lowerCase('Hi', false));
```

Notice that the second argument is passed when the `lowerCase()` function is called.

## Class Types

If you have worked with Java or C#, you can find that the main use of the interface is to define a contract between unrelated classes.

For example, the following `Json` interface can be implemented by any unrelated classes:

```
interface Json {  
    toJSON(): string  
}
```

The following declares a class that implements the `Json` interface:

```
class Person implements Json {  
    constructor(private firstName: string,  
                private lastName: string) {  
    }  
    toJSON(): string {  
        return JSON.stringify(this);  
    }  
}
```

In the `Person` class, we implemented the `toJSON()` method of the `Json` interface.

The following example shows how to use the `Person` class:

```
let person = new Person('John', 'Doe');  
console.log(person.toJSON());
```

Output:

```
{"firstName":"John","lastName":"Doe"}
```

## Summary

- TypeScript interfaces define contracts in your code and provide explicit names for type checking.
- Interfaces may have optional properties or readonly properties.
- Interfaces can be used as function types.
- Interfaces are typically used as class types that make a contract between unrelated classes.

---

## TypeScript Extend Interfaces

in this tutorial, you will learn how to extend an interface that allows you to copy properties and methods of one interface to another.

### Interfaces extending one interface

Suppose that you have an interface called Mailable that contains two methods called send() and queue() as follows:

```
interface Mailable {  
    send(email: string): boolean  
    queue(email: string): boolean  
}
```

And you have many classes that already implemented the Mailable interface.

Now, you want to add a new method to the Mailable interface that sends an email later like this:

```
later(email: string, after: number): void
```

However, adding the later() method to the Mailable interface would break the current code.

To avoid this, you can create a new interface that extends the Mailable interface:

```
interface FutureMailable extends Mailable {  
    later(email: string, after: number): boolean  
}
```

To extend an interface, you use the extends keyword with the following syntax:

```
interface A {  
    a(): void  
}  
  
interface B extends A {  
    b(): void  
}
```

The interface B extends the interface A, which then have both methods a() and b() .

Like classes, the FutureMailable interface inherits the send() and queue() methods from the Mailable interface.

The following shows how to implement the FutureMailable interface:

```
class Mail implements FutureMailable {  
    later(email: string, after: number): boolean {
```



```

        console.log(`Send email to ${email} in ${after} ms.`);
        return true;
    }
    send(email: string): boolean {
        console.log(`Sent email to ${email} after ${after} ms.`);
        return true;
    }
    queue(email: string): boolean {
        console.log(`Queue an email to ${email}.`);
        return true;
    }
}

```

## Interfaces extending multiple interfaces

An interface can extend multiple interfaces, creating a combination of all the interfaces. For example:

```

interface C {
    c(): void
}

interface D extends B, C {
    d(): void
}

```

In this example, the interface D extends the interfaces B and C. So D has all the methods of B and C interfaces, which are a(), b(), and c() methods.

## Interfaces extending classes

TypeScript allows an interface to extend a class. In this case, the interface inherits the properties and methods of the class. Also, the interface can inherit the private and protected members of the class, not just the public members.

It means that when an interface extends a class with private or protected members, the interface can only be implemented by that class or subclasses of that class from which the interface extends.

By doing this, you restrict the usage of the interface to only class or subclasses of the class from which the interface extends. If you attempt to implement the interface from a class that is not a subclass of the class that the interface inherited, you'll get an error. For example:

```

class Control {
    private state: boolean;
}

```

```

interface StatefulControl extends Control {
    enable(): void
}

class Button extends Control implements StatefulControl {
    enable() { }
}
class TextBox extends Control implements StatefulControl {
    enable() { }
}
class Label extends Control { }

// Error: cannot implement
class Chart implements StatefulControl {
    enable() { }
}

```

## Summary

- An interface can extend one or multiple existing interfaces.
- An interface also can extend a class. If the class contains private or protected members, the interface can only be implemented by the class or subclasses of that class.

## Advanced Types

### TypeScript Intersection Types

in this tutorial, you will learn about the TypeScript intersection types.

#### Introduction to TypeScript intersection types

An intersection type creates a new type by combining multiple existing types. The new type has all features of the existing types.

To combine types, you use the & operator as follows:

```

type typeAB = typeA & typeB;

```

The typeAB will have all properties from both typeA and typeB.

Note that the union type uses the | operator that defines a variable which can hold a value of either typeA or typeB

---

```
let varName = typeA | typeB; // union type
```

Suppose that you have three interfaces: BusinessPartner, Identity, and Contact.

```
interface BusinessPartner {  
  name: string;  
  credit: number;  
}  
  
interface Identity {  
  id: number;  
  name: string;  
}  
  
interface Contact {  
  email: string;  
  phone: string;  
}
```

The following defines two intersection types:

```
type Employee = Identity & Contact;  
type Customer = BusinessPartner & Contact;
```

The Employee type contains all properties of the Identity and Contact type:

```
type Employee = Identity & Contact;  
  
let e: Employee = {  
  id: 100,  
  name: 'John Doe',  
  email: 'john.doe@example.com',  
  phone: '(408)-897-5684'  
};
```

And the Customer type contains all properties of the BusinessPartner and Contact type:

```
type Customer = BusinessPartner & Contact;  
  
let c: Customer = {  
  name: 'ABC Inc.',  
  credit: 1000000,  
  email: 'sales@abcinc.com',  
  phone: '(408)-897-5735'
```

```
};
```

Later, if you want to implement employee sales, you can create a new intersection type that contains all properties of Identity, Contact, and BusinessPartner types:

```
type Employee = Identity & BusinessPartner & Contact;

let e: Employee = {
  id: 100,
  name: 'John Doe',
  email: 'john.doe@example.com',
  phone: '(408)-897-5684',
  credit: 1000
};
```

Notice both BusinessPartner and Identity have the property name with the same type. If they do not, then you will have an error.

## Type Order

When you intersect types, the order of the types doesn't matter. For example:

```
type typeAB = typeA & typeB;
type typeBA = typeB & typeA;
```

In this example, typeAB and typeBA have the same properties.

## Summary

- An intersection type combines two or more types to create a new type that has all properties of the existing types.
- The type order is not important when you combine types.

## TypeScript Type Guards

In this tutorial, you will learn about the Type Guard in TypeScript

Type Guards allow you to narrow down the type of a variable within a conditional block.

### typeof

Let's take a look at the following example:

```

type alphanumeric = string | number;

function add(a: alphanumeric, b: alphanumeric) {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    }

    if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b);
    }

    throw new Error('Invalid arguments. Both arguments must be either
numbers or strings.');
```

How it works:

- First, define the alphanumeric type that can hold either a string or a number.
- Next, declare a function that adds two variables a and b with the type of alphanumeric.
- Then, check if both types of arguments are numbers using the typeof operator. If yes, then calculate the sum of arguments using the + operator.
- After that, check if both types of arguments are strings using the typeof operator. If yes, then concatenate two arguments.
- Finally, throw an error if arguments are neither numbers nor strings.

In this example, TypeScript knows the usage of the typeof operator in the conditional blocks. Inside the following [if](#) block, TypeScript realizes that a and b are numbers.

```

if (typeof a === 'number' && typeof b === 'number') {
    return a + b;
}
```

Similarly, in the following if block, TypeScript treats a and b as strings, therefore, you can concatenate them into one:

```

if (typeof a === 'string' && typeof b === 'string') {
    return a.concat(b);
}
```

---

## instanceof

Similar to the `typeof` operator, TypeScript is also aware of the usage of the `instanceof` operator. For example:

```
class Customer {
    isCreditAllowed(): boolean {
        // ...
        return true;
    }
}

class Supplier {
    isInShortList(): boolean {
        // ...
        return true;
    }
}

type BusinessPartner = Customer | Supplier;

function signContract(partner: BusinessPartner) : string {
    let message: string;
    if (partner instanceof Customer) {
        message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' : 'Credit issue';
    }

    if (partner instanceof Supplier) {
        message = partner.isInShortList() ? 'Sign a new contract the supplier' : 'Need to evaluate further';
    }

    return message;
}
```

How it works:

- 
- First, declare the Customer and Supplier classes.  
Second, create a type alias BusinessPartner which is a union type of Customer and Supplier.
  - Third, declare a function signContract() that accepts a parameter with the type BusinessPartner.
  - Finally, check if the partner is an instance of Customer or Supplier, and then provide the respective logic.

Inside the following if block, TypeScript knows that the partner is an instance of the Customer type due to the instanceof operator:

```
if (partner instanceof Customer) {  
    message = partner.isCreditAllowed() ? 'Sign a new contract with the customer' :  
    'Credit issue';  
}
```

Likewise, TypeScript knows that the partner is an instance of Supplier inside the following if block:

```
if (partner instanceof Supplier) {  
    message = partner.isInShortList() ? 'Sign a new contract with the supplier' : 'Need  
to evaluate further';  
}
```

When an if narrows out one type, TypeScript knows that within the else it is not that type but the other. For example:

```
function signContract(partner: BusinessPartner) : string {  
    let message: string;  
    if (partner instanceof Customer) {  
        message = partner.isCreditAllowed() ? 'Sign a new contract  
with the customer' : 'Credit issue';  
    } else {  
        // must be Supplier  
        message = partner.isInShortList() ? 'Sign a new contract with  
the supplier' : 'Need to evaluate further';  
    }  
    return message;  
}
```

in

---

The `in` operator carries a safe check for the existence of a property on an object. You can also use it as a type guard. For example:

```
function signContract(partner: BusinessPartner) : string {
    let message: string;
    if ('isCreditAllowed' in partner) {
        message = partner.isCreditAllowed() ? 'Sign a new contract
with the customer' : 'Credit issue';
    } else {
        // must be Supplier
        message = partner.isInShortList() ? 'Sign a new contract the
supplier ' : 'Need to evaluate further';
    }
    return message;
}
```

## User-defined Type Guards

User-defined type guards allow you to define a type guard or help TypeScript infer a type when you use a function.

A user-defined type guard function is a function that simply returns `arg is aType`. For example:

```
function isCustomer(partner: any): partner is Customer {
    return partner instanceof Customer;
}
```

In this example, the `isCustomer()` is a user-defined type guard function. Now you can use it in as follows:

```
function signContract(partner: BusinessPartner): string {
    let message: string;
    if (isCustomer(partner)) {
        message = partner.isCreditAllowed() ? 'Sign a new contract
with the customer' : 'Credit issue';
    } else {
        message = partner.isInShortList() ? 'Sign a new contract with
the supplier' : 'Need to evaluate further';
    }

    return message;
}
```



---

## Summary

- Type guards narrow down the type of a variable within a conditional block.
- Use the `typeof` and `instanceof` operators to implement type guards in the conditional blocks

## Type Casting

In this tutorial, you will learn about type castings in TypeScript, which allow you to convert a variable from one type to another type.

JavaScript doesn't have a concept of type casting because variables have dynamic types. However, every variable in TypeScript has a type. Type castings allow you to convert a variable from one type to another.

In TypeScript, you can use the `as` keyword or `<>` operator for type castings.

### Type casting using the `as` keyword

The following selects the first input element by using the `querySelector()` method:

```
let input = document.querySelector('input[type="text"]');
```

Since the returned type of the `document.querySelector()` method is the `Element` type, the following code causes a compiler error:

```
console.log(input.value);
```

The reason is that the `value` property doesn't exist in the `Element` type. It only exists on the `HTMLInputElement` type.

To resolve this, you can use type casting that cast the `Element` to `HTMLInputElement` by using the `as` keyword like this:

```
let input = document.querySelector('input[type="text"]') as HTMLInputElement;
```

Now, the `input` variable has the type `HTMLInputElement`. So accessing its `value` property won't cause any error. The following code works:

```
console.log(input.value);
```

Another way to cast the `Element` to `HTMLInputElement` is when you access the property as follows:

---

```
let enteredText = (input as HTMLInputElement).value;
```

Note that the HTMLInputElement type extends the HTMLElement type that extends to the Element type. When you cast the HTMLElement to HTMLInputElement, this type casting is also known as a down casting.

It's also possible to carry an down casting. For example:

```
let e1: HTMLElement;  
e1 = new HTMLInputElement();
```

In this example, the e1 variable has the HTMLElement type. And you can assign it an instance of HTMLInputElement type because the HTMLInputElement type is an subclass of the HTMLElement type.

The syntax for converting a variable from typeA to typeB is as follows:

```
let a: typeA;  
let b = a as typeB;
```

## Type Casting using the <> operator

Besides the as keyword, you can use the <> operator to carry a type casting. For example:

```
let input = <HTMLInputElement>document.querySelector('input[type="text"]');
```

```
console.log(input.value);
```

The syntax for type casting using the <> is:

```
let a: typeA;  
let b = <typeB>a;
```

## Summary

- Type casting allows you to convert a variable from one type to another.
- Use the as keyword or <> operator for type castings.

## Type Assertions

in this tutorial, you will learn about type assertions in TypeScript.

---

## Introduction to Type Assertions in TypeScript

Type assertions instruct the TypeScript compiler to treat a value as a specified type. It uses the `as` keyword to do so:

```
expression as targetType
```

A type assertion is also known as type narrowing. It allows you to narrow a type from a union type. Let's see the following simple function:

```
function getNetPrice(price: number, discount: number, format: boolean): number | string {
    let netPrice = price * (1 - discount);
    return format ? `>${netPrice}` : netPrice;
}
```

The `getNetPrice()` function accepts `price`, `discount`, and `format` arguments and returns a value of the union type `number | string`.

If the `format` is `true`, the `getNetPrice()` returns a formatted net price as a string. Otherwise, it returns the net price as a number.

The following uses the `as` keyword to instruct the compiler that the value assigned to the `netPrice` is a string:

```
let netPrice = getNetPrice(100, 0.05, true) as string;
console.log(netPrice);
```

Output:

```
$95
```

Similarly, the following uses the `as` keyword to instruct the compiler that the returned value of the `getNetPrice()` function is a number.

```
let netPrice = getNetPrice(100, 0.05, false) as number;
console.log(netPrice);
```

Output:

```
95
```

Note that a type assertion does not carry any type casting. It only tells the compiler which type it should apply to a value for the type checking purposes.

## The alternative Type Assertion syntax

---

You can also use the angle bracket syntax `<>` to assert a type, like this:

```
<targetType> value
```

For example:

```
let netPrice = <number>getNetPrice(100, 0.05, false);
```

Note that you cannot use angle bracket syntax `<>` with some libraries such as React. For this reason, you should use the `as` keyword for type assertions.

## Summary

- Type assertions instruct the compiler to treat a value as a specified type.
- Type assertions do not carry any type conversion.
- Type assertions use the `as` keyword or an angle bracket `<>` syntax.

## Generics

### TypeScript Generics

In this tutorial, you'll learn about TypeScript generics that allow you to use types as formal parameters.

#### Introduction to TypeScript Generics

TypeScript generics allow you to write the reusable and generalized form of functions, classes, and interfaces. In this tutorial, you're focusing on developing generic functions.

It'll be easier to explain TypeScript generics through a simple example.

Suppose you need to develop a function that returns a random element in an array of numbers.

The following `getRandomNumberElement()` function takes an array of numbers as its parameter and returns a random element from the array:

```
function getRandomNumberElement(items: number[]): number {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}
```

To get a random element of an array, you need to:

- Find the random index first.
- Get the random element based on the random index.

To find the random index of an array, we used the `Math.random()` that returns a random number between 0 and 1, multiplied it with the length of the array, and applied the `Math.floor()` on the result.

The following shows how to use the `getRandomNumberElement()` function:

```
let numbers = [1, 5, 7, 4, 2, 9];
console.log(getRandomNumberElement(numbers));
```

Assuming that you need to get a random element from an array of strings. This time, you may come up with a new function:

```
function getRandomStringElement(items: string[]): string {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

The logic of the `getRandomStringElement()` function is the same as the one in the `getRandomNumberElement()` function.

This example shows how to use the `getRandomStringElement()` function:

```
let colors = ['red', 'green', 'blue'];
console.log(getRandomStringElement(colors));
```

Later you may need to get a random element in an array of objects. Creating a new function every time you want to get a random element from a new array type is not scalable.

## Using the any type

One solution for this issue is to set the type of the array argument as `any[]`. By doing this, you need to write just one function that works with an array of any type.

```
function getRandomAnyElement(items: any[]): any {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

The `getRandomAnyElement()` works with an array of the any type including an array of numbers, strings, objects, etc.:

```
let numbers = [1, 5, 7, 4, 2, 9];
let colors = ['red', 'green', 'blue'];
```

---

```
console.log(getRandomAnyElement(numbers));
console.log(getRandomAnyElement(colors));
```

This solution works fine. However, it has a drawback.

It doesn't allow you to enforce the type of the returned element. In other words, it isn't type-safe.

A better solution to avoid code duplication while preserving the type is to use generics.

## TypeScript generics come to rescue

The following shows a generic function that returns the random element from an array of type T:

```
function getRandomElement<T>(items: T[]): T {
  let randomIndex = Math.floor(Math.random() * items.length);
  return items[randomIndex];
}
```

This function uses type variable T. The T allows you to capture the type that is provided at the time of calling the function. Also, the function uses the T type variable as its return type.

This getRandomElement() function is generic because it can work with any data type including string, number, objects.

By convention, we use the letter T as the type variable. However, you can freely use other letters such as A, B C.

## Calling a generic function

The following shows how to use the getRandomElement() with an array of numbers:

```
let numbers = [1, 5, 7, 4, 2, 9];
let randomEle = getRandomElement<number>(numbers);
console.log(randomEle);
```

This example explicitly passes number as the T type into the getRandomElement() function.

In practice, you'll use type inference for the argument. It means that you let the TypeScript compiler set the value of T automatically based on the type of argument that you pass into, like this:

```
let numbers = [1, 5, 7, 4, 2, 9];
let randomEle = getRandomElement(numbers);
console.log(randomEle);
```

In this example, we didn't pass the number type to the getRandomElement() explicitly. The compiler just looks at the argument and sets T to its type.

---

Now, the `getRandomElement()` function is also type-safe. For example, if you assign the returned value to a string variable, you'll get an error:

```
let numbers = [1, 5, 7, 4, 2, 9];
let returnElem: string;
returnElem = getRandomElement(numbers); // compiler error
```

## Generic functions with multiple types

The following illustrates how to develop a generic function with two type variables `U` and `V`:

```
function merge<U, V>(obj1: U, obj2: V) {
  return {
    ...obj1,
    ...obj2
  };
}
```

The `merge()` function merges two objects with the type `U` and `V`. It combines the properties of the two objects into a single object.

Type inference infers the returned value of the `merge()` function as an intersection type of `U` and `V`, which is `U & V`

The following illustrates how to use the `merge()` function that merges two objects:

```
let result = merge(
  { name: 'John' },
  { jobTitle: 'Frontend Developer' }
);

console.log(result);
```

Output:

```
{ name: 'John', jobTitle: 'Frontend Developer' }
```

## Benefits of TypeScript generics

The following are benefits of TypeScript generics:

- Leverage type checks at the compile time.
- Eliminate type castings.
- Allow you to implement generic algorithms.

---

## Summary

- Use TypeScript generics to develop reusable, generalized, and type-safe functions, interfaces, and classes.

## TypeScript Generic Constraints

in this tutorial, you'll learn about the generic constraints in TypeScript.

### Introduction to generic constraints in TypeScript

Consider the following example:

```
function merge<U, V>(obj1: U, obj2: V) {  
    return {  
        ...obj1,  
        ...obj2  
    };  
}
```

The merge() is a generic function that merges two objects. For example:

```
let person = merge(  
    { name: 'John' },  
    { age: 25 }  
);  
  
console.log(result);
```

Output:

```
{ name: 'John', age: 25 }
```

It works perfectly fine.

The merge() function expects two objects. However, it doesn't prevent you from passing a non-object like this:

```
let person = merge(  
    { name: 'John' },  
    25  
);  
  
console.log(person);
```



---

Output:

```
{ name: 'John' }
```

TypeScript doesn't issue any error.

Instead of working with all types, you may want to add a constraint to the `merge()` function so that it works with objects only.

To do this, you need to list out the requirement as a constraint on what `U` and `V` types can be.

In order to denote the constraint, you use the `extends` keyword. For example:

```
function merge<U extends object, V extends object>(obj1: U, obj2: V) {  
  return {  
    ...obj1,  
    ...obj2  
  };  
}
```

Because the `merge()` function is now constrained, it will no longer work with all types. Instead, it works with the `object` type only.

The following will result in an error:

```
let person = merge(  
  { name: 'John' },  
  25  
);
```

Error:

```
Argument of type '25' is not assignable to parameter of type 'object'.
```

## Using type parameters in generic constraints

TypeScript allows you to declare a type parameter constrained by another type parameter.

The following `prop()` function accepts an object and a property name. It returns the value of the property.

```
function prop<T, K>(obj: T, key: K) {  
  return obj[key];  
}
```

The compiler issues the following error:

```
Type 'K' cannot be used to index type 'T'.
```

---

To fix this error, you add a constraint to K to ensure that it is a key of T as follows:

```
function prop<T, K extends keyof T>(obj: T, key: K) {  
    return obj[key];  
}
```

If you pass into the prop function a property name that exists on the obj, the compiler won't complain. For example:

```
let str = prop({ name: 'John' }, 'name');  
console.log(str);
```

Output:

```
John
```

However, if you pass a key that doesn't exist on the first argument, the compiler will issue an error:

```
let str = prop({ name: 'John' }, 'age');
```

Error:

```
Argument of type '"age"' is not assignable to parameter of type '"name"'.
```

## Summary

- Use extends keyword to constrain the type parameter to a specific type.
- Use extends keyof to constrain a type that is the property of another object.

## TypeScript Generic Interfaces

in this tutorial, you will learn how to develop TypeScript generic interfaces.

### Introduction to TypeScript generic interfaces

Like classes, interfaces also can be generic. A generic interface has generic type parameter list in an angle brackets <> following the name of the interface:

```
interface interfaceName<T> {  
    // ...  
}
```

This make the type parameter T visible to all members of the interface.

The type parameter list can have one or multiple types. For example:

---

```
interface interfaceName<U,V> {  
    // ...  
}
```

## TypeScript generic interface examples

Let's take some examples of declaring generic interfaces.

### 1) Generic interfaces that describe object properties

The following show how to declare a generic interface that consists of two members key and value with the corresponding types K and V:

```
interface Pair<K, V> {  
    key: K;  
    value: V;  
}
```

Now, you can use the Pair interface for defining any key/value pair with any type. For example:

```
let month: Pair<string, number> = {  
    key: 'Jan',  
    value: 1  
};  
  
console.log(month);
```

In this example, we declare a month key-value pair whose key is a string and value is a number.

### 2) Generic interfaces that describe methods

The following declares a generic interface with two methods add() and remove():

```
interface Collection<T> {  
    add(o: T): void;  
    remove(o: T): void;  
}
```

And this List<T> generic class implements the Collection<T> generic interface:

```
class List<T> implements Collection<T>{  
    private items: T[] = [];  
  
    add(o: T): void {  
        this.items.push(o);  
    }  
    remove(o: T): void {  
        let index = this.items.indexOf(o);
```

```

        if (index > -1) {
            this.items.splice(index, 1);
        }
    }
}

```

From the `List<T>` class, you can create a list of values of the various type e.g., numbers, or strings.

For example, the following shows how to use the `List<T>` generic class to create a list of numbers:

```

let list = new List<number>();
for (let i = 0; i < 10; i++) {
    list.add(i);
}

```

### 3) Generic interfaces that describe index types

The following declare an interface that describes an index type:

```

interface Options<T> {
    [name: string]: T
}

let inputOptions: Options<boolean> = {
    'disabled': false,
    'visible': true
};

```

In this tutorial, you have learned about the TypeScript generic interfaces.

## TypeScript Generic Classes

in this tutorial, you will learn how to develop TypeScript generic classes.

### Introduction to TypeScript generic classes

A generic class has a generic type parameter list in an angle brackets `<>` that follows the name of the class:

```

class className<T>{
    //...
}

```

TypeScript allows you to have multiple generic types in the type parameter list. For example:

```

class className<K, T>{
    //...
}

```

---

```
}
```

The generic constraints are also applied to the generic types in the class:

```
class className<T extends TypeA>{  
    //...  
}
```

Placing the type parameter on the class allows you to develop methods and properties that work with the same type.

## TypeScript generic classes example

In this example, we will develop a generic Stack class.

A stack is a data structure that works on the last-in-first-out (or LIFO) principle. It means that the first element you place into the stack is the last element you can get from the stack.

Typically, a stack has a size. By default, it is empty. The stack has two main operations:

- Push: push an element into the stack.
- Pop: pop an element from the stack.

The following shows a complete generic Stack class called Stack<T>:

```
class Stack<T> {  
    private elements: T[] = [];  
  
    constructor(private size: number) {  
    }  
    isEmpty(): boolean {  
        return this.elements.length === 0;  
    }  
    isFull(): boolean {  
        return this.elements.length === this.size;  
    }  
    push(element: T): void {  
        if (this.elements.length === this.size) {  
            throw new Error('The stack is overflow!');  
        }  
        this.elements.push(element);  
    }  
}
```

```

    pop(): T {
        if (this.elements.length == 0) {
            throw new Error('The stack is empty!');
        }
        return this.elements.pop();
    }
}

```

The following creates a new stack of numbers:

```

let numbers = new Stack<number>(5);

```

This function returns a random number between two numbers, low and high:

```

function randBetween(low: number, high: number): number {
    return Math.floor(Math.random() * (high - low + 1) + low);
}

```

Now, you can use the randBetween() function to generate random numbers for pushing into the numbers stack:

```

let numbers = new Stack<number>(5);

while (!numbers.isFull()) {
    let n = randBetween(1, 10);
    console.log(`Push ${n} into the stack.`);
    numbers.push(n);
}

```

Output:

```

Push 3 into the stack.
Push 2 into the stack.
Push 1 into the stack.
Push 8 into the stack.
Push 9 into the stack.

```

The following shows how to pop elements from the stack until it is empty:

```

while (!numbers.isEmpty()) {
    let n = numbers.pop();
    console.log(`Pop ${n} from the stack.`);
}

```

Output:

---

```
Pop 9 from the stack.  
Pop 8 from the stack.  
Pop 1 from the stack.  
Pop 2 from the stack.  
Pop 3 from the stack.
```

Similarly, you can create a stack of strings. For example:

```
let words = 'The quick brown fox jumps over the lazy dog'.split(' ');  
  
let wordStack = new Stack<string>(words.length);  
  
// push words into the stack  
words.forEach(word => wordStack.push(word));  
  
// pop words from the stack  
while (!wordStack.isEmpty()) {  
    console.log(wordStack.pop());  
}
```

How it works:

- First, split the sentences into words.
- Second, create a stack whose size is equal to the number of words in the words array.
- Third, push elements of the words array into the stack.
- Finally, pop words from the stack until it is empty.

In this tutorial, you have learned how to develop generic classes in TypeScript.

## Modules

### TypeScript Modules

In this tutorial, you will learn about the TypeScript modules and how to use them to structure your code.

#### Introduction to TypeScript modules

Since ES6, JavaScript started supporting modules as the native part of the language. TypeScript shares the same module concept with JavaScript.

---

A TypeScript module can contain both declarations and code. A module executes within its own scope, not in the global scope. It means that when you declare variables, functions, classes, interfaces, etc., in a module, they are not visible outside the module unless you explicitly export them using export statement.

On the other hand, if you want to access variables, functions, classes, etc., from a module, you need to import them using the import statement.

Like ES6, when TypeScript file contains a top-level import or export, it is treated as a module.

## Creating a new module

The following creates a new module called Validator.ts and declares an interface named Validator:

```
export interface Validator {  
    isValid(s: string): boolean  
}
```

In this module, we place the export keyword before the interface keyword to expose it to other modules.

In other words, if you do not use the export keyword, the Validator interface is private in the Validator.ts module, therefore, it cannot be used by other modules.

## Export statements

Another way to export a declaration from a module is to use the export statement. For example:

```
interface Validator {  
    isValid(s: string): boolean  
}  
  
export { Validator };
```

TypeScript also allows you to rename declarations for module consumers, like this:

```
interface Validator {  
    isValid(s: string): boolean  
}  
  
export { Validator as StringValidator };
```

In this example other modules will use the Validator interface as the StringValidator interface.



---

## Importing a new module

To consume a module, you use the import statement. The following creates a new module `EmailValidator.ts` that uses the `Validator.ts` module:

```
import { Validator } from './Validator';

class EmailValidator implements Validator {
    isValid(s: string): boolean {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(s);
    }
}

export { EmailValidator };
```

When you import a module, you can rename it like this:

```
import { Validator as StringValidator } from './Validator';
```

Inside the `EmailValidator` module, you use the `Validator` interface as the `StringValidator` interface instead:

```
import { Validator as StringValidator } from './Validator';

class EmailValidator implements StringValidator {
    isValid(s: string): boolean {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(s);
    }
}

export { EmailValidator };
```

The following illustrates how to use the `EmailValidator` module in the `App.ts` file:

```
import { EmailValidator } from './EmailValidator';

let email = 'john.doe@typescripttutorial.net';
let validator = new EmailValidator();
let result = validator.isValid(email);

console.log(result);
```

Output:

---

```
true
```

## Importing types

The following declares a type called in Types.ts module:

```
export type alphanumeric = string | number;
```

To import the alphanumeric type from the Types.ts module, you can use the import type statement:

```
import type {alphanumeric} from './Types';
```

Note that TypeScript has supported the import type statement since version 3.8. Prior to TypeScript 3.8, you need to use the import statement instead:

```
import {alphanumeric} from './Types';
```

## Importing everything from a module

To import everything from a module, you use the following syntax:

```
import * from 'module_name';
```

## Re-exports

The following creates a new module called ZipCodeValidator.ts that uses the Validator.ts module:

```
import { Validator } from './Validator';

class ZipCodeValidator implements Validator {
  isValid(s: string): boolean {
    const numberRegex = /^[0-9]+$/;
    return s.length === 5 && numberRegex.test(s);
  }
}

export { ZipCodeValidator };
```

You can wrap the EmailValidator and ZipCodeValidator modules in a new module by combining all their exports using the following syntax:

```
export * from 'module_name';
```

The following example illustrates how to wrap the the EmailValidator.ts and ZipCodeValidator.ts modules in the FormValidator.ts module:

```
export * from './EmailValidator';
export * from './ZipCodeValidator';
```

---

## Default Exports

TypeScript allows each module to have one default export. To mark an export as a default export, you use the default keyword.

The following shows how to export the ZipCodeValidator as a default export:

```
import { Validator } from './Validator';

export default class ZipCodeValidator implements Validator {
  isValid(s: string): boolean {
    const numberRegex = /^[0-9]+$/;
    return s.length === 5 && numberRegex.test(s);
  }
}
```

To import a default export, you use a different import syntax like this:

```
import default_export from 'module_name';
```

The following shows how to use the default export from the ZipCodeValidator in the App.ts file:

```
import ZipCodeValidator from './ZipCodeValidator';

let validator = new ZipCodeValidator();
let result = validator.isValid('95134');

console.log(result);
```

Output:

```
true
```

## Summary

- TypeScript shares the same module concept with ES6 module. A module can contain both declarations and code.
- In a module, variables, functions, classes, interfaces, etc., executes on its own scope, not the global scope.
- Use export statement to export variables, functions, classes, interfaces, type, etc., from a module.

- 
- Use import statement to access exports from other modules.