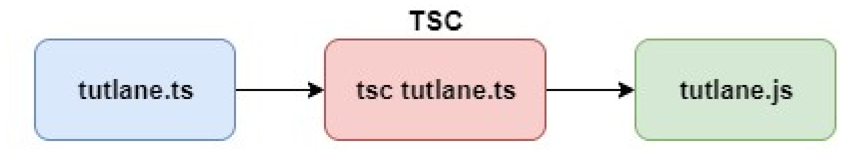
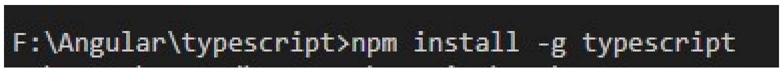
TypeScript is an open-source object-oriented programming language and it's a typed superset of JavaScript that compiles to plain JavaScript that means the code whatever we wrote in typescript will be compiled into plain JavaScript code by Typescript compiler.

Generally, we will write the TypeScript code in a file with `.ts` extension and the typescript compiler (TSC) will compile and convert `.ts` extension file to plain JavaScript file with `tsc filename.ts` command like as shown below.

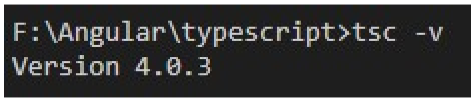


TypeScript Installation

Install TypeScript using Node.js NPM

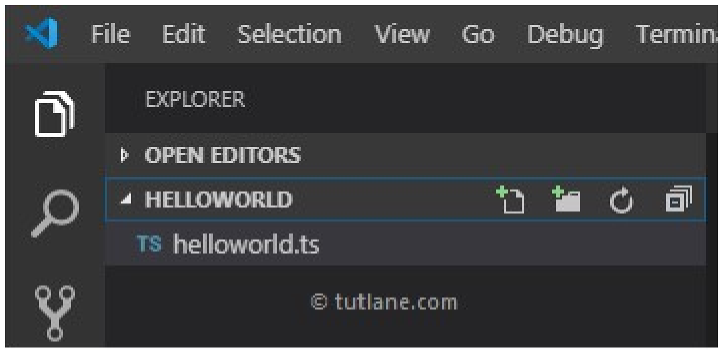


```
F:\Angular\typescript>npm install -g typescript
```

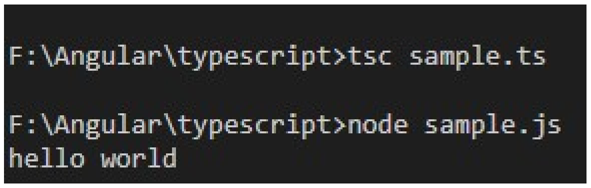


```
F:\Angular\typescript>tsc -v
Version 4.0.3
```

TypeScript Hello World Program



Compile and run the TypeScript:



```
F:\Angular\typescript>tsc sample.ts

F:\Angular\typescript>node sample.js
hello world
```

TypeScript Data Types

TypeScript Data Types Syntax

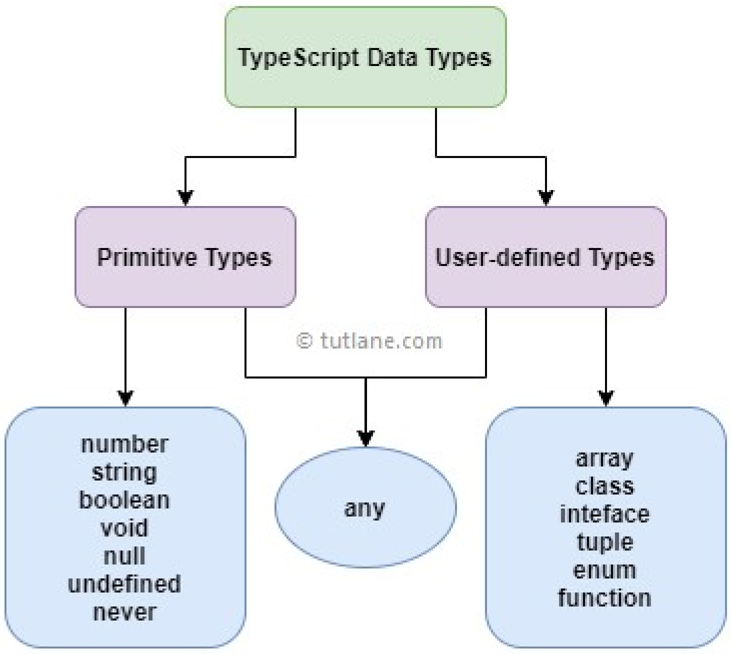
```
[Keyword] [Variable Name]: [Data Type] = [Value];
[Keyword] [Variable Name]: [Data Type];
```

- **[Keyword]** - It's a keyword of the variable either `var`, `let` or `const` to define the scope and usage of variable.
- **[Variable Name]** - Its name of the variable to hold the values in our application.
- **[Data Type]** - It's a type of data the variable can hold such as number, string, boolean, etc.
- **[Value]** - Assigning a required value to the variable.

TypeScript Data Types Example

```
let uname: string = "Suresh Dasari"; // string
let ulocation: string = "Hyderabad"; // string
let age: number = 33; // number
let isCompleted: boolean = true; // boolean
```

TypeScript Different Data Types



TypeScript Number Data Type

TypeScript Number Data Type Example

```
let num1: number = 446; // number
let num2: number = 78.65; // number
let num3: number = 0x120A; // hexadecimal
let num4: number = 0b1110; // binary
let num5: number = 0o211; // octal
```

TypeScript Number Object

```
[Keyword] [Variable Name] = new Number(Value);
```

```
let num1 = new Number(446);
let num2 = new Number(46.6521);
```

TypeScript String Data Type

TypeScript String Data Type Example

```
let uname: string = "Suresh Dasari";
let ulocation: string = 'Hyderabad';
let ueducation: string = "B.Tech"
```

TypeScript Template Strings

In ES6, the template strings have been introduced to support multiline and embedded expressions in strings. From 1.4 version, the TypeScript also supporting template strings to embed expressions into the string.

```
let uname: string = "Suresh Dasari";
let ulocation: string = 'Hyderabad';
// Normal string
let nstr: string = uname + " staying in " + ulocation;
// Template string
let tstr: string = `${uname} staying in ${ulocation}`
console.log(nstr); // Suresh Dasari staying in Hyderabad
console.log(tstr); // Suresh Dasari staying in Hyderabad
```

```
// Multi-line Normal String
let msg: string = "Hello, " + uname + "\n" + "You are from " + ulocation + ". Enjoy Lear
ning";
console.log(msg);
// Multi-line Template String
let msg1: string = `Hello, ${uname}
You are from ${ulocation}. Enjoy Learning`
console.log(msg1);
```

```
Hello, Suresh Dasari
You are from Hyderabad. Enjoy Learning
Hello, Suresh Dasari
You are from Hyderabad. Enjoy learning
```

Example: Type Annotation of Parameters

```
function display(id:number, name:string)
{
    console.log("Id = " + id + ", Name = " + name);
}
```

Example: Type Annotation in Object

```
var employee : {
    id: number;
    name: string;
};

employee = {
  id: 100,
  name : "John"
}
```

TypeScript - Arrays

An array is a special type of data type which can store multiple values of different data types sequentially using a special syntax.

There are two ways to declare an array:

1. Using square brackets. This method is similar to how you would declare arrays in JavaScript.

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];
```

2. Using a generic array type, Array<elementType>.

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

Example: Array Declaration and Initialization

```
let fruits: Array<string>;
fruits = ['Apple', 'Orange', 'Banana'];

let ids: Array<number>;
ids = [23, 34, 100, 124, 44];
```

Example: Multi Type Array

```
let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
// or
let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana']
```

Example: Access Array Elements using Loop

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];

for(var index in fruits)
{
    console.log(fruits[index]);  // output: Apple Orange Banana
}

for(var i = 0; i < fruits.length; i++)
{
    console.log(fruits[i]); // output: Apple Orange Banana
}
```

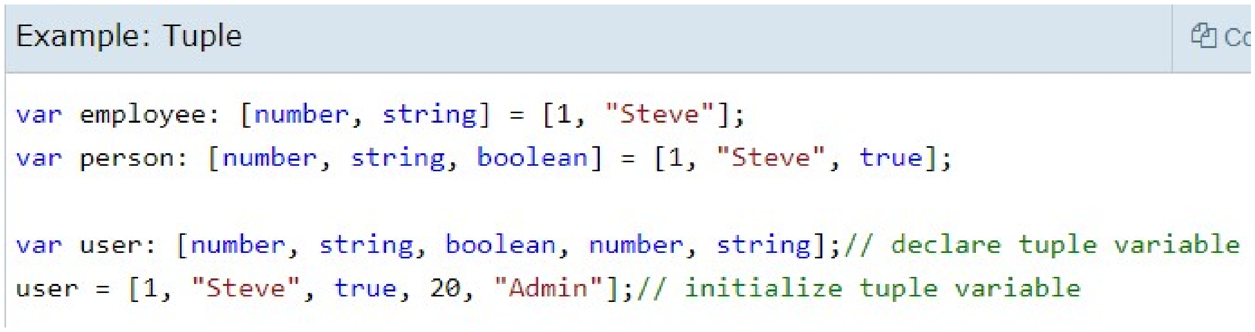
TypeScript - Tuples

TypeScript introduced a new data type called Tuple. Tuple can contain two values of different data types.

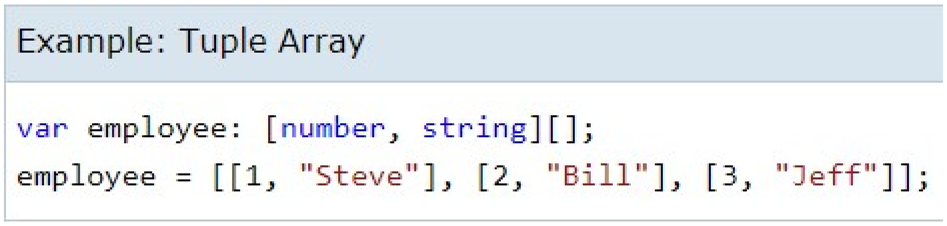
Example: Tuple vs Other Data Types

```
var empId: number = 1;
var empName: string = "Steve";

// Tuple type variable
var employee: [number, string] = [1, "Steve"];
```

A tuple type variable can include multiple data types as shown below.


Example: Tuple

```
var employee: [number, string] = [1, "Steve"];
var person: [number, string, boolean] = [1, "Steve", true];

var user: [number, string, boolean, number, string];// declare tuple variable
user = [1, "Steve", true, 20, "Admin"];// initialize tuple variable
```


Example: Tuple Array

```
var employee: [number, string][];
employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]];
```

TypeScript Data Type – Enum

enums allow us to declare a set of named constants i.e. a collection of related values that can be numeric or string values.

There are three types of enums:

1. Numeric enum
2. String enum
3. Heterogeneous enum

Numeric Enum

Numeric enums are number-based enums i.e. they store string values as numbers.

Example: Numeric Enum

```
enum PrintMedia {
  Newspaper,
  Newsletter,
  Magazine,
  Book
}
```

Here, enum values start from zero and increment by 1 for each member. It would be represented as:

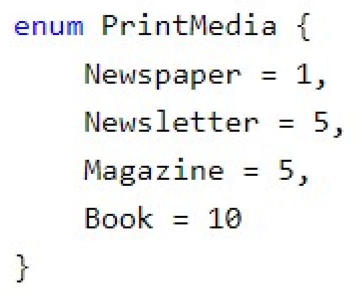
```
Newspaper = 0
Newsletter = 1
Magazine = 2
Book = 3
```

We also have the option to initialize the first numeric value ourselves. For example, we can write the same enum as:

```
enum PrintMedia {
  Newspaper = 1,
  Newsletter,
  Magazine,
  Book
}
```

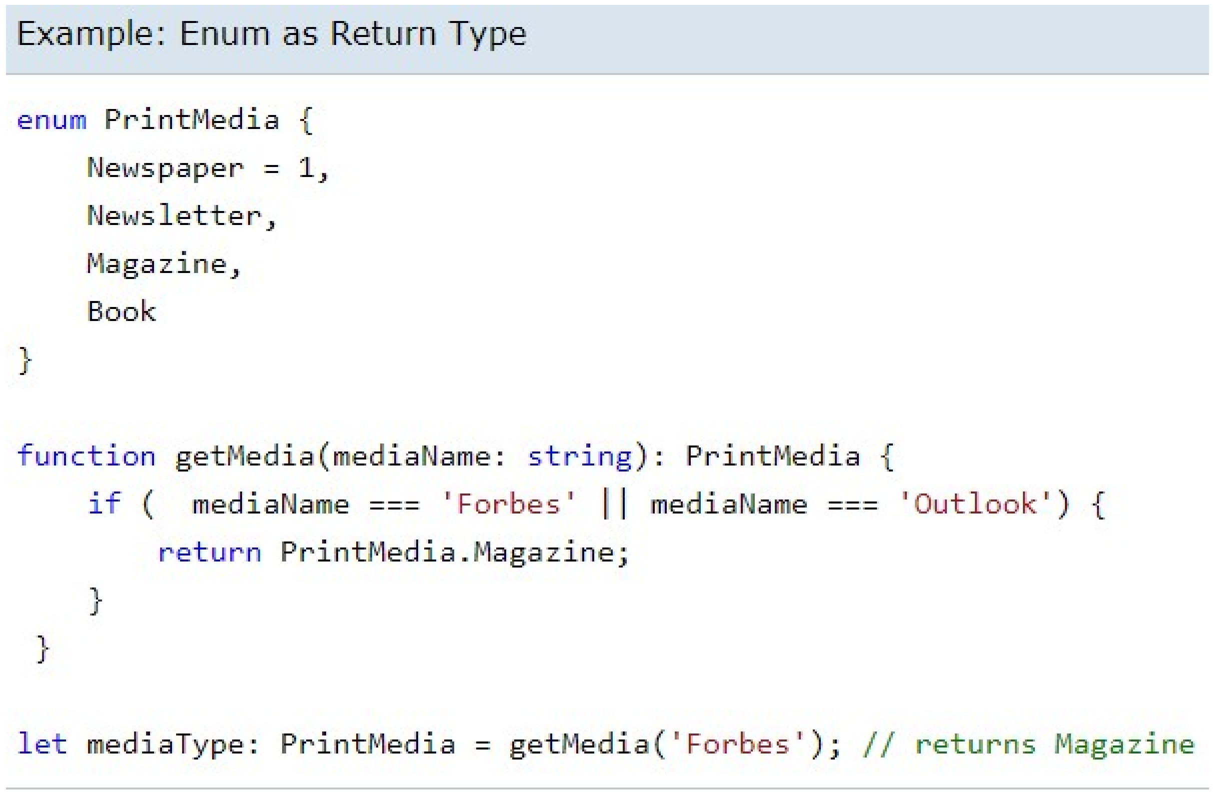
The first member, Newspaper, is initialized with the numeric value 1. The remaining members will be incremented by 1 from the numeric value of the first value. Thus, in the above example, Newsletter would be 2, Magazine would be 3 and Book would be 4.

It is not necessary to assign sequential values to Enum members. They can have any values.



```
enum PrintMedia {
    Newspaper = 1,
    Newsletter = 5,
    Magazine = 5,
    Book = 10
}
```

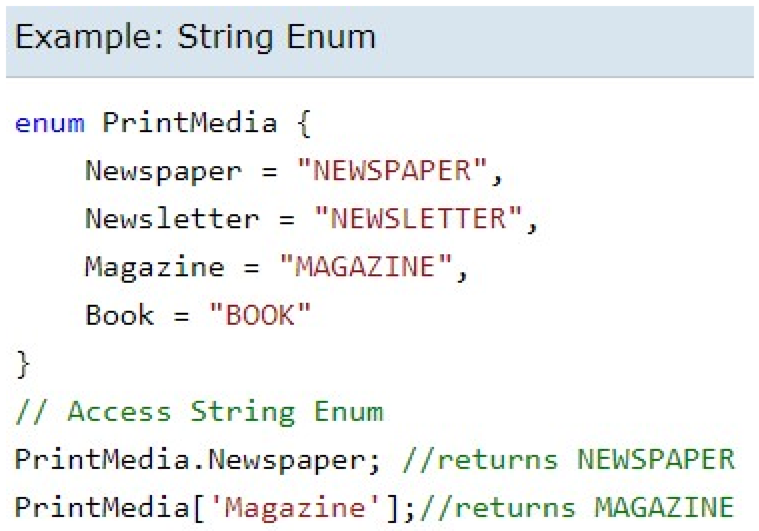
The enum can be used as a function parameter or return type, as shown below:



Example: Enum as Return Type

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter,
    Magazine,
    Book
}

function getMedia(mediaName: string): PrintMedia {
    if (  mediaName === 'Forbes' || mediaName === 'Outlook') {
        return PrintMedia.Magazine;
    }
 }

let mediaType: PrintMedia = getMedia('Forbes'); // returns Magazine
```

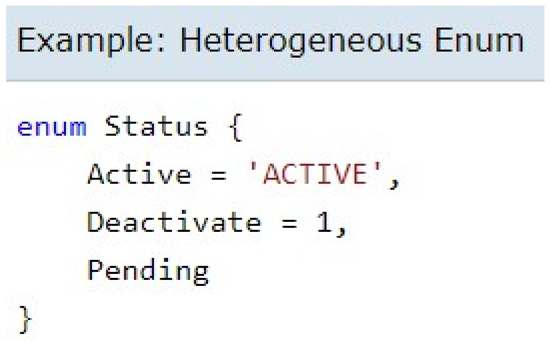
String Enum



Example: String Enum

```
enum PrintMedia {
    Newspaper = "NEWSPAPER",
    Newsletter = "NEWSLETTER",
    Magazine = "MAGAZINE",
    Book = "BOOK"
}
// Access String Enum
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

Heterogeneous Enum

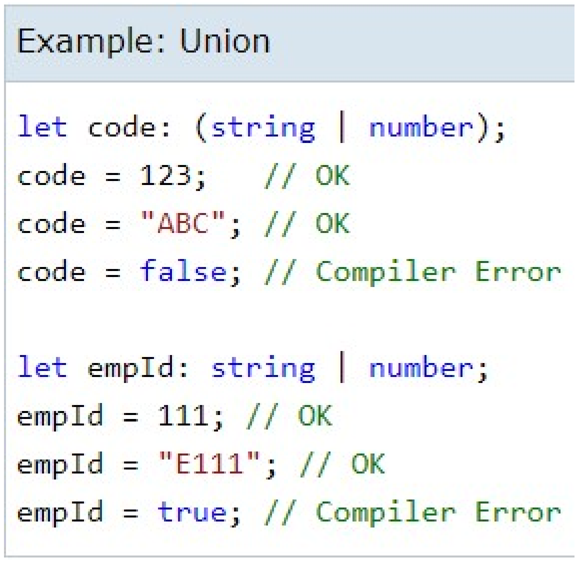


Example: Heterogeneous Enum

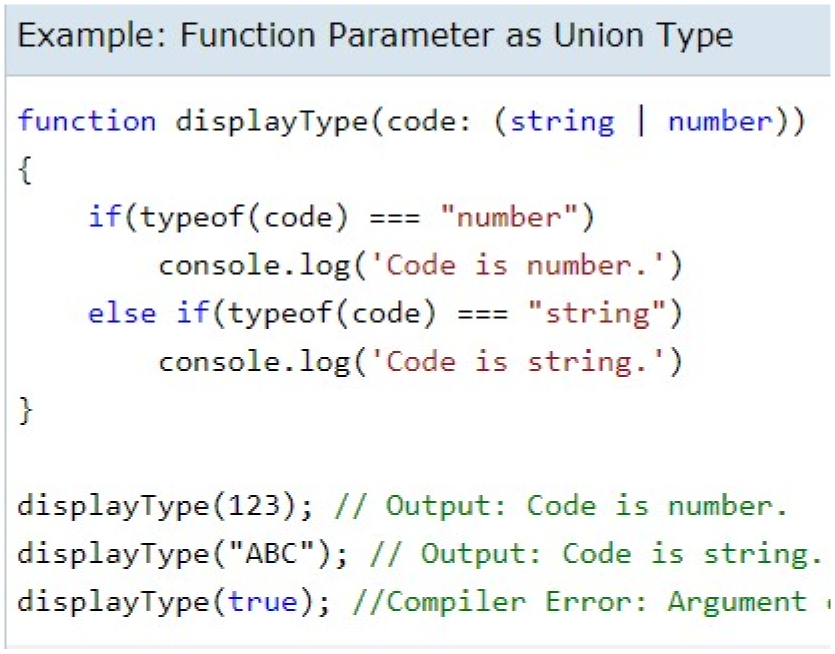
```
enum Status {
    Active = 'ACTIVE',
    Deactivate = 1,
    Pending
}
```

TypeScript - Union

TypeScript allows us to use more than one data type for a variable or a function parameter. This is called union type.

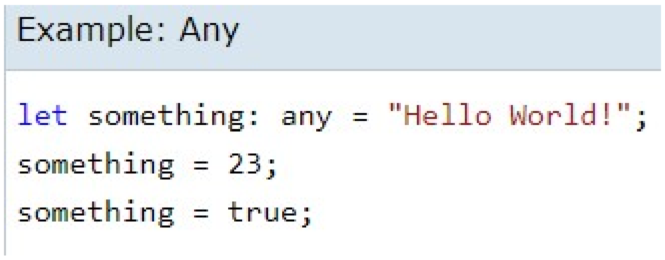

Example: Union

```
let code: (string | number);
code = 123;   // OK
code = "ABC"; // OK
code = false; // Compiler Error

let empId: string | number;
empId = 111; // OK
empId = "E111"; // OK
empId = true; // Compiler Error
```


Example: Function Parameter as Union Type

```
function displayType(code: (string | number))
{
    if(typeof(code) === "number")
        console.log('Code is number.')
    else if(typeof(code) === "string")
        console.log('Code is string.')
}

displayType(123); // Output: Code is number.
displayType("ABC"); // Output: Code is string.
displayType(true); //Compiler Error: Argument
```

TypeScript Data Type - Any


Example: Any

```
let something: any = "Hello World!";
something = 23;
something = true;
```

Example: Any type Array

```
let arr: any[] = ["John", 212, true];
arr.push("Smith");
console.log(arr); //Output: [ 'John', 212, true, 'Smith' ]
```

TypeScript Data Type - Void

if a function does not return any value then you can specify void as return type.

Example: void

```
function sayHi(): void {
    console.log('Hi!')
}

let speech: void = sayHi();
console.log(speech); //Output: undefined
```

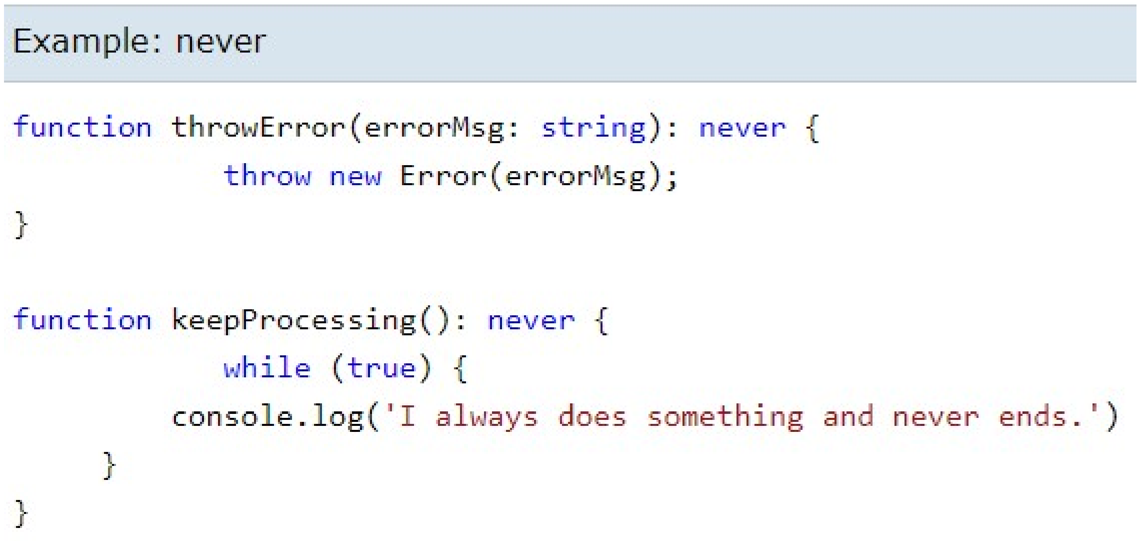
There is no meaning to assign void to a variable, as only null or undefined is assignable to void.

```
let nothing: void = undefined;
let num: void = 1; // Error
```

TypeScript Data Type - Never

TypeScript introduced a new type `never`, which indicates the values that will never occur.

The never type is used when you are sure that something is never going to occur. For example, you write a function which will not return to its end point or always throws an exception.

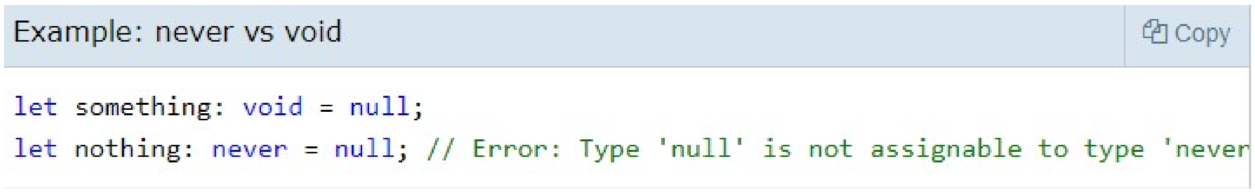


Example: never

```
function throwError(errorMsg: string): never {
            throw new Error(errorMsg);
}

function keepProcessing(): never {
            while (true) {
         console.log('I always does something and never ends.')
     }
}
```

Difference between never and void

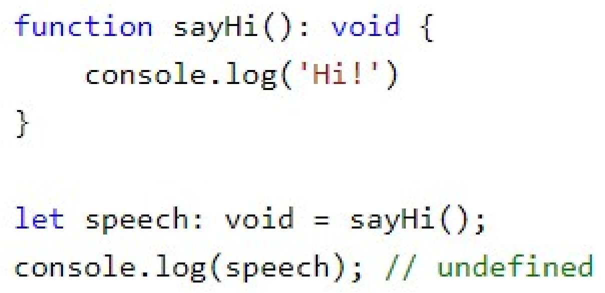
The void type can have undefined or null as a value where as never cannot have any value.



Example: never vs void

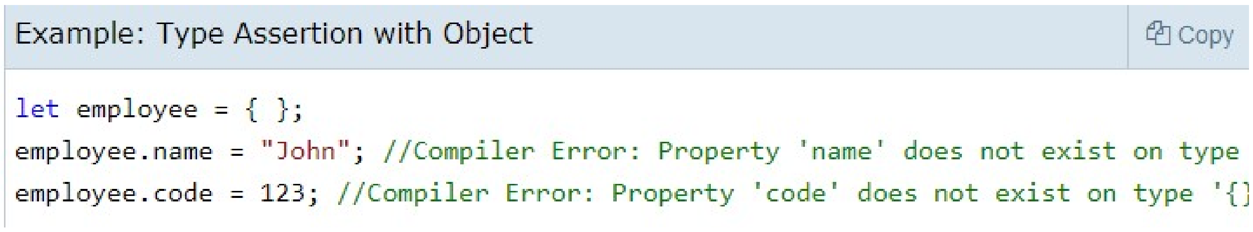
```
let something: void = null;
let nothing: never = null; // Error: Type 'null' is not assignable to type 'never
```

In TypeScript, a function that does not return a value, actually returns undefined. Consider the following example.



```
function sayHi(): void {
    console.log('Hi!')
}

let speech: void = sayHi();
console.log(speech); // undefined
```

As you can see in the above example, `speech` is undefined, because the `sayHi` function internally returns undefined even if return type is void. If you use never type, `speech:never` will give a compile time error, as void is not assignable to never.

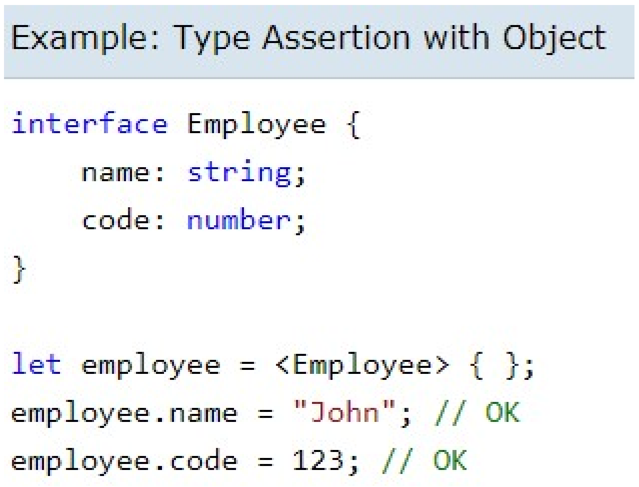


Example: Type Assertion with Object

Copy

```
let employee = { };
employee.name = "John"; //Compiler Error: Property 'name' does not exist on type
employee.code = 123; //Compiler Error: Property 'code' does not exist on type '{}
```

The above example will give a compiler error, because the compiler assumes that the type of employee is {} with no properties. But, we can avoid this situation by using type assertion, as shown below.



Example: Type Assertion with Object

```
interface Employee {
    name: string;
    code: number;
}

let employee = <Employee> { };
employee.name = "John"; // OK
employee.code = 123; // OK
```

In the above example, we created an interface `Employee` with the properties name and code. We then used this type assertion on employee.

Be careful while using type assertion. The TypeScript compiler will auto completes `Employee` properties, but it won't show any compile time error if you forgot to add the properties. For example:

```
interface Employee {
    name: string;
    code: number;
}

let employee = <Employee> {
    // Compiler will provide autocomplete properties,
    but will not give an error if you forgot to add the properties
};
console.log(employee.name); // undefined;
```

There are two ways to do type assertion in TypeScript:

1. Using the angular bracket <> syntax. So far in this section, we have used angular brackets to show type assertion.

```
let code: any = 123;
let employeeCode = <number> code;
```

2. using as keyword

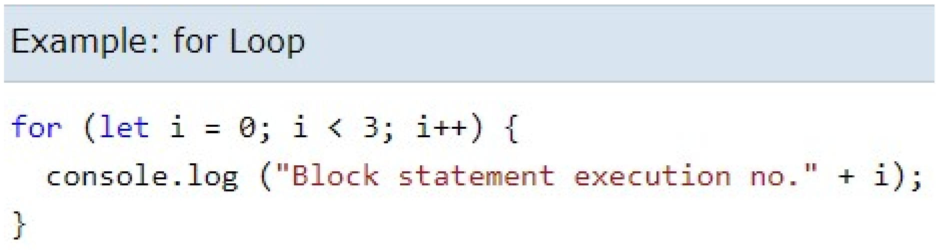
Example: as syntax

```
let code: any = 123;
let employeeCode = code as number;
```

TypeScript - for Loops

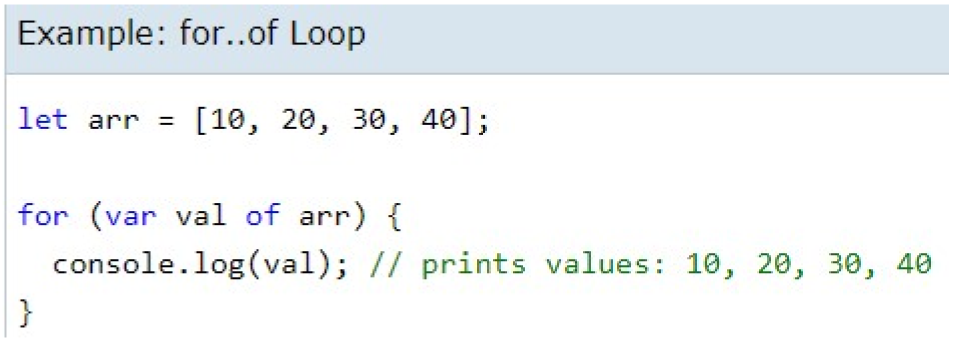
TypeScript supports the following for loops:

1. for loop
2. for..of loop
3. for..in loop



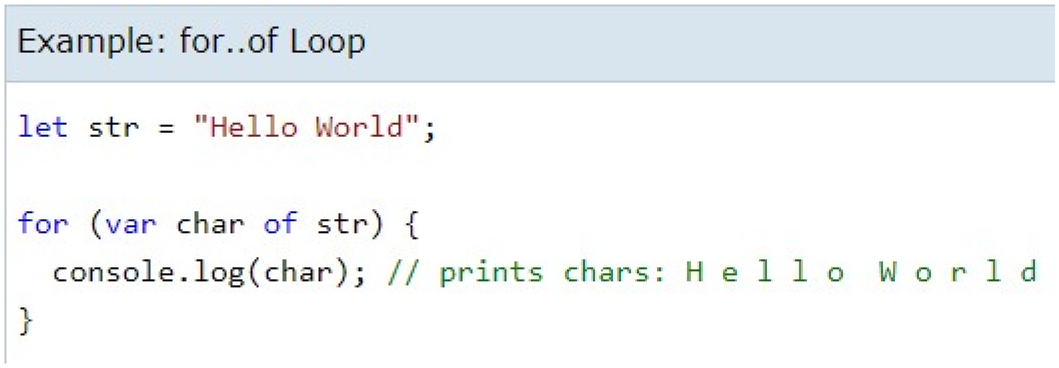
Example: for Loop

```
for (let i = 0; i < 3; i++) {
  console.log ("Block statement execution no." + i);
}
```



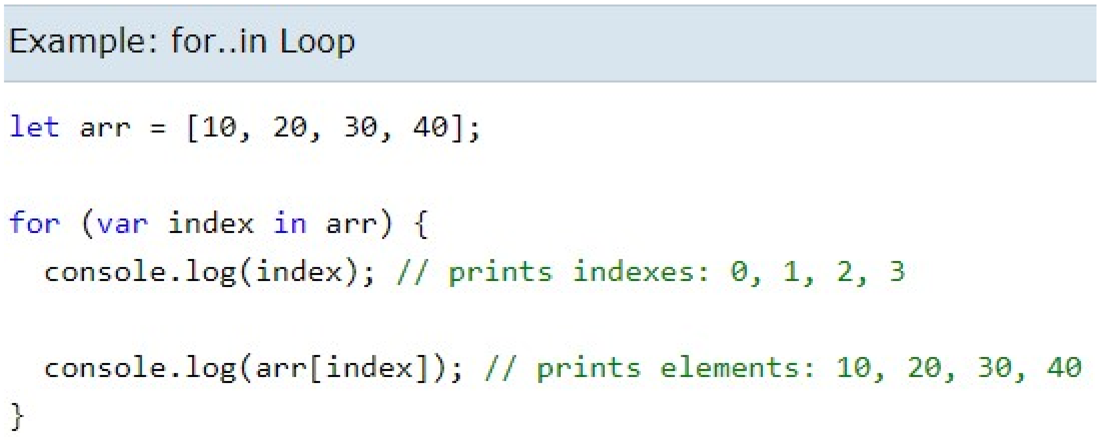
Example: for..of Loop

```
let arr = [10, 20, 30, 40];

for (var val of arr) {
  console.log(val); // prints values: 10, 20, 30, 40
}
```



Example: for..of Loop

```
let str = "Hello World";

for (var char of str) {
  console.log(char); // prints chars: H e l l o  W o r l d
}
```



Example: for..in Loop

```
let arr = [10, 20, 30, 40];

for (var index in arr) {
  console.log(index); // prints indexes: 0, 1, 2, 3

  console.log(arr[index]); // prints elements: 10, 20, 30, 40
}
```

TypeScript - Functions

Named Functions

Example: Named Function

```
function display() {
    console.log("Hello TypeScript!");
}

display(); //Output: Hello TypeScript
```

Example: Function with Parameter and Return Types

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

Sum(2,3); // returns 5
```

Anonymous Function

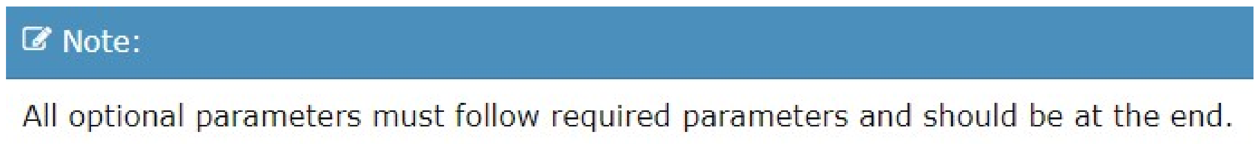
Example: Anonymous Function

```
let greeting = function() {
    console.log("Hello TypeScript!");
};

greeting(); //Output: Hello TypeScript!
```

Example: Function with Paramter and Return Types

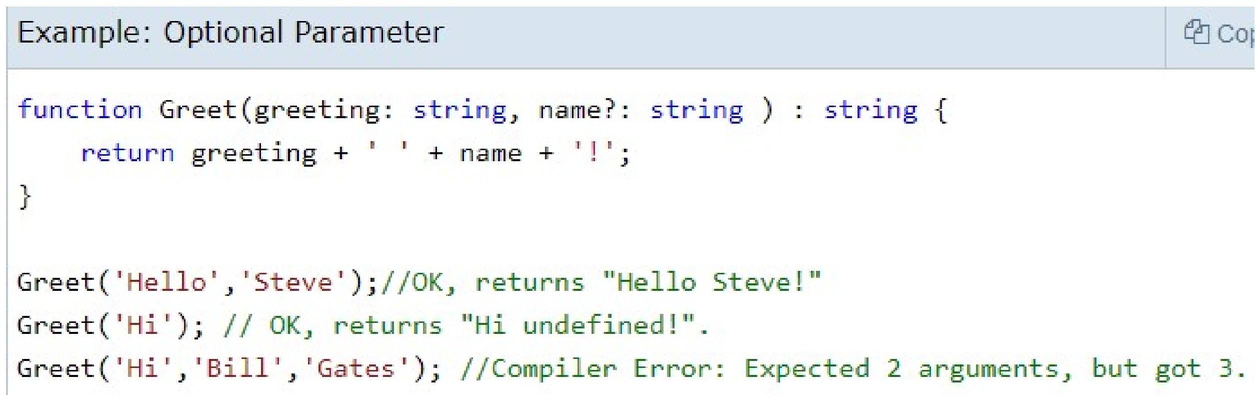
```
let Sum = function(x: number, y: number) : number
{
    return x + y;
}

Sum(2,3); // returns 5
```

Optional Parameters



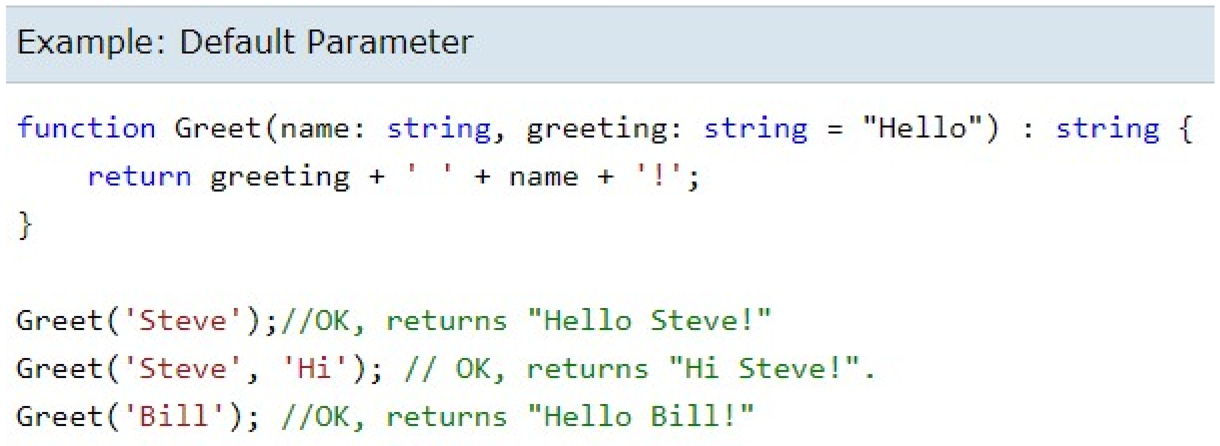
Note:

All optional parameters must follow required parameters and should be at the end.



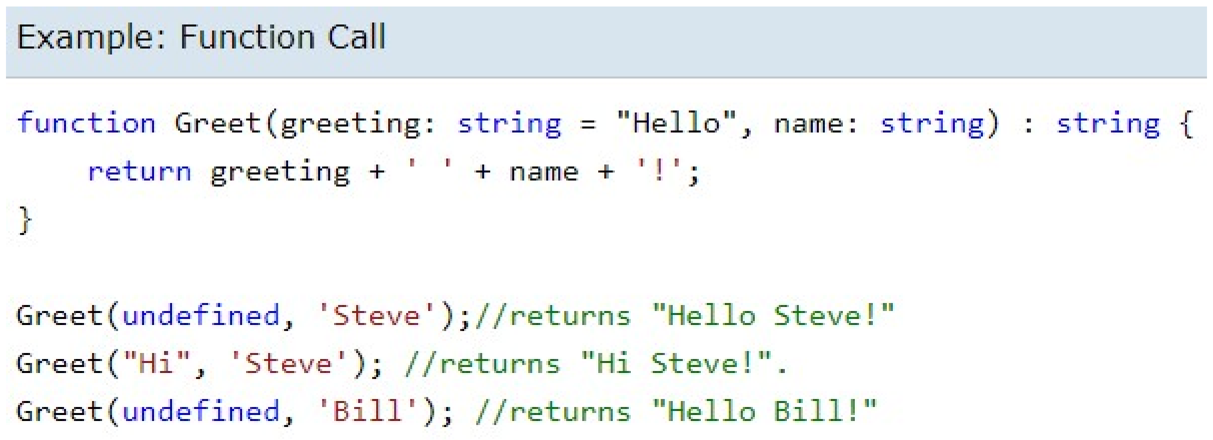
Example: Optional Parameter

```
function Greet(greeting: string, name?: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // OK, returns "Hi undefined!".
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```



Example: Default Parameter

```
function Greet(name: string, greeting: string = "Hello") : string {
    return greeting + ' ' + name + '!';
}

Greet('Steve');//OK, returns "Hello Steve!"
Greet('Steve', 'Hi'); // OK, returns "Hi Steve!".
Greet('Bill'); //OK, returns "Hello Bill!"
```

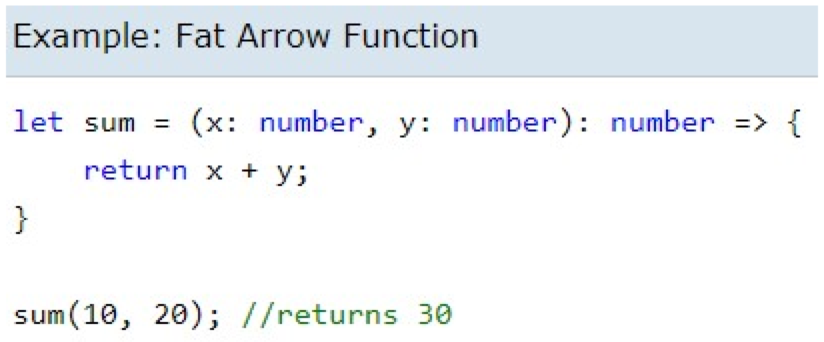


Example: Function Call

```
function Greet(greeting: string = "Hello", name: string) : string {
    return greeting + ' ' + name + '!';
}

Greet(undefined, 'Steve');//returns "Hello Steve!"
Greet("Hi", 'Steve'); //returns "Hi Steve!".
Greet(undefined, 'Bill'); //returns "Hello Bill!"
```

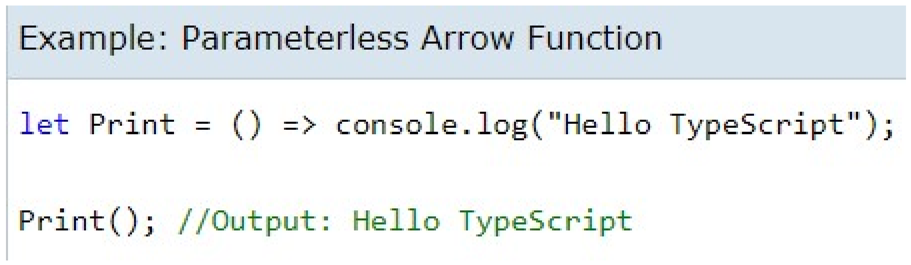
TypeScript - Arrow Functions

Fat arrow notations are used for anonymous functions i.e for function expressions. They are also called lambda functions in other languages.



Example: Fat Arrow Function

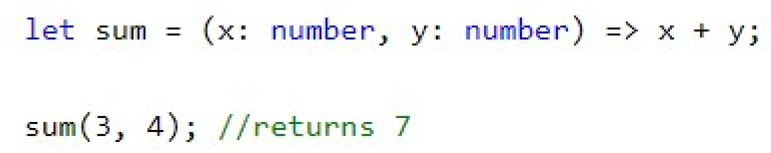
```
let sum = (x: number, y: number): number => {
    return x + y;
}

sum(10, 20); //returns 30
```



Example: Parameterless Arrow Function

```
let Print = () => console.log("Hello TypeScript");

Print(); //Output: Hello TypeScript
```

if the function body consists of only one statement then no need for the curly brackets and the return keyword, as shown below.



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

sum(3, 4); //returns 7
```

Example: Arrow Function in Class

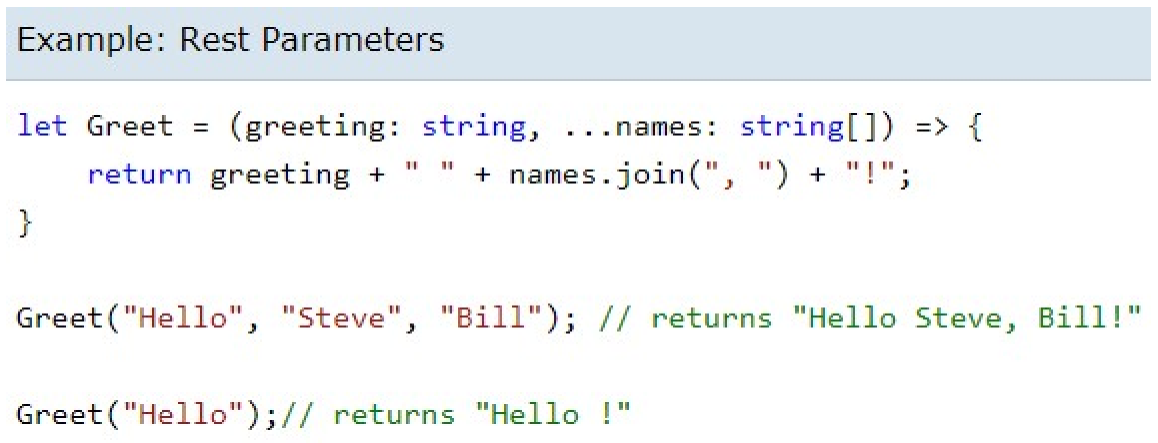
```
class Employee {
    empCode: number;
    empName: string;

    constructor(code: number, name: string) {
        this.empName = name;
        this.empCode = code;
    }

    display = () => console.log(this.empCode +' ' + this.empName)
}
let emp = new Employee(1, 'Ram');
emp.display();
```

TypeScript - Rest Parameters

When the number of parameters that a function will receive is not known or can vary, we can use rest parameters.

Example: Rest Parameters

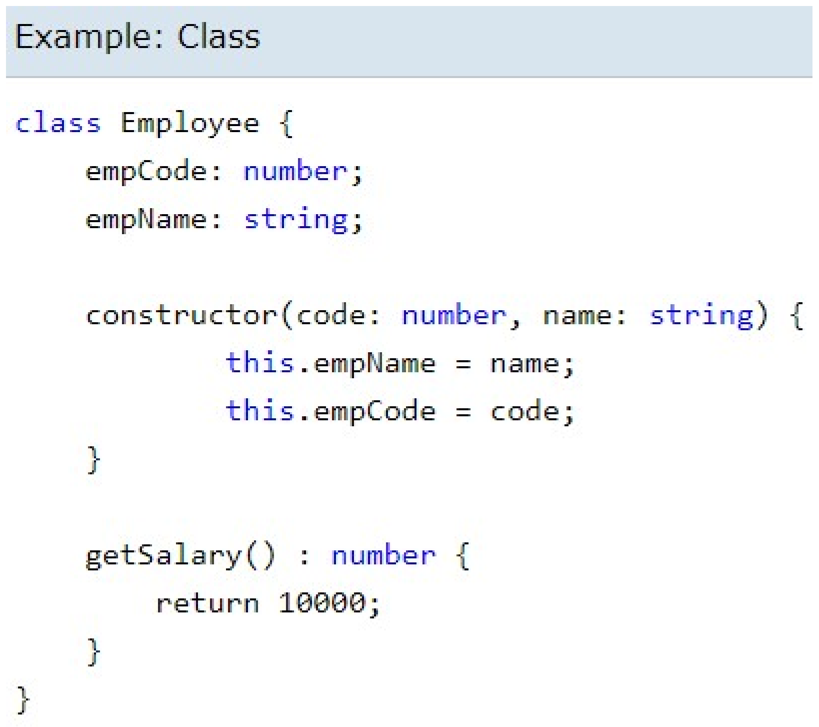
```
function Greet(greeting: string, ...names: string[]) {
    return greeting + " " + names.join(", ") + "!";
}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"
```



Example: Rest Parameters

```
let Greet = (greeting: string, ...names: string[]) => {
    return greeting + " " + names.join(", ") + "!";
}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"
```

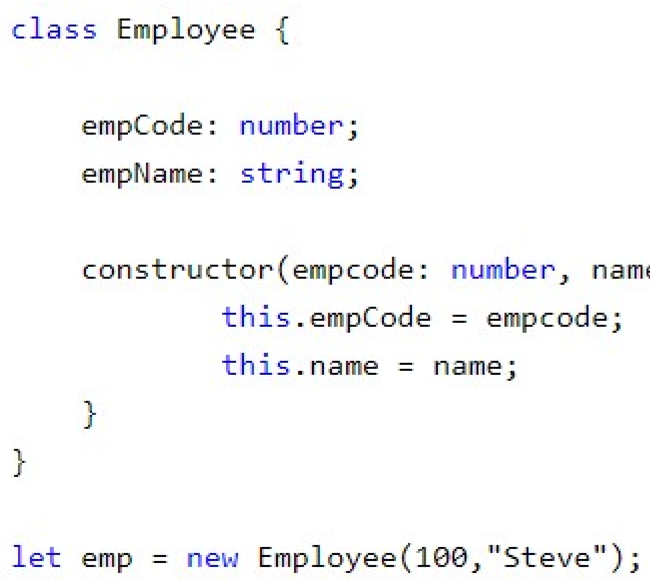
TypeScript - Classes



Example: Class

```
class Employee {
    empCode: number;
    empName: string;

    constructor(code: number, name: string) {
            this.empName = name;
            this.empCode = code;
    }

    getSalary() : number {
        return 10000;
    }
}
```

Object creation



```
class Employee {

    empCode: number;
    empName: string;

    constructor(empcode: number, name: string ) {
            this.empCode = empcode;
            this.name = name;
    }
}

let emp = new Employee(100,"Steve");
```

TypeScript - Abstract Class

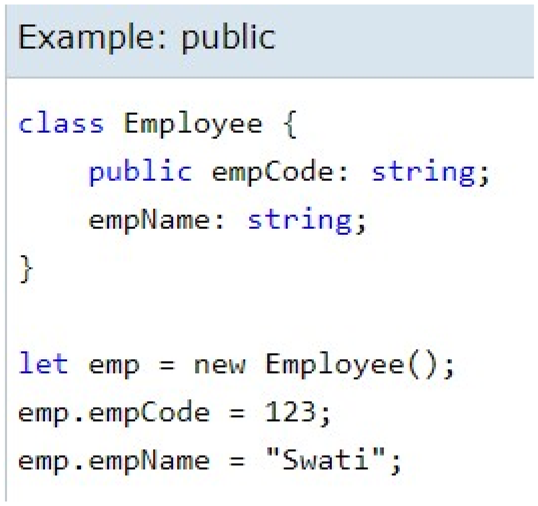
Example: Abstract Class

```
abstract class Person {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    display(): void{
        console.log(this.name);
    }

    abstract find(string): Person;
}
class Employee extends Person {
    empCode: number;

    constructor(name: string, code: number) {
        super(name); // must call super()
        this.empCode = code;
    }

    find(name:string): Person {
        // execute AJAX request to find an employee from a db
        return new Employee(name, 1);
    }
}

let emp: Person = new Employee("James", 100);
emp.display(); //James

let emp2: Person = emp.find('Steve');
```

TypeScript - Data Modifiers

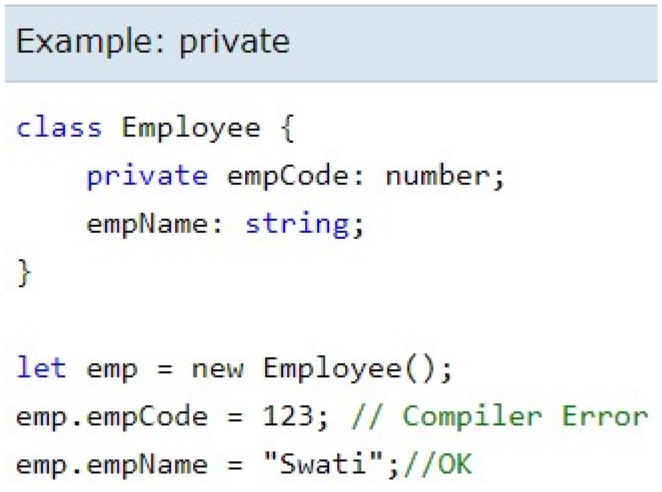
There are three types of access modifiers in TypeScript: public, private and protected.

TypeScript treats properties and methods as public by default if no modifier is applied to them.



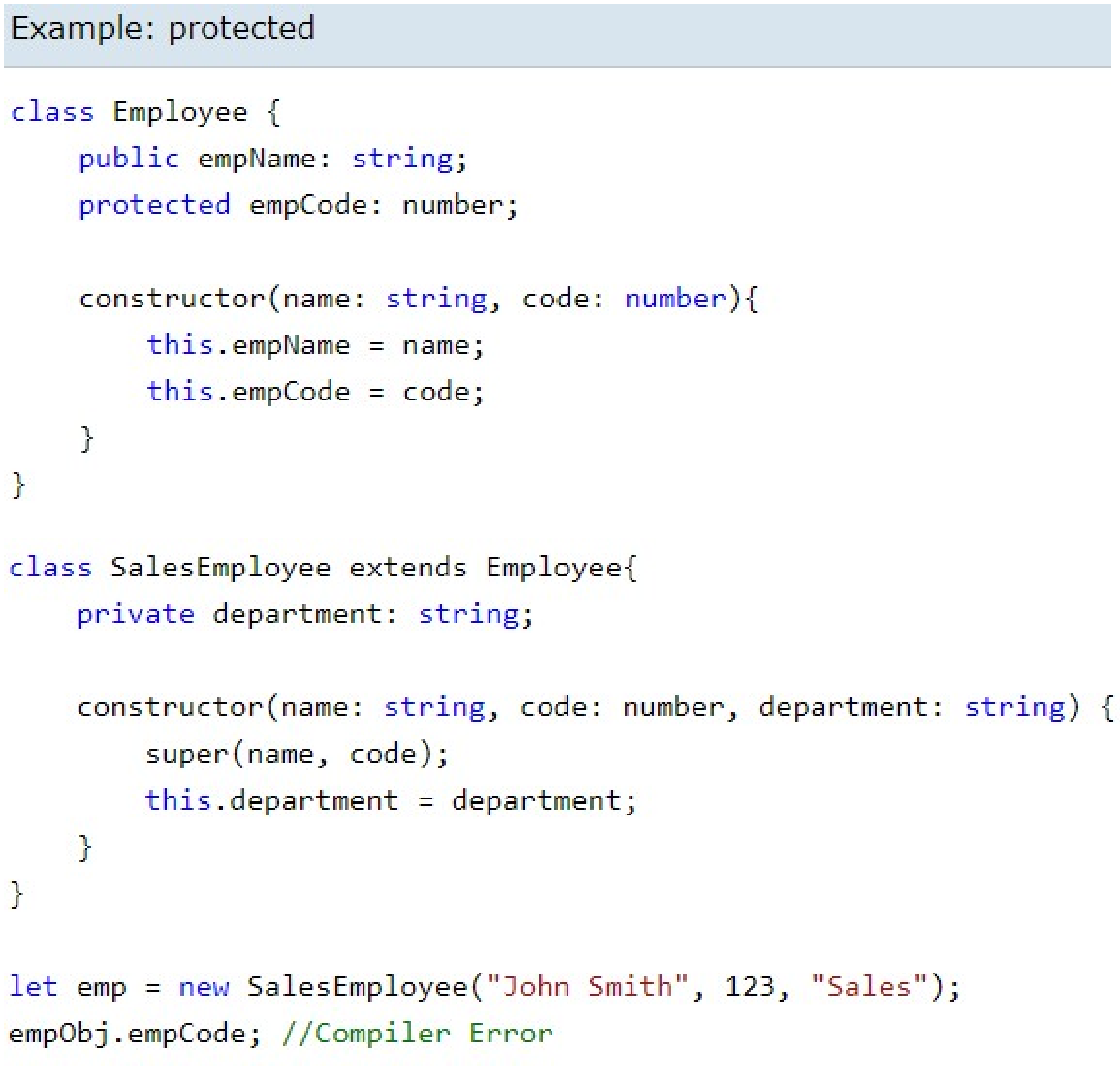
Example: public

```
class Employee {
    public empCode: string;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123;
emp.empName = "Swati";
```



Example: private

```
class Employee {
    private empCode: number;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123; // Compiler Error
emp.empName = "Swati";//OK
```

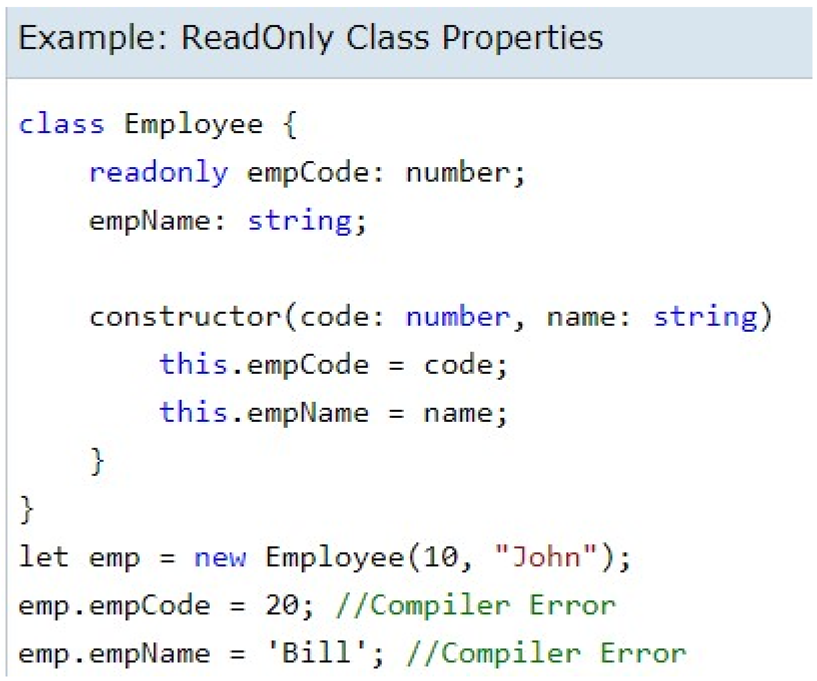


Example: protected

```
class Employee {
    public empName: string;
    protected empCode: number;

    constructor(name: string, code: number){
        this.empName = name;
        this.empCode = code;
    }
}

class SalesEmployee extends Employee{
    private department: string;

    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.department = department;
    }
}

let emp = new SalesEmployee("John Smith", 123, "Sales");
empObj.empCode; //Compiler Error
```

Property 'empCode' is protected and only accessible within class 'Employee' and its subclasses.

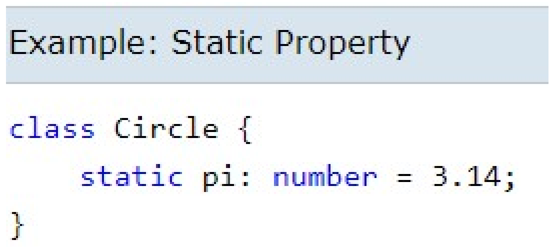
TypeScript - ReadOnly



Example: ReadOnly Class Properties

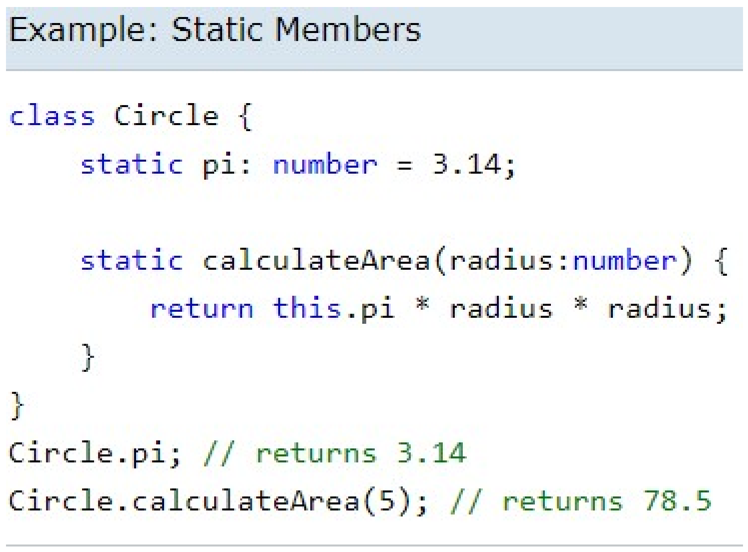
```
class Employee {
    readonly empCode: number;
    empName: string;

    constructor(code: number, name: string)
        this.empCode = code;
        this.empName = name;
    }
}
let emp = new Employee(10, "John");
emp.empCode = 20; //Compiler Error
emp.empName = 'Bill'; //Compiler Error
```

TypeScript - Static



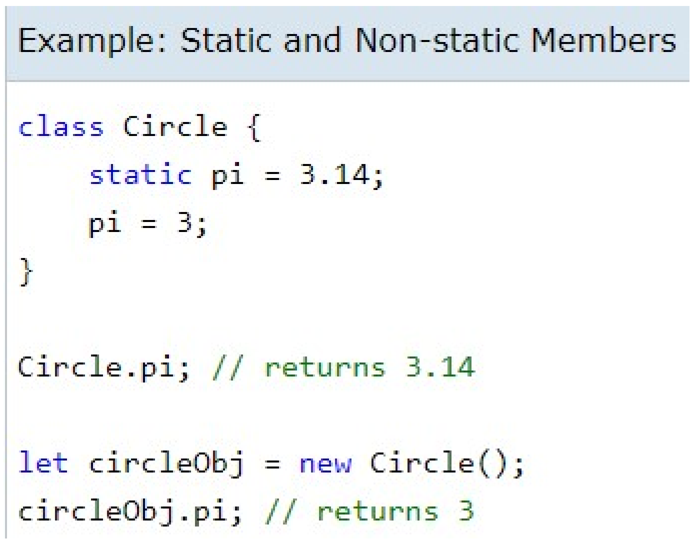
Example: Static Property

```
class Circle {
    static pi: number = 3.14;
}
```



Example: Static Members

```
class Circle {
    static pi: number = 3.14;

    static calculateArea(radius:number) {
        return this.pi * radius * radius;
    }
}
Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5
```



Example: Static and Non-static Members

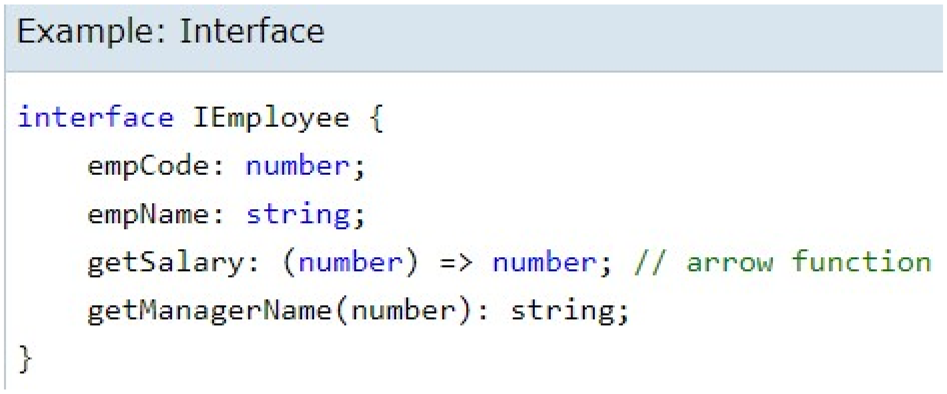
```
class Circle {
    static pi = 3.14;
    pi = 3;
}

Circle.pi; // returns 3.14

let circleObj = new Circle();
circleObj.pi; // returns 3
```

As you can see, static and non-static fields with the same name can exists without any error. The static field will be accessed using dot notation and the non-static field can be accessed using an object.

TypeScript - Interfaces

Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

The TypeScript compiler does not convert interface to JavaScript. It uses interface for type checking. This is also known as "duck typing" or "structural subtyping".

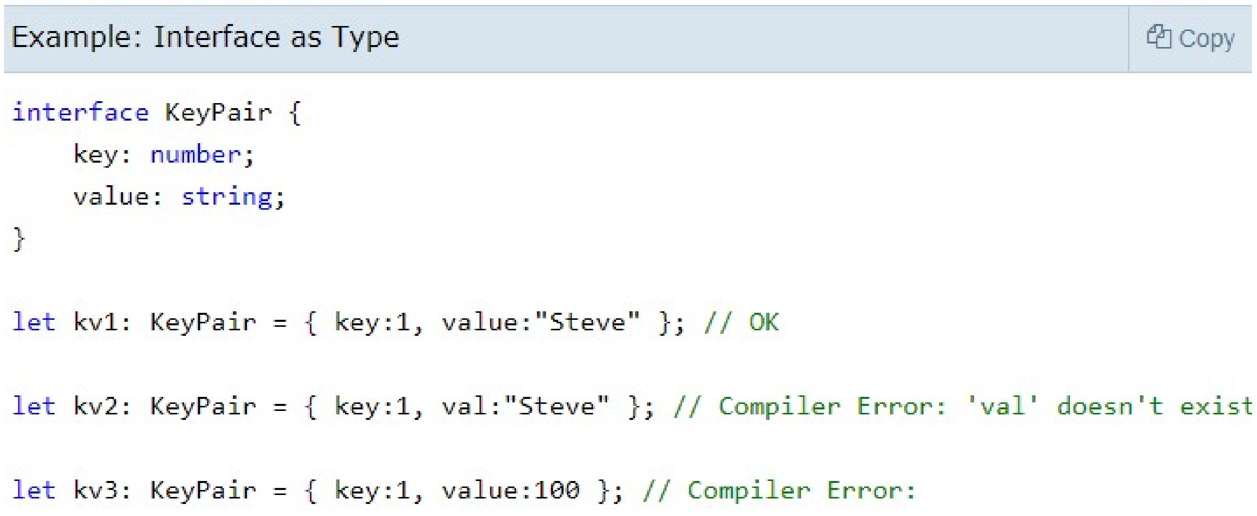


Example: Interface

```
interface IEmployee {
    empCode: number;
    empName: string;
    getSalary: (number) => number; // arrow function
    getManagerName(number): string;
}
```

Interface as Type

Interface in TypeScript can be used to define a type and also to implement it in the class.



Example: Interface as Type

```
interface KeyPair {
    key: number;
    value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK

let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val' doesn't exist

let kv3: KeyPair = { key:1, value:100 }; // Compiler Error:
```