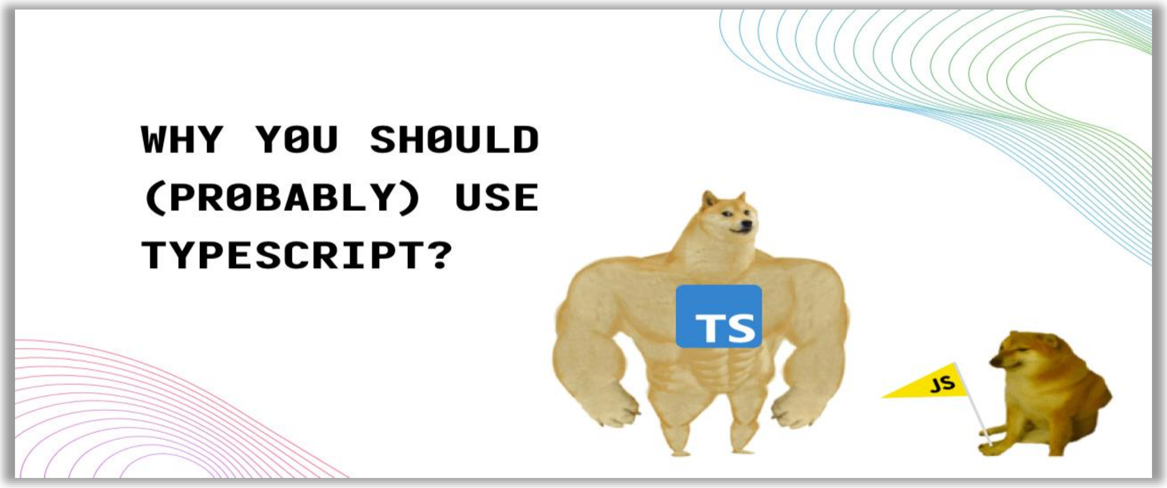
<TypeScript> Programming Language

TypeScript is a super-set of JavaScript, meaning that it does everything that JavaScript does, but with some added features.

The main reason for using TyoeScript is to cannot be changed at any point in a program.

On the other hand, JavaScript is a dynamically typed programming language, meaning variable can change type.

Why you should use TypeScript Programmimg?



The top reasons for why should use TypeScript Programmimg language in your next project are as follows:

- Research has shown that TypeScript can spot 15% of common bugs.
- **Readability**:- It is easier to see what the code it supposed to do. And when working in a team, it is easier to see what the other developers intended to.
- **It’s popular**:- Knowing TypeScript will enable you to apply to more good jobs.
- Learning TypeScript will give you a better understanding, and a new perspective, on JavaScript.
- TypeScript Programming language can prevent most of the irritating bugs.

Datatypes

❖Built-in datatypes:

Built-in datatypes are also called Primitive datatypes.
Primitives are immutable means that they can’t be altered.
It is important not to confuse a primitive itself with a variable assigned a primitive value.
A primitive value is data that is not an object and has no methods.
The variable may be reassigned by a new value, but the existing value can’t be changed.

TypesScript Built-in Types

1) Number:

In TypeScript, Number Built-in Data Type is represented by keyword called ‘number’.

It is used to represent both integer as well as Floating-Poing numbers.

2) Boolean:

In TypeScript, Boolean Built-in data type is represented by the keyword called ‘boolean’.

It represents the values of true and false only.

3) String:

In TypeScript, string Built-in data type is represented by keyword called ‘string’.
It is represented a sequence of characters.

4) Void:

In TypeScript, Void Built-in data type is represented by the keyword called ‘void’.
It is generally used on function return types.

5) Null:

In TypeScript, Boolean Built-in data type is represented by keyword called ‘null’.
It is used when an object does not have any value.

6) Undefined:

In TypeScript, Undefined Built-in data type is represented by the keyword called ‘undefined’.

It denoted value given to uninitialized variable.

7) Any:

In TypeScript, All built-in data types is represented by the keyword called ‘any’.
If variable is declared with any data type then any type of value can be assigned to that variable.

❖User-defined datatypes:

Apart from built-in datatypes, user can also defined its own data type.
User-defined types include Enumeration (enums), classes, interfaces, arrays, tuple, interfaces and functions.

Arrays

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

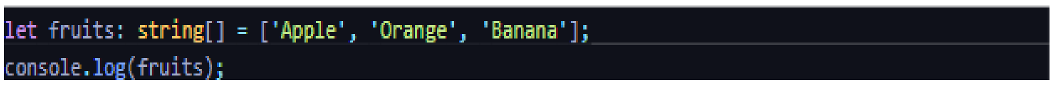
TypeScript support arrays, similar to JavaScript.

There are two ways to declare an array:

1. Using square brackets:

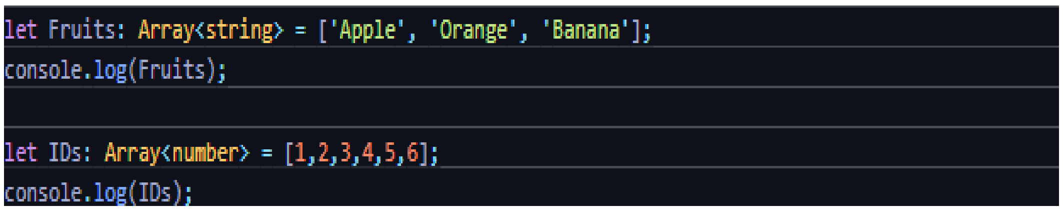
   This method is similar to how you declare array in JavaScript.

   e.g.



```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];
console.log(fruits);
```

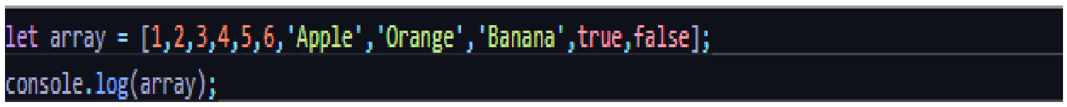
2. Using a generic array type, Array<elementType>:



```
let Fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
console.log(Fruits);

let IDs: Array<number> = [1,2,3,4,5,6];
console.log(IDs);
```

   Both methods produce the same output.

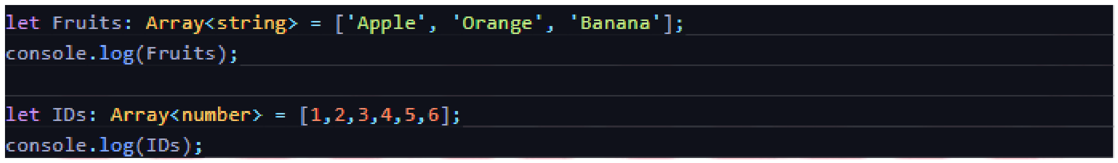
Of course, we can always initialize an array like shown blow, but you will not get the advantage of TypeScript type system.



```
let array = [1,2,3,4,5,6,'Apple','Orange','Banana',true,false];
console.log(array);
```

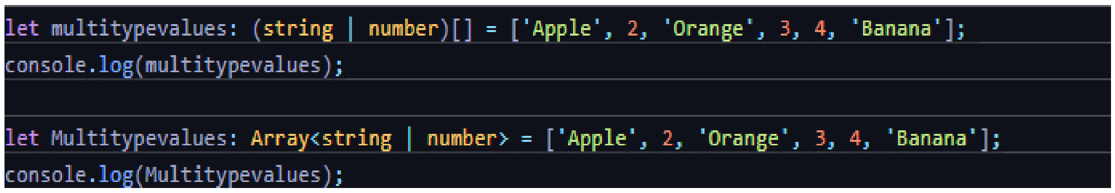
Array can contain elements of any data type, numbers, string, or even objects.

Array can declare and initialized separately like code:



```
let Fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
console.log(Fruits);

let IDs: Array<number> = [1,2,3,4,5,6];
console.log(IDs);
```

An array in TypeScript can contain elements of different data type using a generic array type:



```
let multitypevalues: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
console.log(multitypevalues);

let Multitypevalues: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
console.log(Multitypevalues);
```

TypeScript Array Methods:

The following table lists all Array methods which can be used for different purpose.

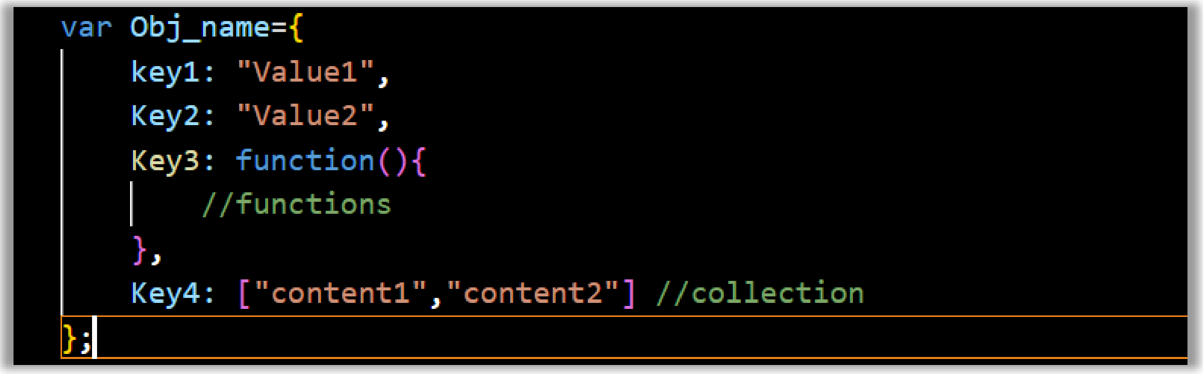
S.no	Method	Description
1	pop()	It removes the last element of the array and return that element.
2	push()	It adds new elements to the array and returns the new array length.
3	sort()	It sort all the elements of the array
4	concat()	It joins two array and return the combined result
5	indexOf()	It returns the index of the first match of a value in the array (-1 if not found)

6	copyWithin()	It copies a sequence of elements within the array.
7	fill()	It fills the array with a static value from the provided start index to the end index.
8	shift()	It removes and return the first element of the array.
9	splice()	It adds or removes elements from the array.
10	unshift()	It adds one or more elements to the beginning of the array.
11	includes()	It checks whether the array contains a certain element.
12	join()	It joins all elements to the array into string

13	lastindexOf()	It returns the last index of an element in the array
14	slice()	It extracts a section of the array and returns the new array.
15	toString()	It returns a string representation of the array.
16	toLocalStrint()	It returns a localized string representing the array.

Objects

An object is an instance which contains set of key value pairs.



```
var Obj_name={
    key1: "Value1",
    Key2: "Value2",
    Key3: function(){
        //functions
    },
    Key4: ["content1","content2"] //collection
};
```

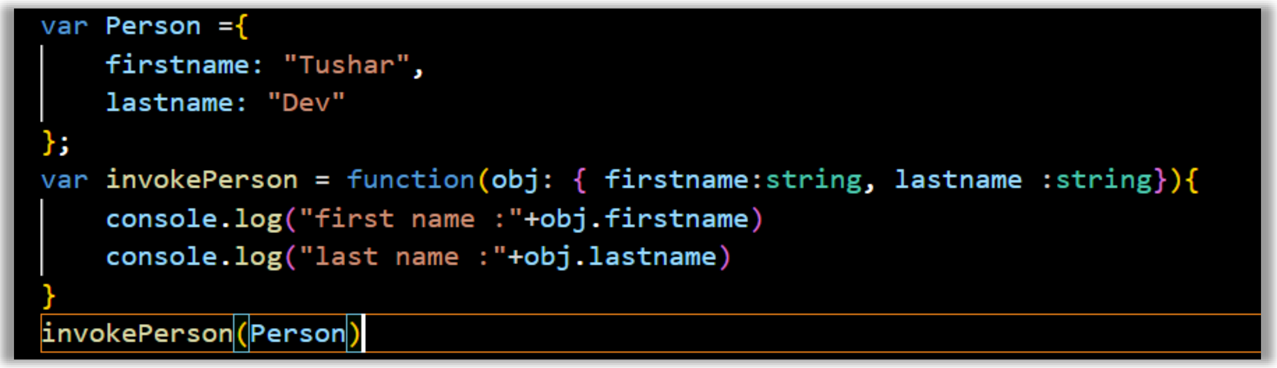
The values can be scalar value or functions or even array of other objects.

An object can contain scalar values (single values), functions and structures like arrays and tuples.

Objects as Function Parameters:

Object can also be passed as parameters to a function in TypeScript.

e.g.



```
var Person ={
    firstname: "Tushar",
    lastname: "Dev"
};
var invokePerson = function(obj: { firstname:string, lastname :string}){
    console.log("first name :"+obj.firstname)
    console.log("last name :"+obj.lastname)
}
invokePerson(Person)
```

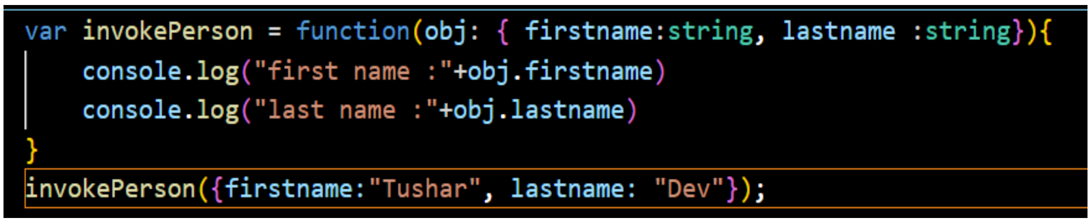
The example declare an object literal.

The function expression is invoked passing Person object.

Anonymous Object:

You can create and pass an anonymous object.

e.g.



```
var invokePerson = function(obj: { firstname:string, lastname :string}){
    console.log("first name :"+obj.firstname)
    console.log("last name :"+obj.lastname)
}
invokePerson({firstname:"Tushar", lastname: "Dev"});
```

Duck-Typing

In duck-typing, two objects are considered to be of the same type if both share the same set of properties.

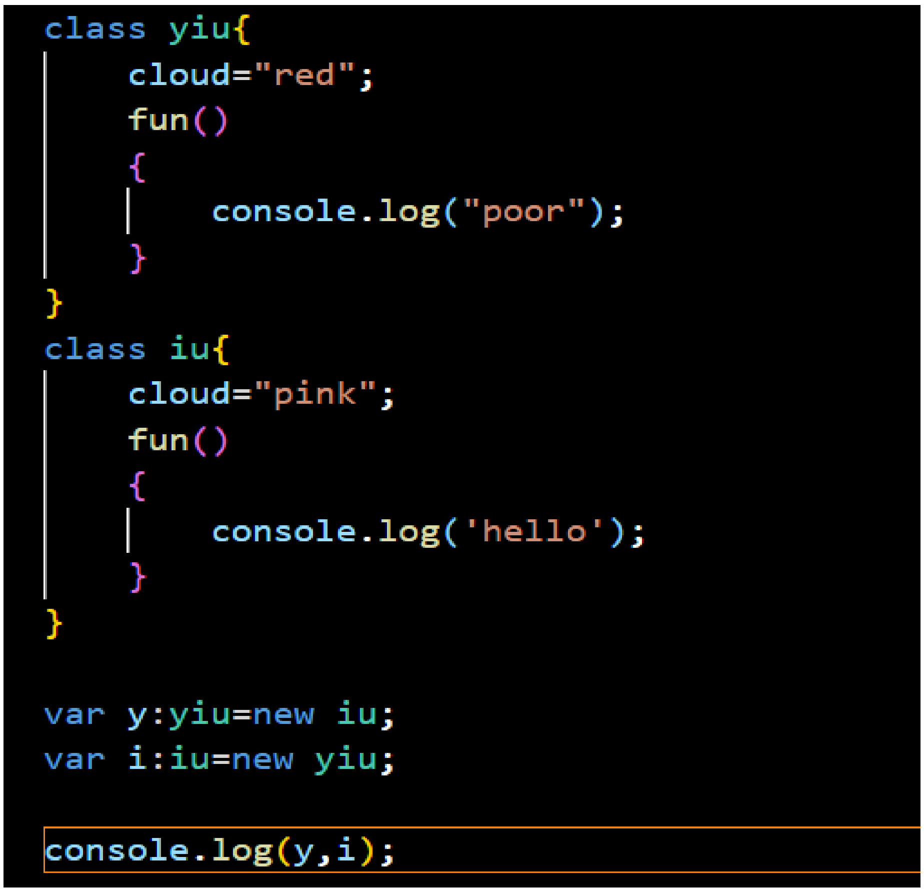
Duck-typing verifies the presence of certain properties in the objects, rather than their actual type, to check their suitability.

The concept is generally explained by the following phrase:

“When I see a bird that walks like a duck and swim like a duck and quacks like a duck, I call that bird a duck.”

The TypeScript compiler implements the duck-typing system that allows object creation while keeping type safety.

e.g.



```
class yiu{
    cloud="red";
    fun()
    {
        console.log("poor");
    }
}
class iu{
    cloud="pink";
    fun()
    {
        console.log('hello');
    }
}

var y:yiu=new iu;
var i:iu=new yiu;

console.log(y,i);
```

Functions

Functions are the primary blocks of any program.

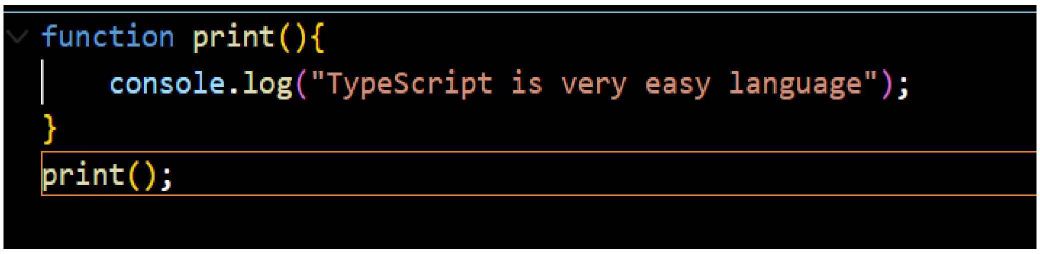
With functions, you can implement the concepts of object-oriented programming like classes, objects, Polymorphism and abstraction.

In TypeScript, functions can be two types:

1. Named Functions:

A named function is one where you declare and call a function by its given name.

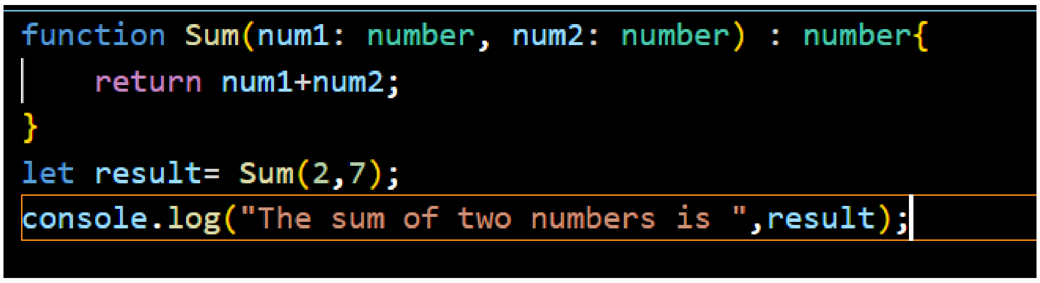
e.g.



```
function print(){
    console.log("TypeScript is very easy language");
}
print();
```

Functions can also include parameter type and return types.

e.g.



```
function Sum(num1: number, num2: number) : number{
    return num1+num2;
}
let result= Sum(2,7);
console.log("The sum of two numbers is ",result);
```

2. Anonymous Function:

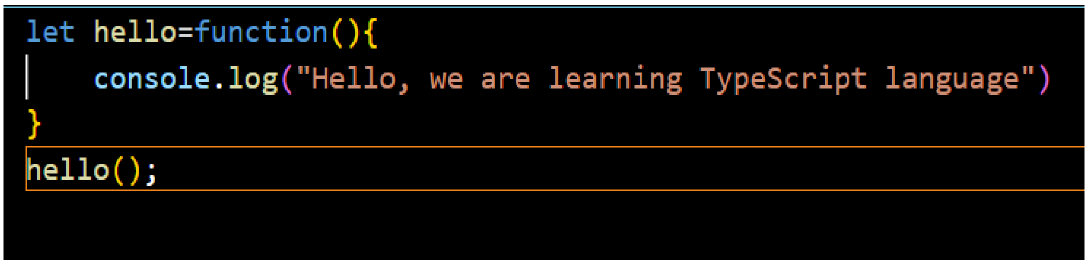
An anonymous function is one which is defined as an expression.

This expression is stored in a variable.

So, that function itself does not have a name.

These function are invoked using the variable name that the function is stored in.

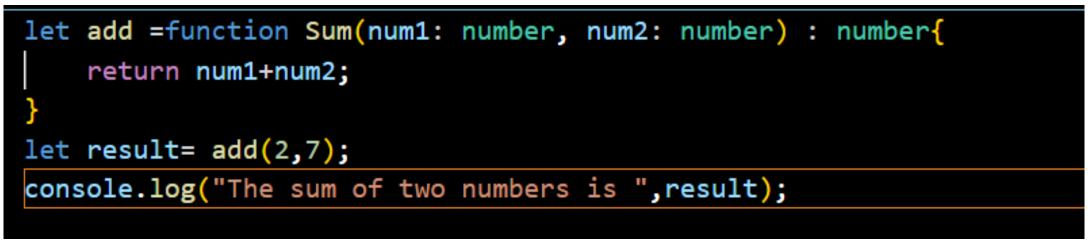
e.g.



```
let hello=function(){
    console.log("Hello, we are learning TypeScript language")
}
hello();
```

An anonymous function can also include parameter types and return type.

e.g.



```
let add =function Sum(num1: number, num2: number) : number{
    return num1+num2;
}
let result= add(2,7);
console.log("The sum of two numbers is ",result);
```

Arrow Function

Fat arrow notation are used for anonymous function i.e. for function expressions.

They are also called lambda functions in other languages.

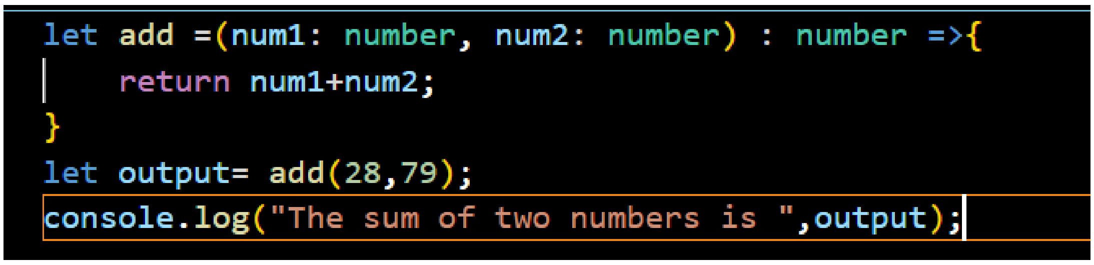
Syntax:

(param1, param2,…..paramN)=> expression

Using fat arrow =>, we dropped the need to use the function keyword.

Parameters are passed in the parenthesis (), and the function expression is enclosed is enclosed within the curly brackets {}.

e.g.

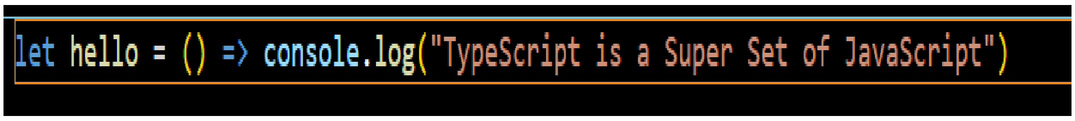


```
let add =(num1: number, num2: number) : number =>{
    return num1+num2;
}
let output= add(28,79);
console.log("The sum of two numbers is ",output);
```

Parameterless Arrow Function:

The arrow function without passing and parameters is called as Parameterless Arrow Function.

e.g.



```
let hello = () => console.log("TypeScript is a Super Set of JavaScript")
```

Dynamic Type (any)

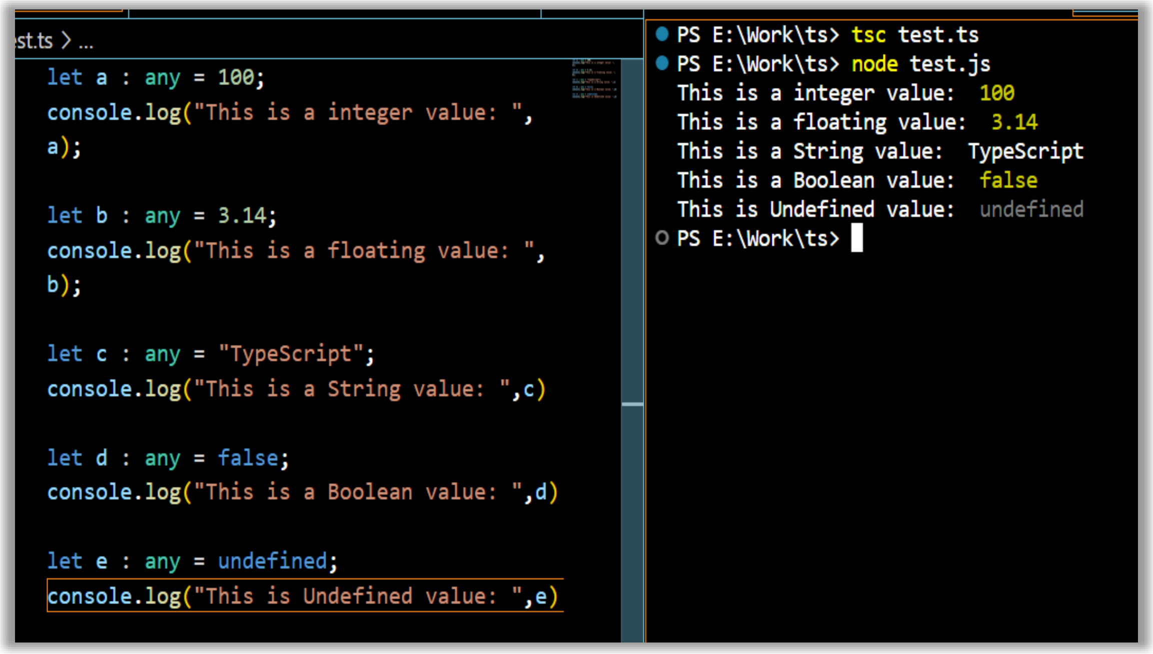
Dynamic type (any) is a built-in data type represented by the keyword as “any”.

Dynamic type (any) is a special data-type, also called as the super data type of all data type.

If variable is declared with any data type then any type of value can be assigned to that variable.

Using the dynamic type (any) type, we can basically revert TypeScript into JavaScript.

e.g.



```
let a : any = 100;
console.log("This is a integer value: ",
a);

let b : any = 3.14;
console.log("This is a floating value: ",
b);

let c : any = "TypeScript";
console.log("This is a String value: ",c)

let d : any = false;
console.log("This is a Boolean value: ",d)

let e : any = undefined;
console.log("This is Undefined value: ",e)
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
This is a integer value:  100
This is a floating value:  3.14
This is a String value:  TypeScript
This is a Boolean value:  false
This is Undefined value:  undefined
PS E:\Work\ts>
```

Type Aliases

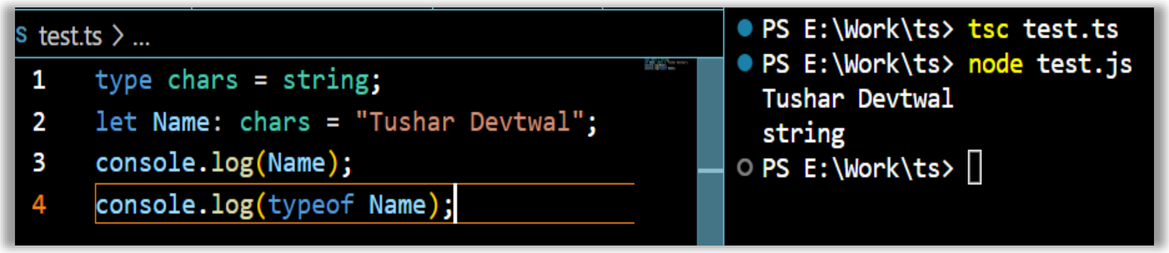
Type aliases allows you to create a new name for an existing type.

Syntax:

type alias = existingType;

The existing type can be any valid TypeScript type.

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

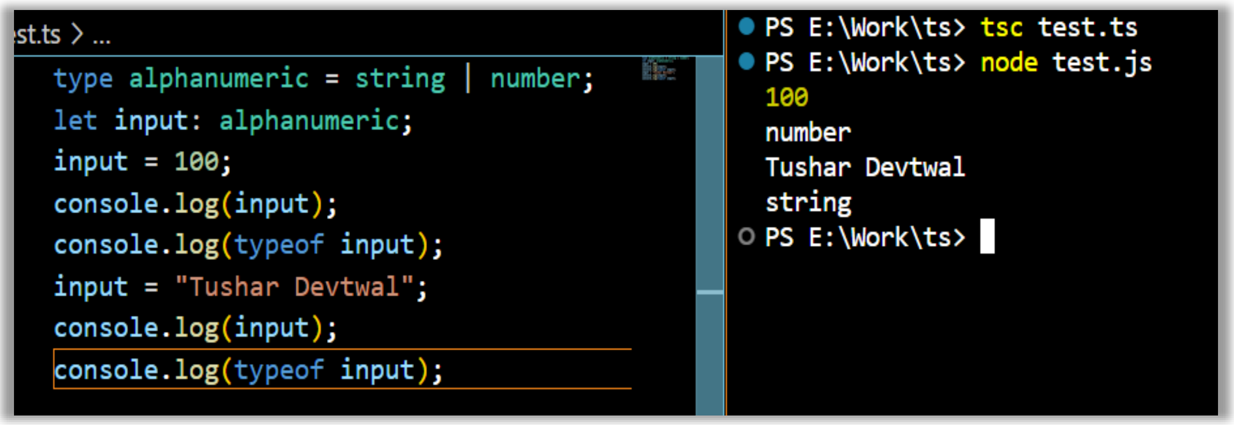


```
test.ts > ...
type chars = string;
let Name: chars = "Tushar Devtwal";
console.log(Name);
console.log(typeof Name);
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
Tushar Devtwal
string
PS E:\Work\ts>
```

It's useful to create type aliases for union type.

e.g.



```
st.ts > ...
type alphanumeric = string | number;
let input: alphanumeric;
input = 100;
console.log(input);
console.log(typeof input);
input = "Tushar Devtwal";
console.log(input);
console.log(typeof input);
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
100
number
Tushar Devtwal
string
PS E:\Work\ts>
```

Use type aliases to define new names for existing data type.

Type Aliases can reduce code duplication, keeping your code DRY (Don’t Repeat Yourself).

Interface

An interface is a syntactical contract that an entity should conform to.

In other words, an interface defines the syntax that any entity must adhere to.

Interface define properties, methods, and events, which are the members of the interface.

Interface contains only the declaration of the members.

It is the responsibility of the deriving class to define the members.

It often helps in providing a standard structure that the deriving classes would follow.

Declaring Interfaces:

The interface keyword is used to declare an interface.

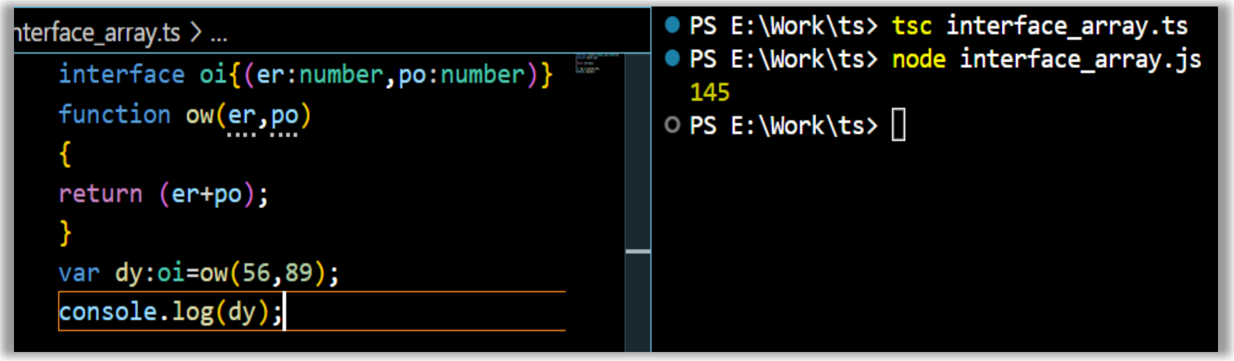
Syntax:

```
interface interface_name{


}
```

Interface for Array Type:

An interface can also define the type of an array where you can define the type of index as well as values.

e.g.



```
nterface_array.ts > ...
interface oi{(er:number,po:number)}
function ow(er,po)
{
return (er+po);
}
var dy:oi=ow(56,89);
console.log(dy);
```

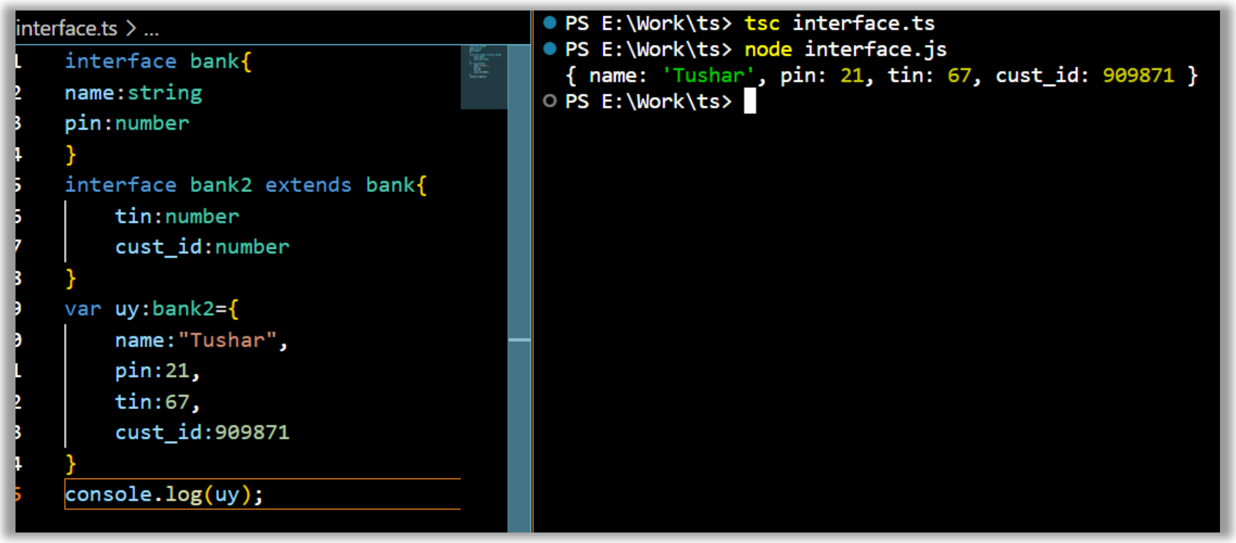
```
PS E:\Work\ts> tsc interface_array.ts
PS E:\Work\ts> node interface_array.js
145
PS E:\Work\ts>
```

Extending Interfaces:

Interfaces can extend one or more interfaces.

This makes writing interfaces flexible and reusable.

e.g.



interface.ts > ...

```
interface bank{
name:string
pin:number
}
interface bank2 extends bank{
    tin:number
    cust_id:number
}
var uy:bank2={
    name:"Tushar",
    pin:21,
    tin:67,
    cust_id:909871
}
console.log(uy);
```

```
PS E:\Work\ts> tsc interface.ts
PS E:\Work\ts> node interface.js
{ name: 'Tushar', pin: 21, tin: 67, cust_id: 909871 }
PS E:\Work\ts>
```

Generics

When writing programs, one of the most important aspects is to build reusable components.

This ensure that the program if flexible as well as scalable in the long-term.

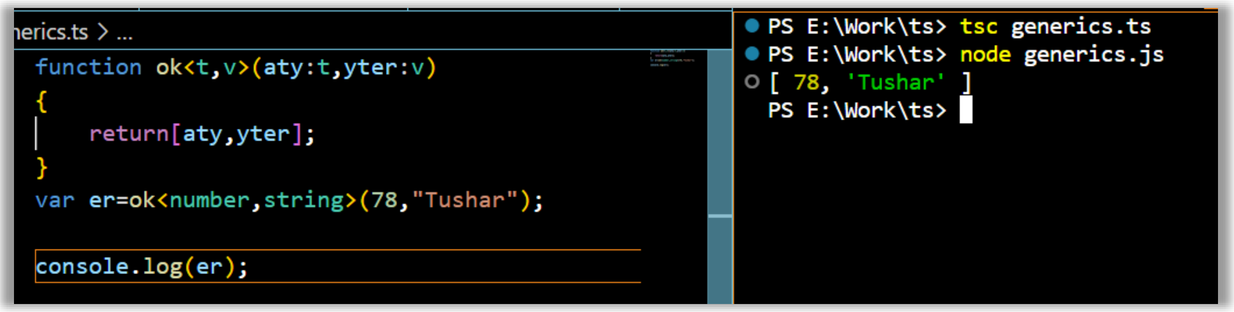
Generics offer a way to create reusable components.

Generics provide a way to make components work with any data type and not restrict to one data type.

So, components can be called or used with a variety of data types.

Generics in TypeScript is almost similar to C# generics.

TypeScript generics allow you to write the reusable and generalized form of function, classes, and interfaces.



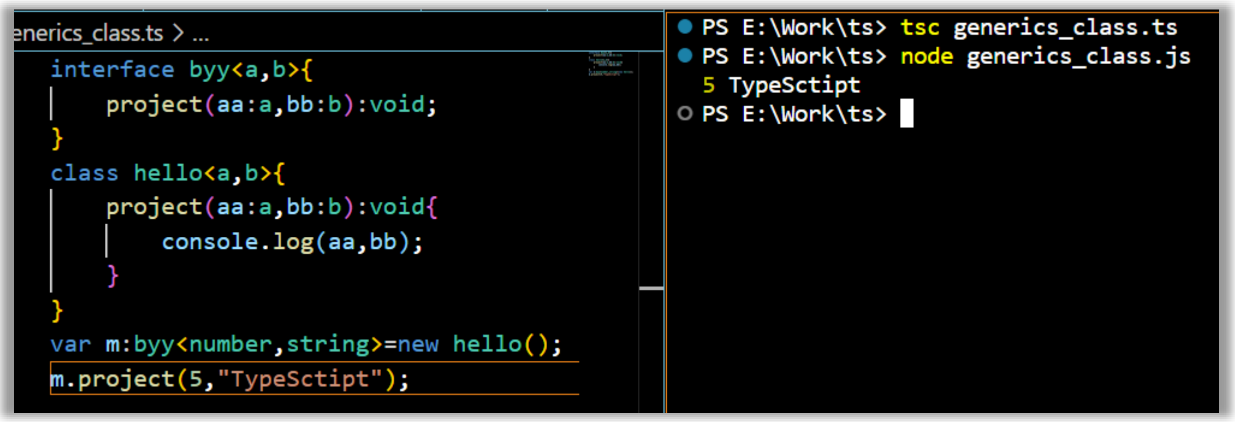
```
nerics.ts > ...
function ok<t,v>(aty:t,yter:v)
{
    return[aty,yter];
}
var er=ok<number,string>(78,"Tushar");

console.log(er);
```

```
PS E:\Work\ts> tsc generics.ts
PS E:\Work\ts> node generics.js
[ 78, 'Tushar' ]
PS E:\Work\ts>
```

TypeScript supports generic classes

The generic type parameter is specified in angle brackets after the name of the class.

A generic class can have generic field like member variables or methods.



```
enerics_class.ts > ...
interface byy<a,b>{
    project(aa:a,bb:b):void;
}
class hello<a,b>{
    project(aa:a,bb:b):void{
        console.log(aa,bb);
    }
}
var m:byy<number,string>=new hello();
m.project(5,"TypeSctipt");
```

```
PS E:\Work\ts> tsc generics_class.ts
PS E:\Work\ts> node generics_class.js
5 TypeSctipt
PS E:\Work\ts>
```

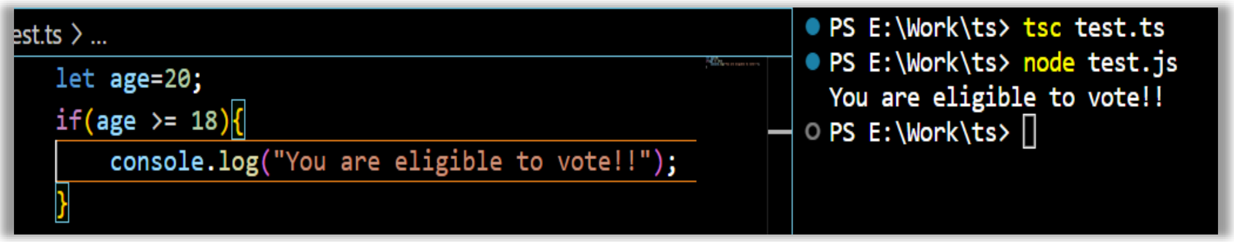
Conditional Statement

1. 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:

   Syntax:

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

e.g.

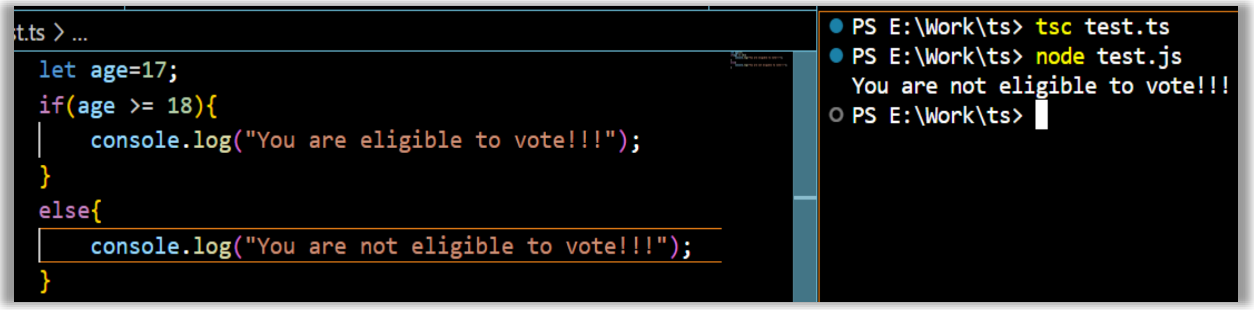


```
est.ts > ...
    let age=20;
    if(age >= 18){
        console.log("You are eligible to vote!!");
    }
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
You are eligible to vote!!
PS E:\Work\ts>
```

2. 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….ekse statement:

e.g.



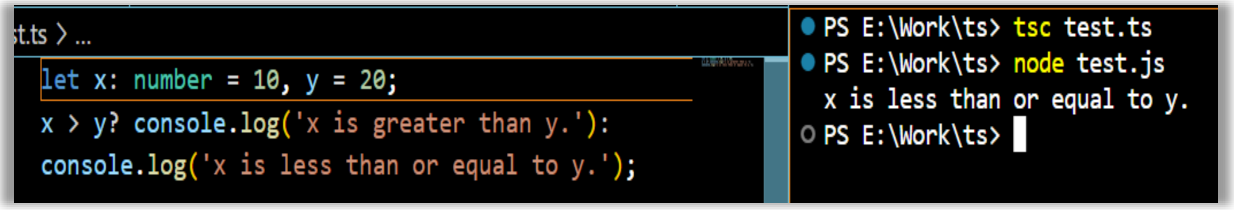
```
t.ts > ...
let age=17;
if(age >= 18){
    console.log("You are eligible to vote!!!");
}
else{
    console.log("You are not eligible to vote!!!");
}
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
You are not eligible to vote!!!
PS E:\Work\ts>
```

3. Ternary Operator ?:
   A ternary operator is denoted by '?' and is used as a short cut for an if….else statement.

   It checks for a Boolean condition and executes one of the two statements, depending on the result of the Boolean condition.

e.g.



```
t.ts > ...
let x: number = 10, y = 20;
x > y? console.log('x is greater than y.'):
console.log('x is less than or equal to y.');
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
x is less than or equal to y.
PS E:\Work\ts>
```

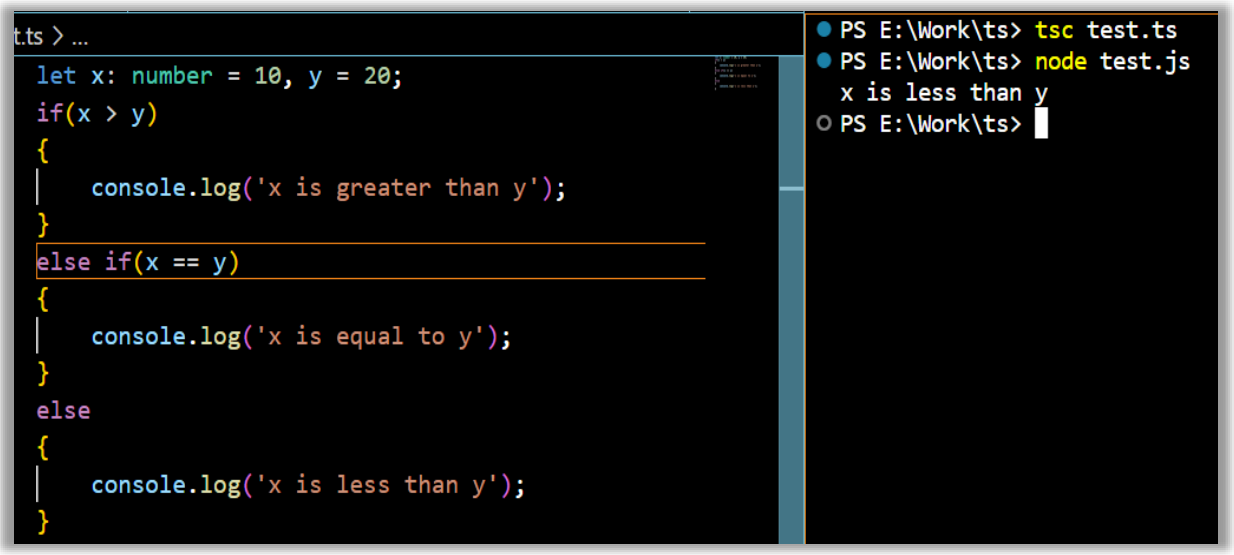
4. If…else if…else statement:

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

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

The else if statement can be used after the is statement.

e.g.



```
t.ts > ...
let x: number = 10, y = 20;
if(x > y)
{
    console.log('x is greater than y');
}
else if(x == y)
{
    console.log('x is equal to y');
}
else
{
    console.log('x is less than y');
}
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
x is less than y
PS E:\Work\ts>
```

Switch case

The switch statement is used to check for multiple values and executes set of statements for each of those values.

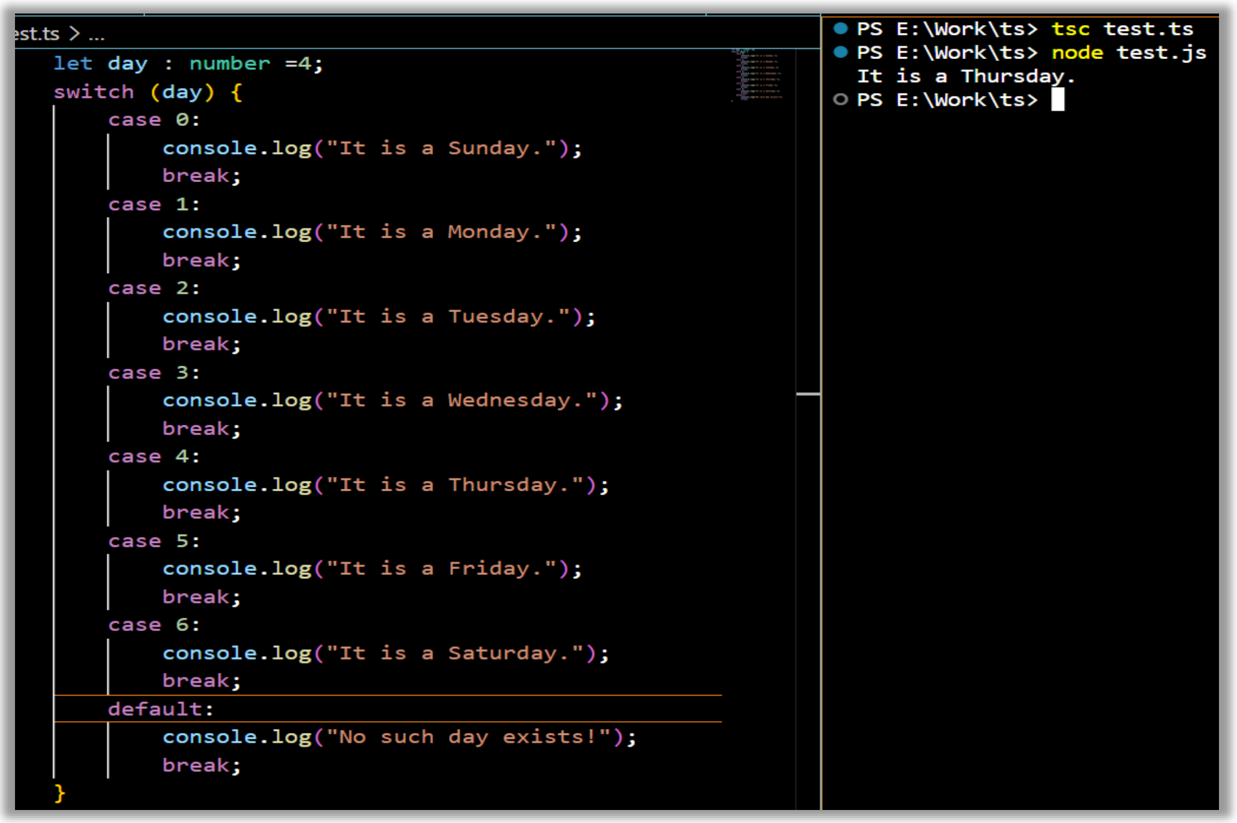
A switch statement has one block of code corresponding to each value and can have any number of such blocks.

When the match to a value if found, the corresponding block of code is executed.

The following rules are applied on the switch statement:

- The switch statement can include constant or variable expression which can return a value of any data type.
- There can be any number of case statement within a switch. The case can include a constant or an expression.
- We must use break keyword at the each case block to stop the execution of the case block.
- The return type of the switch expression and case expression must match.
- The default block is optional.

e.g.



```
est.ts > ...
  let day : number =4;
  switch (day) {
      case 0:
          console.log("It is a Sunday.");
          break;
      case 1:
          console.log("It is a Monday.");
          break;
      case 2:
          console.log("It is a Tuesday.");
          break;
      case 3:
          console.log("It is a Wednesday.");
          break;
      case 4:
          console.log("It is a Thursday.");
          break;
      case 5:
          console.log("It is a Friday.");
          break;
      case 6:
          console.log("It is a Saturday.");
          break;
      default:
          console.log("No such day exists!");
          break;
  }
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
It is a Thursday.
PS E:\Work\ts>
```

For loop

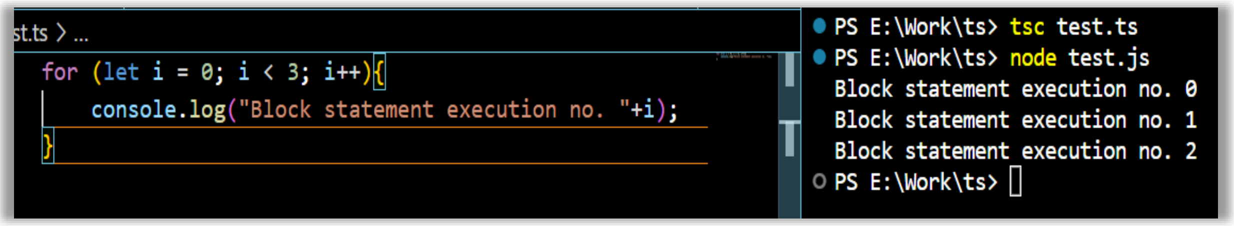
TypeScript supports the following for loops:

- for loop
- for..of loop
- for..in loop

- for loop:

The for loop is used to execute a block of code a given number of times, which is specified by a condition.

e.g.



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

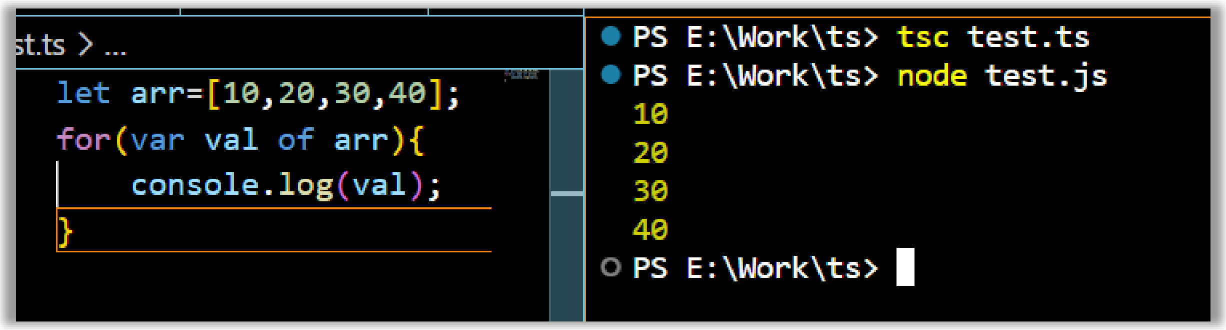
```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
Block statement execution no. 0
Block statement execution no. 1
Block statement execution no. 2
PS E:\Work\ts>
```

➢ for....of loop:

TypeScript includes the for...of loop to iterate and access elements of an array, list, or tuple collection.

The for...of loop returns elements from a collection e.g. array, list or tuple, and ss.

e.g.

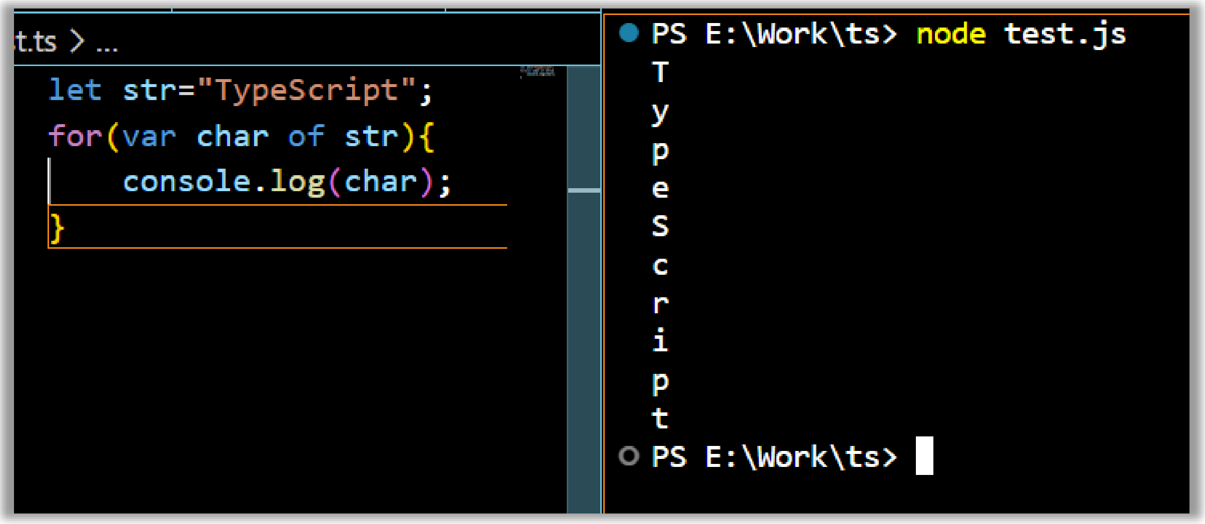


```
st.ts > ...
let arr=[10,20,30,40];
for(var val of arr){
    console.log(val);
}
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
10
20
30
40
PS E:\Work\ts>
```

The for...of loop can also return a character from a string value.

e.g.



```
t.ts > ...
let str="TypeScript";
for(var char of str){
    console.log(char);
}
```

```
PS E:\Work\ts> node test.js
T
y
p
e
S
c
r
i
p
t
PS E:\Work\ts>
```

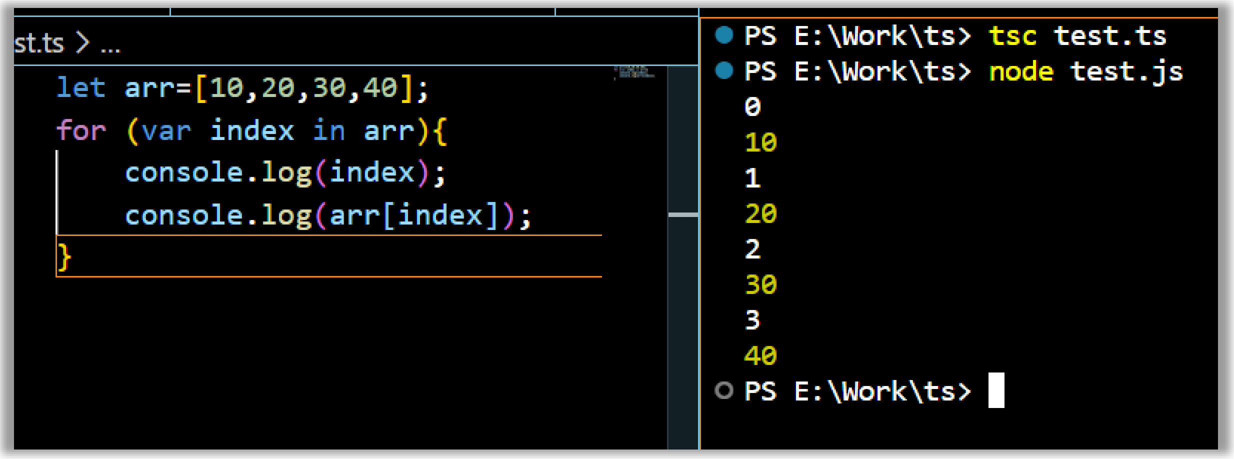
➢ for...in loop:

Another form of for loop is for...in.

This can be used with an array, list, or tuple.

The for...in loop iterates through a list or collection and returns an index on each iteration.

e.g.



```
st.ts > ...
let arr=[10,20,30,40];
for (var index in arr){
    console.log(index);
    console.log(arr[index]);
}
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
0
10
1
20
2
30
3
40
PS E:\Work\ts>
```

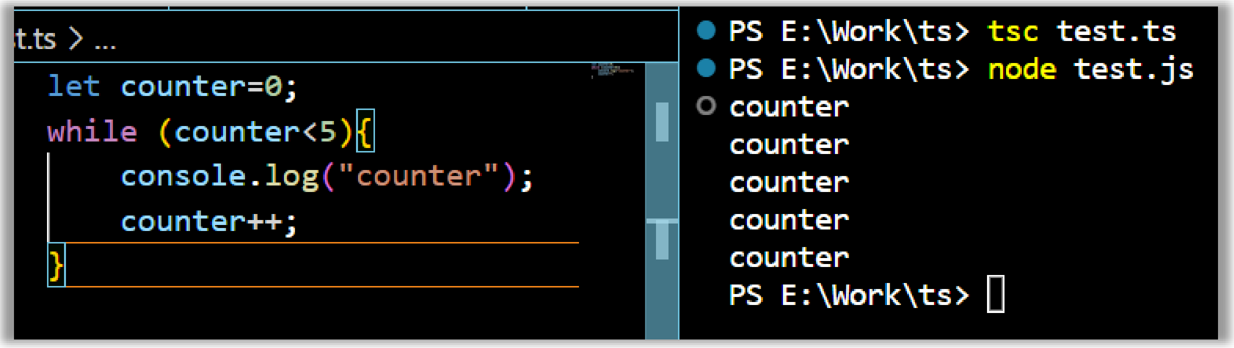
While loop

The while loop is another type of loop that checks for a specified condition before beginning to execute the block of statements.

The loop runs until the condition value is met.

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

e.g.



```
t.ts > ...
let counter=0;
while (counter<5){
    console.log("counter");
    counter++;
}
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
counter
counter
counter
counter
counter
PS E:\Work\ts>
```

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 increment it by one.
- Finally, repeat the above step as long counter is less than 5.

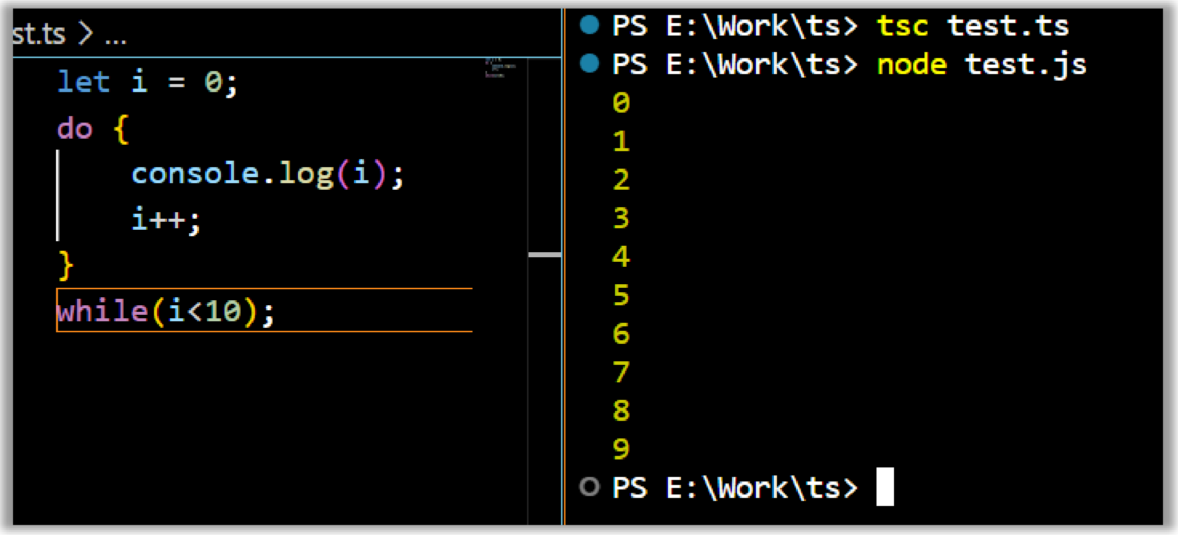
Do while loop

The do..while loop is similar to the while loop, except that the condition is given at the end of the loop.

The do..while loop runs the block of code at least once before checking for the specified condition.

For the rest of the iterations, it runs the block of code only if the specified condition is met.

e.g.



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

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
0
1
2
3
4
5
6
7
8
9
PS E:\Work\ts>
```

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 greater than or equal 10.

Classes

In object-oriented programming languags like Java and C#, classes are the fundamental entities used to create reusable components.

Functionalities are passed down to classes and object are created from classes.

TypeScript introduced classes to avail the benefit of object-oriented techniques like encapsulation and abstraction.

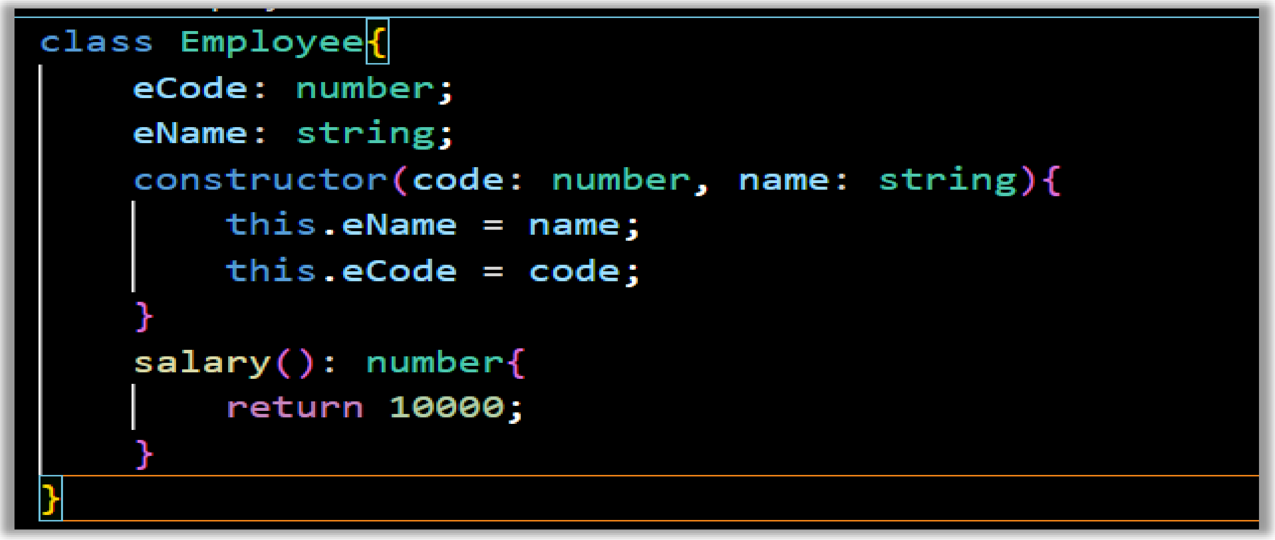
The class in TypeScript is compiled to plain JavaScript functions by the

TypeScript compiler to work across platform and browser.

A class can include the following:

- Constructor.
- Properties.
- Methods.

e.g.



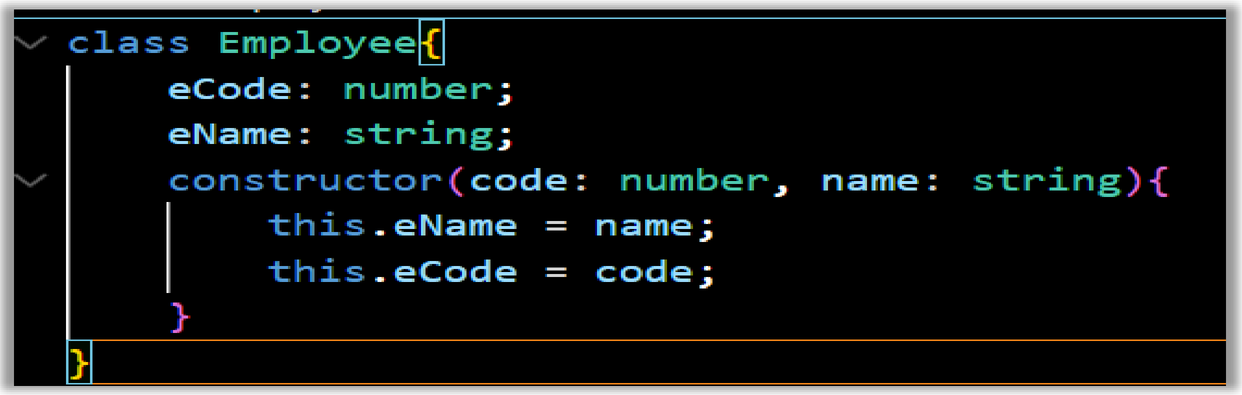
```
class Employee{
    eCode: number;
    eName: string;
    constructor(code: number, name: string){
        this.eName = name;
        this.eCode = code;
    }
    salary(): number{
        return 10000;
    }
}
```

Constructor:

The constructor is a special type of method which is called when creating an object.

In TypeScript, the constructor method is always defined with the name “constructor”.

e.g.

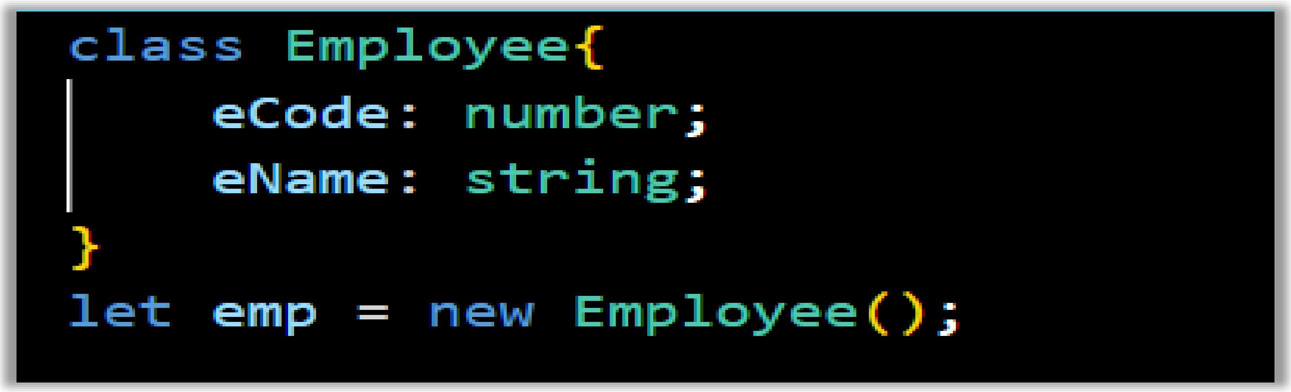


```
class Employee{
    eCode: number;
    eName: string;
    constructor(code: number, name: string){
        this.eName = name;
        this.eCode = code;
    }
}
```

Creating an Object of Class:

An object of the class can be created using the new keyword.

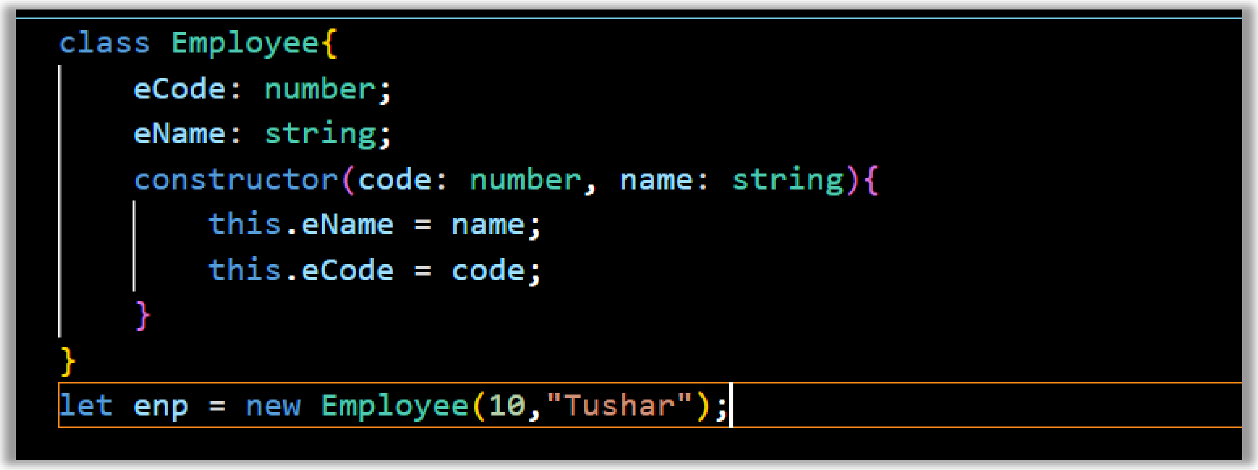
e.g.



```
class Employee{
    eCode: number;
    eName: string;
}
let emp = new Employee();
```

in the class a parameterized constructor, then we can pass values while creating the object.

e.g.

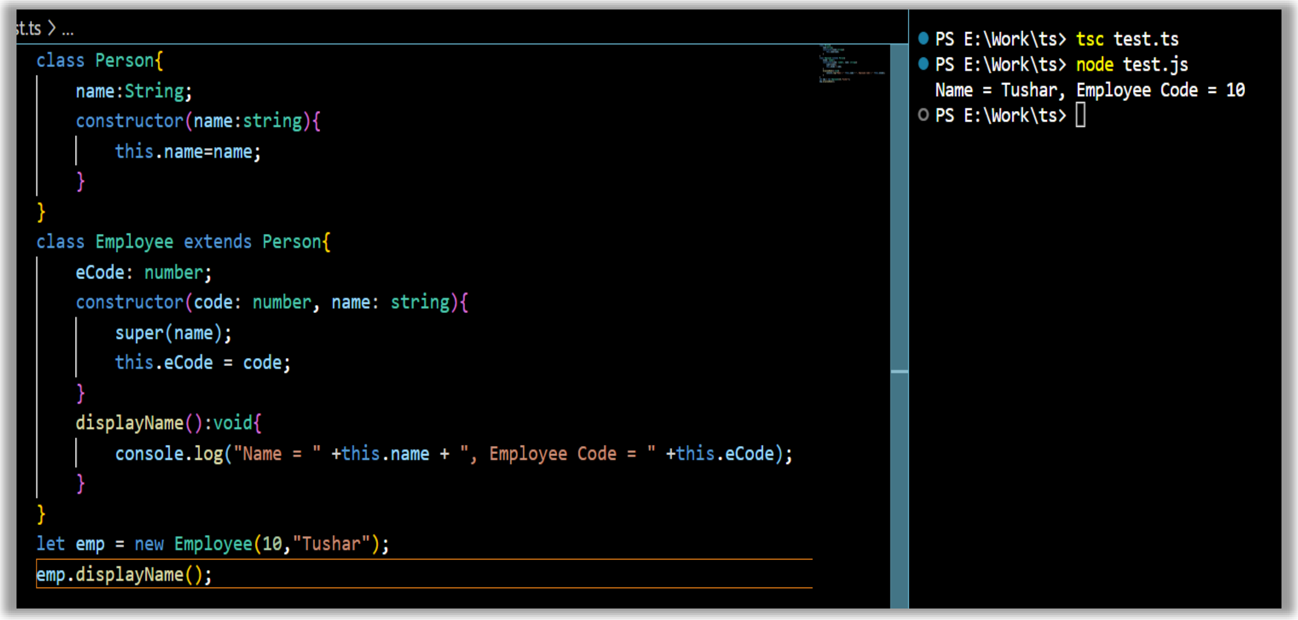


```
class Employee{
    eCode: number;
    eName: string;
    constructor(code: number, name: string){
        this.eName = name;
        this.eCode = code;
    }
}
let enp = new Employee(10,"Tushar");
```

Inheritance:

Just like object-oriented languages such as Java and C#, TypeScript classes can be extended to create new classes with inheritance, using the keyword extends.

e.g.



```
t.ts > ...
class Person{
    name:String;
    constructor(name:string){
        this.name=name;
    }
}
class Employee extends Person{
    eCode: number;
    constructor(code: number, name: string){
        super(name);
        this.eCode = code;
    }
    displayName():void{
        console.log("Name = " +this.name + ", Employee Code = " +this.eCode);
    }
}
let emp = new Employee(10,"Tushar");
emp.displayName();
```

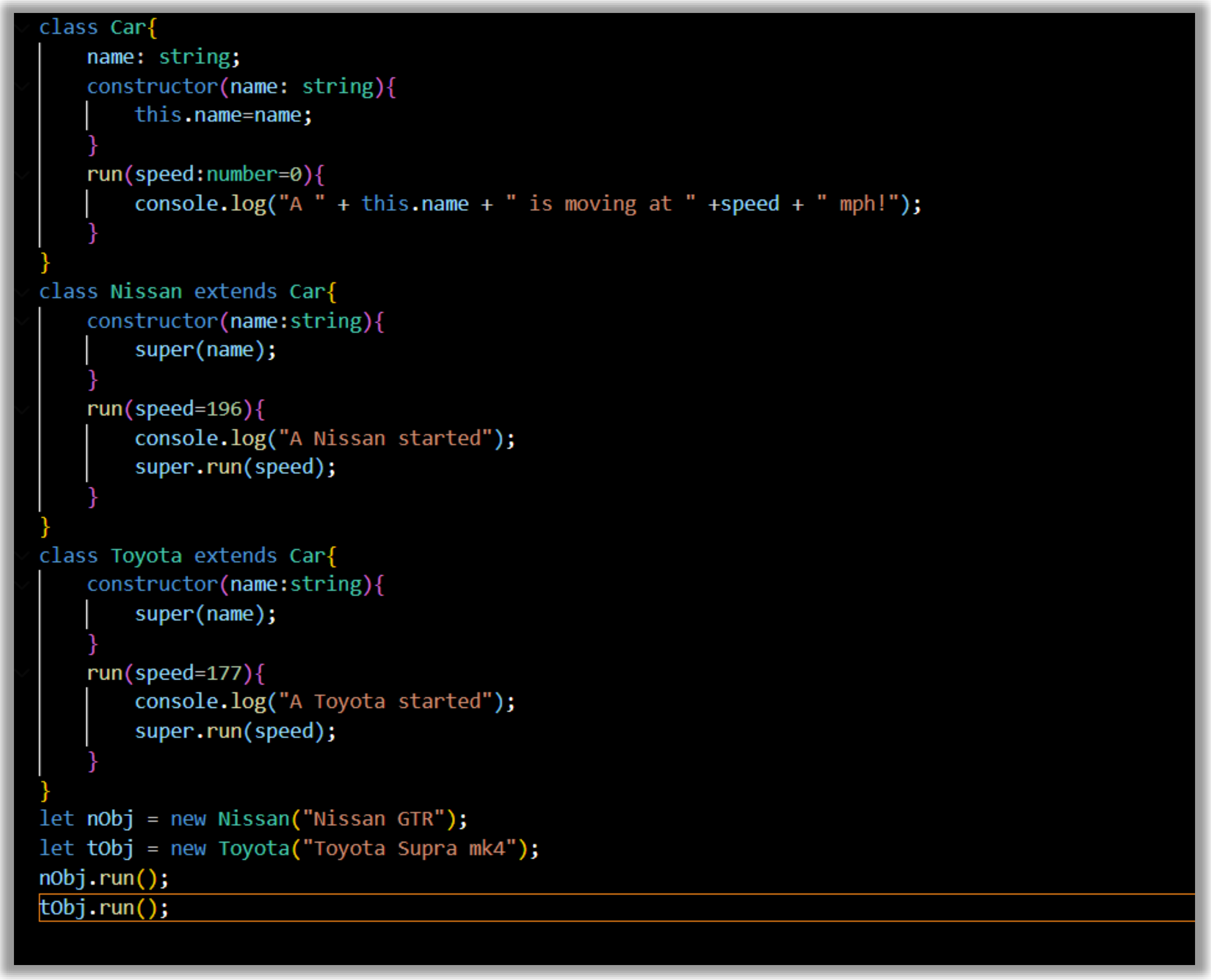
```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
Name = Tushar, Employee Code = 10
PS E:\Work\ts>
```

The super keyword is used to call the parent constructor and passes the property values.

We must call super() method first before assigning values to properties in the constructor of the derived class.

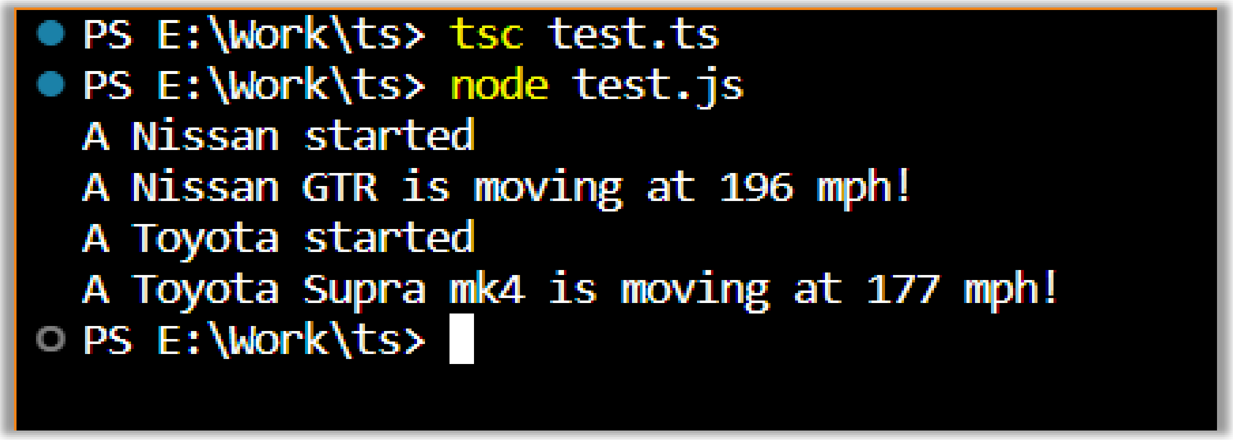
Method Overriding:

When a child class defines its own implementation of a method from the parent class, it is called method overriding.



```
class Car{
    name: string;
    constructor(name: string){
        this.name=name;
    }
    run(speed:number=0){
        console.log("A " + this.name + " is moving at " +speed + " mph!");
    }
}
class Nissan extends Car{
    constructor(name:string){
        super(name);
    }
    run(speed=196){
        console.log("A Nissan started");
        super.run(speed);
    }
}
class Toyota extends Car{
    constructor(name:string){
        super(name);
    }
    run(speed=177){
        console.log("A Toyota started");
        super.run(speed);
    }
}
let nObj = new Nissan("Nissan GTR");
let tObj = new Toyota("Toyota Supra mk4");
nObj.run();
tObj.run();
```

Output:



```
● PS E:\Work\ts> tsc test.ts
● PS E:\Work\ts> node test.js
  A Nissan started
  A Nissan GTR is moving at 196 mph!
  A Toyota started
  A Toyota Supra mk4 is moving at 177 mph!
○ PS E:\Work\ts> █
```

Abstract Classes

We define an abstract class in TypeScript using the abstract keyword.

Abstract classes are mainly for inheritance where other classes may derive from them.

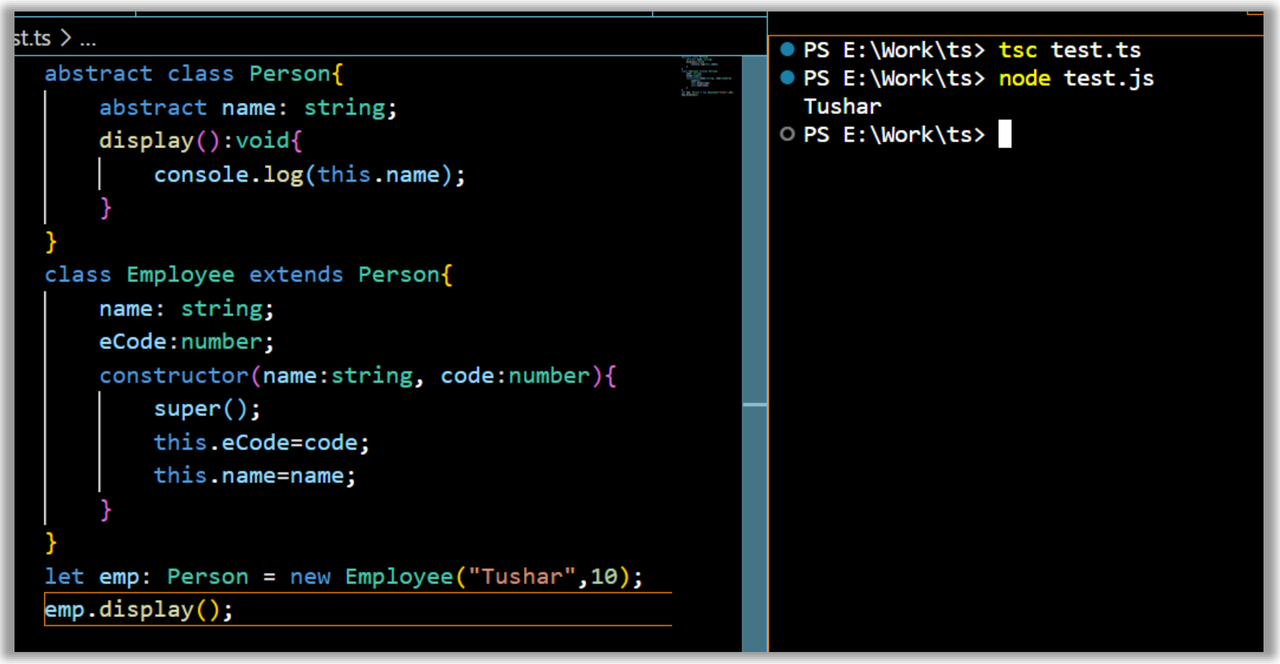
We cannot create an instance of an abstract class.

An abstract class typically includes one or more abstract methods or property declarations.

The class which extends the abstract class must define all the abstract methods.

e.g.

The following abstract class declares a normal method display().



```
abstract class Person{
    abstract name: string;
    display():void{
        console.log(this.name);
    }
}
class Employee extends Person{
    name: string;
    eCode:number;
    constructor(name:string, code:number){
        super();
        this.eCode=code;
        this.name=name;
    }
}
let emp: Person = new Employee("Tushar",10);
emp.display();
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
Tushar
PS E:\Work\ts>
```

Tuples

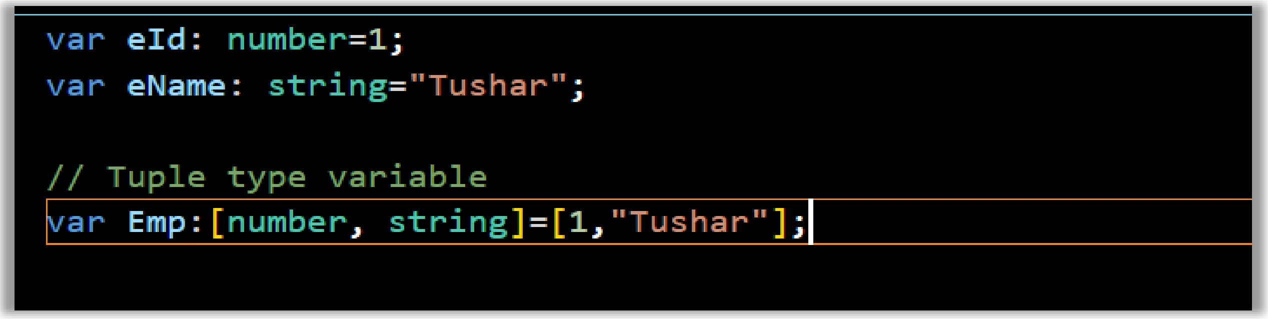
TypeScript introduced a new data type called Tuple.

A tuple works like an array with some additional considerations:

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

Tuple can contains two values of different data type.

e.g.



```
var eId: number=1;
var eName: string="Tushar";

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

Here, we declare and assigned two variables to id and name of an employee.

The same thing can be achieved by using a single tuple type variable.

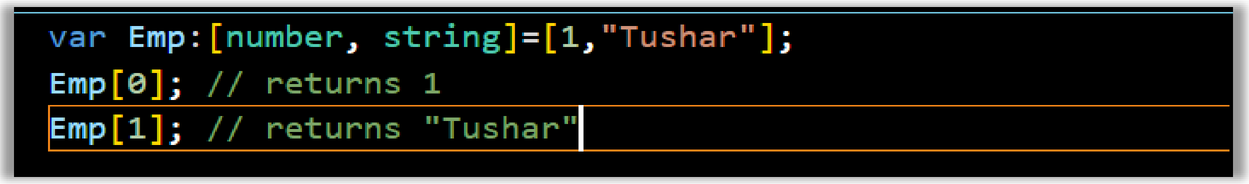
Employee is the tuple type variable with two values of numbers and string type.

Thus, removing the need to declare two different variables.

Accessing Tuple Elements:

We can access tuple elements using index, the same way as an array.

An index starts from zero.

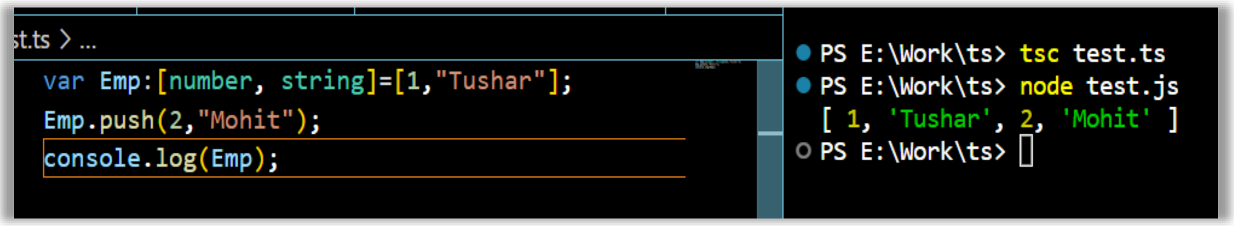


```
var Emp:[number, string]=[1,"Tushar"];
Emp[0]; // returns 1
Emp[1]; // returns "Tushar"
```

Add elements into tuple:

1. push():

You can add new elements to a tuple using the push() method.

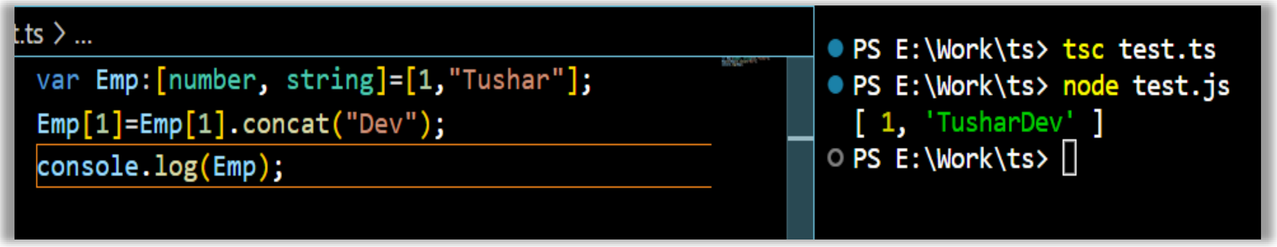


```
st.ts > ...
var Emp:[number, string]=[1,"Tushar"];
Emp.push(2,"Mohit");
console.log(Emp);
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
[ 1, 'Tushar', 2, 'Mohit' ]
PS E:\Work\ts>
```

2. pop()/concat():

You can remove an element from a tuple using concat() method.



```
t.ts > ...
var Emp:[number, string]=[1,"Tushar"];
Emp[1]=Emp[1].concat("Dev");
console.log(Emp);
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
[ 1, 'TusharDev' ]
PS E:\Work\ts>
```

Enums

Enums or enumeration are a new data type supported in TypeScript.

Most object-oriented languages like Java and C# use enums.

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

A enum is a group of named constant values.

To define an enum, you follow these steps:

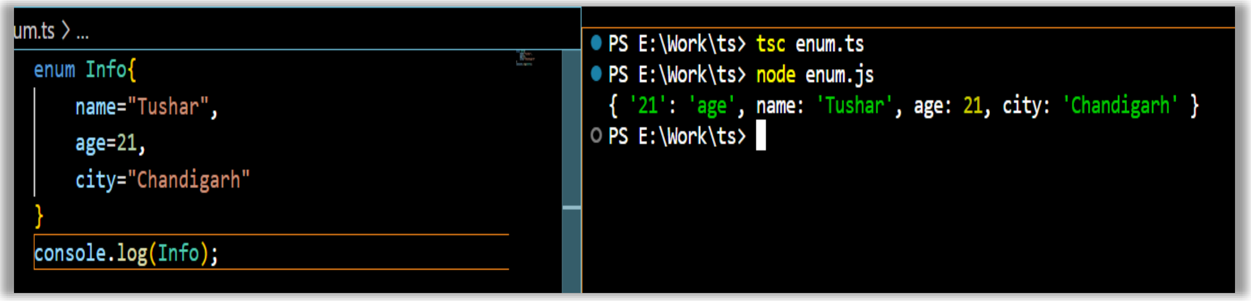
- First, use the enum keyword followed by the name of the enum.
- Then, define constant values for the enum.

Syntax:

Enum name {constant1, constant2,..};

Enums are a special feature that TypeScript brings to JavaScript.

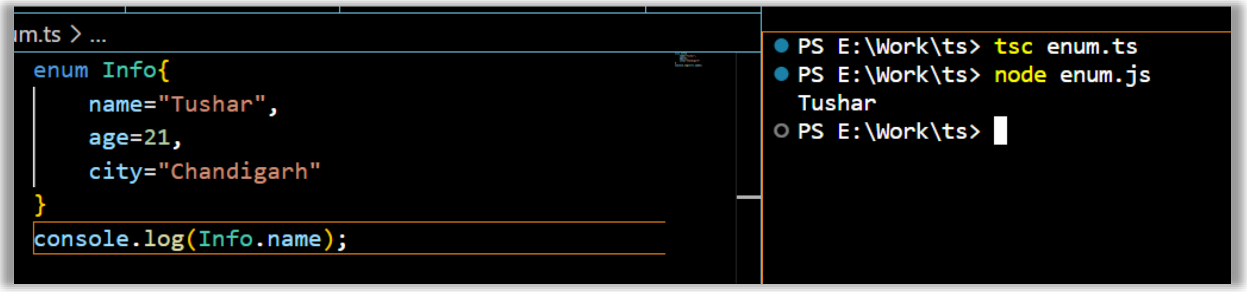
Enums allow us to declare a collection of related values that can be numbers or strings, as a set of named constants.



```
um.ts > ...
enum Info{
    name="Tushar",
    age=21,
    city="Chandigarh"
}
console.log(Info);
```

```
PS E:\Work\ts> tsc enum.ts
PS E:\Work\ts> node enum.js
{ '21': 'age', name: 'Tushar', age: 21, city: 'Chandigarh' }
PS E:\Work\ts>
```

You can access the values by following:



```
um.ts > ...
enum Info{
    name="Tushar",
    age=21,
    city="Chandigarh"
}
console.log(Info.name);
```

```
PS E:\Work\ts> tsc enum.ts
PS E:\Work\ts> node enum.js
Tushar
PS E:\Work\ts>
```

Union Type

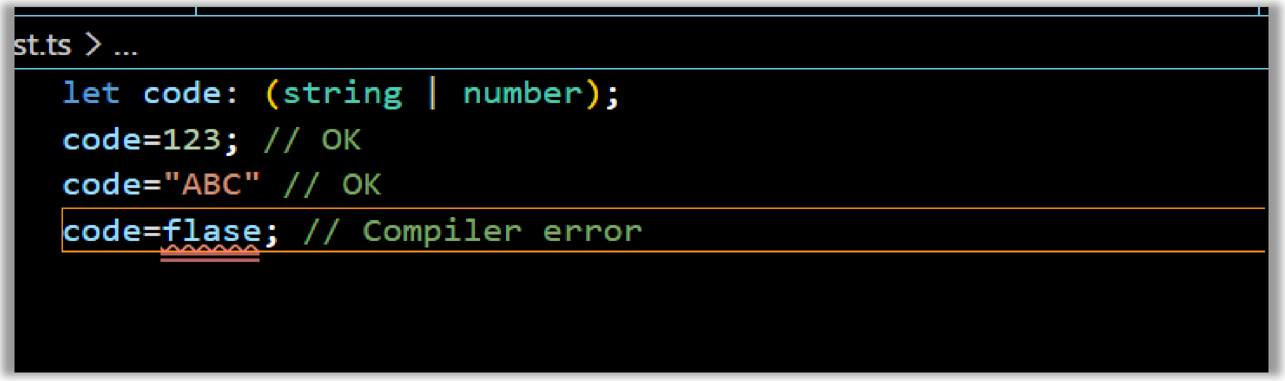
TypeScript allows us to use more than one data type for a variable or a function parameter. This is called union type.

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

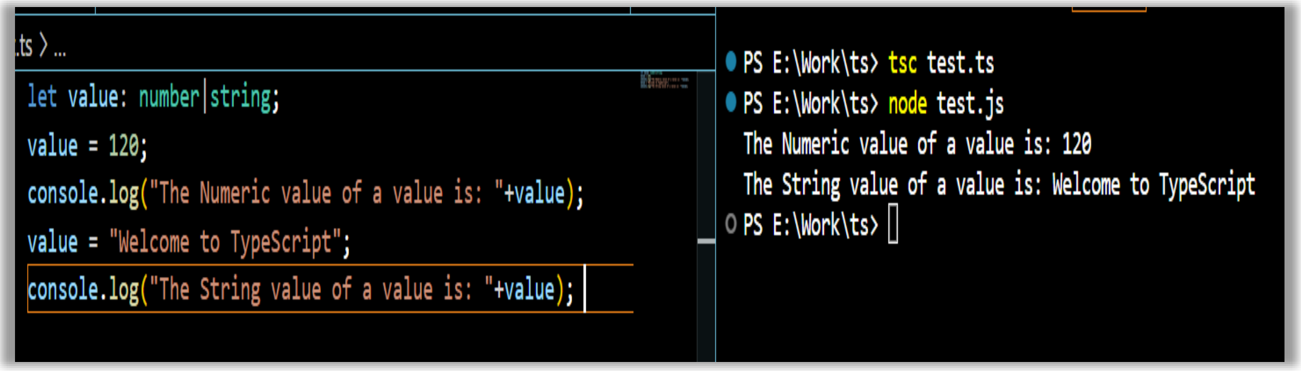
Syntax:

```
    (type1 | type2 | type3.. | typeN)
```

e.g.



```
st.ts > ...
let code: (string | number);
code=123; // OK
code="ABC" // OK
code=flase; // Compiler error
```



```
.ts > ...
let value: number|string;
value = 120;
console.log("The Numeric value of a value is: "+value);
value = "Welcome to TypeScript";
console.log("The String value of a value is: "+value);
```

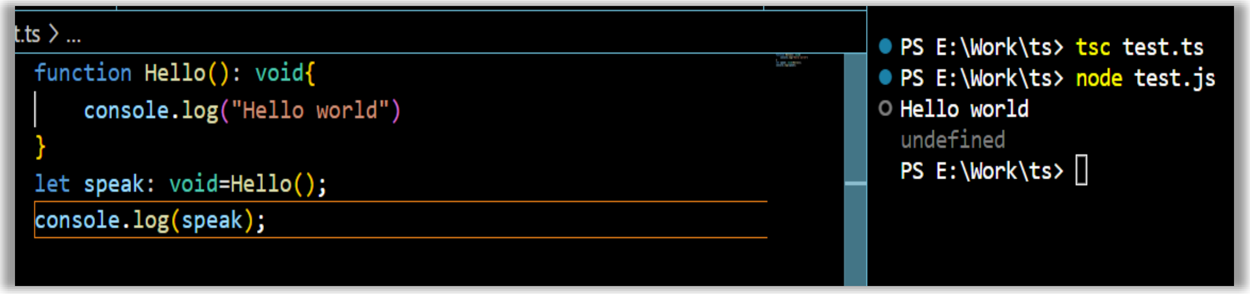
```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
The Numeric value of a value is: 120
The String value of a value is: Welcome to TypeScript
PS E:\Work\ts>
```

Void Type

Similar to languages like Java, void is used where there is no data.

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

e.g.

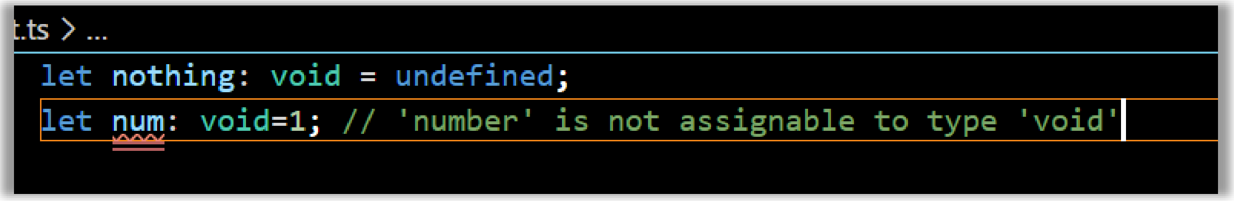


```
.ts > ...
function Hello(): void{
    console.log("Hello world")
}
let speak: void=Hello();
console.log(speak);
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
Hello world
undefined
PS E:\Work\ts>
```

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

e.g.



```
t.ts > ...
let nothing: void = undefined;
let num: void=1; // 'number' is not assignable to type 'void'
```

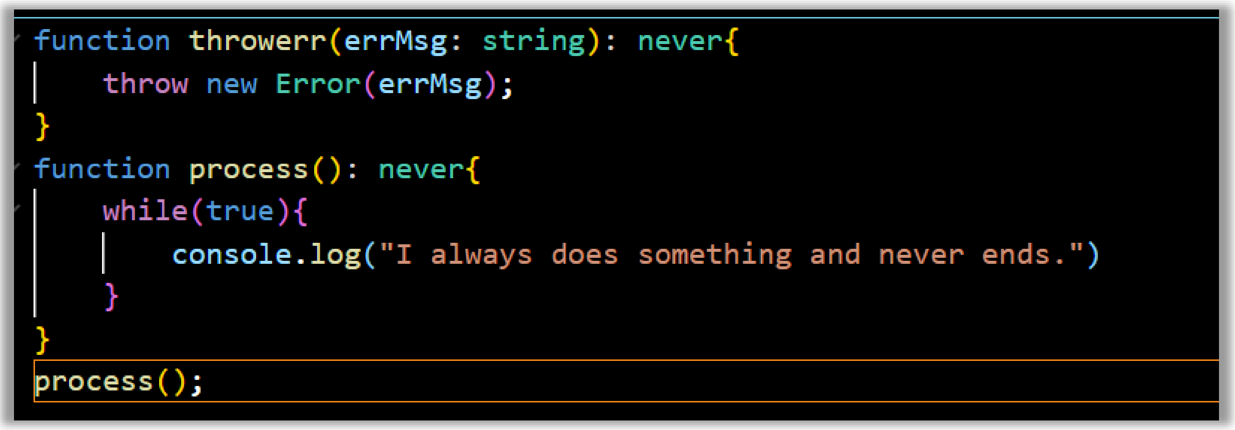
Never Type

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.

e.g.



```
function throwerr(errMsg: string): never{
    throw new Error(errMsg);
}
function process(): never{
    while(true){
        console.log("I always does something and never ends.")
    }
}
process();
```

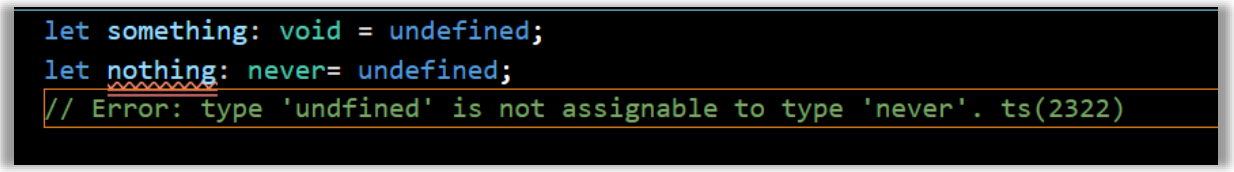
Thus, never type is used to indicate the value that will never occur from a function.

Different Between never and void:

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

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

e.g.



```
let something: void = undefined;
let nothing: never= undefined;
// Error: type 'undfined' is not assignable to type 'never'. ts(2322)
```

Narrowing

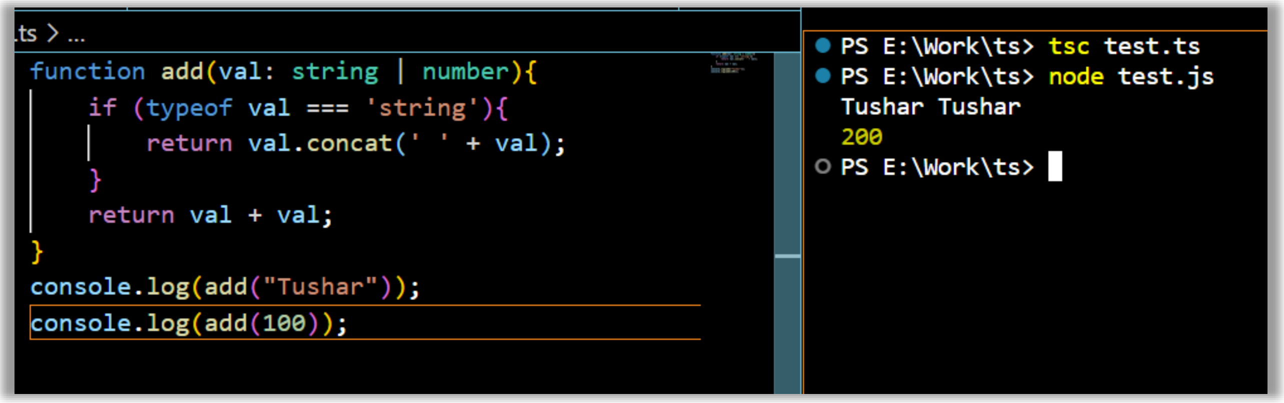
In a TypeScript program, a variable can move from a less precise type to a precise type.

This process is called narrowing.

e.g.

string | number to more specific types when we use if-statements with typeof:

e.g.



```
.ts > ...
function add(val: string | number){
    if (typeof val === 'string'){
        return val.concat(' ' + val);
    }
    return val + val;
}
console.log(add("Tushar"));
console.log(add(100));
```

```
PS E:\Work\ts> tsc test.ts
PS E:\Work\ts> node test.js
Tushar Tushar
200
PS E:\Work\ts>
```