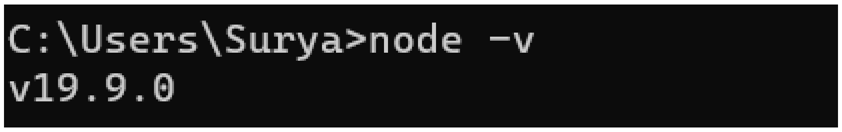
<u>TypeScript</u>

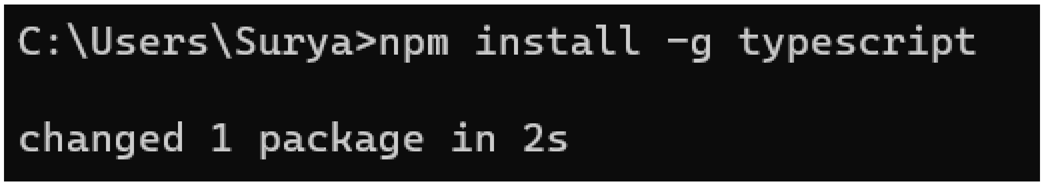
<u>Install</u>

Step-1 Install Node.js. It is used to setup TypeScript on our local computer.
Command to verify the installation was successful:



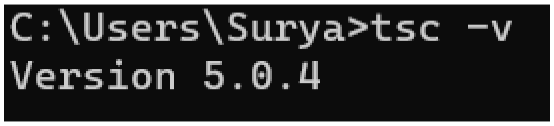
```
C:\Users\Surya>node -v
v19.9.0
```

Step-2 Install TypeScript. To install TypeScript, the following command is used.



```
C:\Users\Surya>npm install -g typescript

changed 1 package in 2s
```

Command to verify the installation was successful:



```
C:\Users\Surya>tsc -v
Version 5.0.4
```

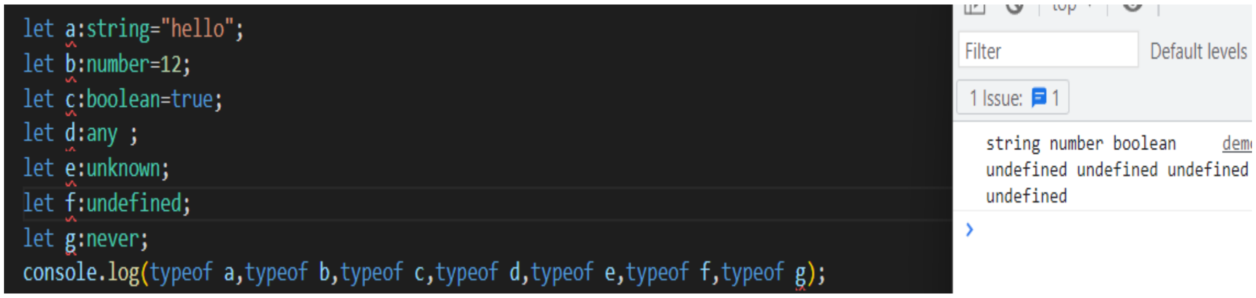
<u>Basic:</u>

TypeScript is a superset of JavaScript that adds strong type checking and is compiled into plain JavaScript code.TypeScript has all the features of JavaScript as well as some additional features.

<u>Data type:</u>

1. String- Text value
2. Number- Whole numbers and floating point values
3. Boolean- True or False values
4. Array- Set of data type.
5. Object- Set of keys and values.
6. Any- Any is a type that disables type checking and effectively allows all types to be used.
7. Unknown- Unknown is similar to any. TypeScript will prevent unknown types from being used
8. Never- never effectively throws an error whenever it is defined.
9. Undefine -When variable is not defined.

<u>Code:</u>



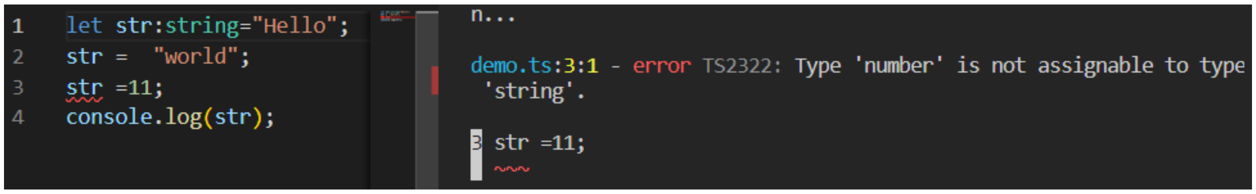
```
let a:string="hello";
let b:number=12;
let c:boolean=true;
let d:any ;
let e:unknown;
let f:undefined;
let g:never;
console.log(typeof a,typeof b,typeof c,typeof d,typeof e,typeof f,typeof g);
```

```
string number boolean
undefined undefined undefined
undefined
```

Variables:

The type syntax for declaring a variable in TypeScript is to include a colon (:) after the variable name, followed by its type. The TypeScript compiler will generate errors, if we attempt to assign a value to a variable that is not of the same type. The Strong typing syntax ensures that the types specified on either side of the assignment operator (=) are the same.

Code:



```
let str:string="Hello";
str =  "world";
str =11;
console.log(str);
```

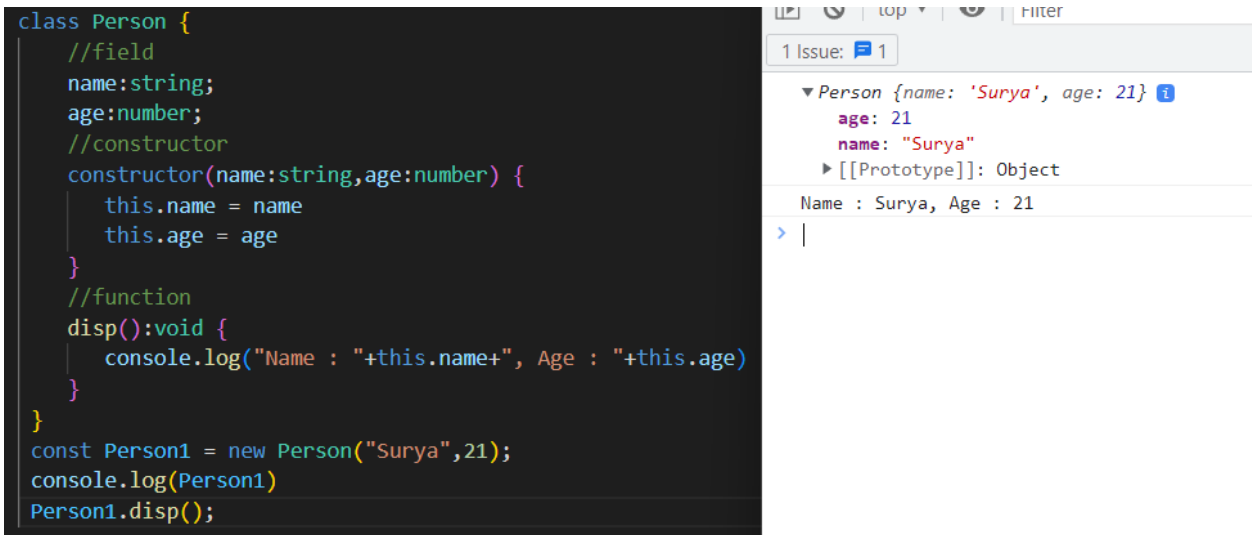
```
n...

demo.ts:3:1 - error TS2322: Type 'number' is not assignable to type
 'string'.

3 str =11;
  ~~~
```

Classes

TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. There are three main visibility modifiers in TypeScript:

1. public - (default) allows access to the class member from anywhere.
2. private - only allows access to the class member from within the class.
3. protected - allows access to the class member from itself and any classes that inherit it, which is covered in the inheritance section below.
4. Readonly - the readonly keyword can prevent class members from being changed.

Code:



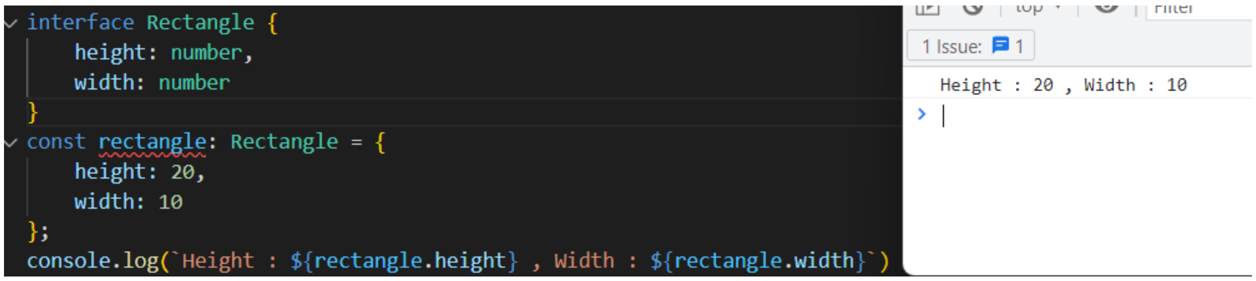
```
class Person {
    //field
    name:string;
    age:number;
    //constructor
    constructor(name:string,age:number) {
        this.name = name
        this.age = age
    }
    //function
    disp():void {
        console.log("Name : "+this.name+", Age : "+this.age)
    }
 }
 const Person1 = new Person("Surya",21);
 console.log(Person1)
 Person1.disp();
```

```
▼Person {name: 'Surya', age: 21}
    age: 21
    name: "Surya"
  ▶[[Prototype]]: Object
Name : Surya, Age : 21
```

Interfaces

In TypeScript, an interface is an abstract type that tells the compiler which property names a given object can have. TypeScript creates implicit interfaces when you define an object with properties.

Code:



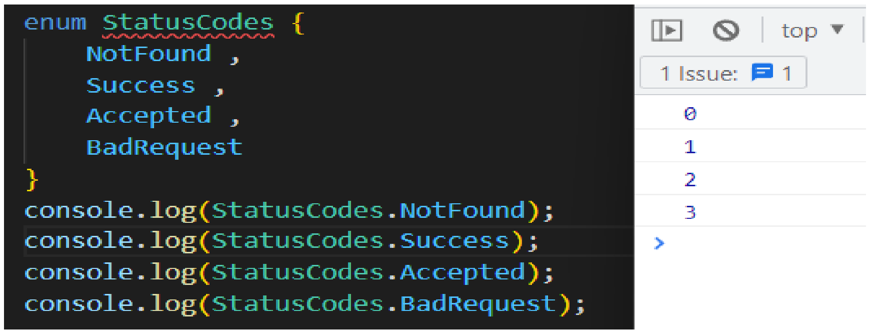
```
interface Rectangle {
    height: number,
    width: number
}
const rectangle: Rectangle = {
    height: 20,
    width: 10
};
console.log(`Height : ${rectangle.height} , Width : ${rectangle.width}`)
```

```
Height : 20 , Width : 10
```

Enum:

Enums allow a developer to define a set of named constants. TypeScript provides both numeric and string-based enums.You can set the value of the first numeric enum and have it auto increment from that.By default, enums will initialize the first value to 0.

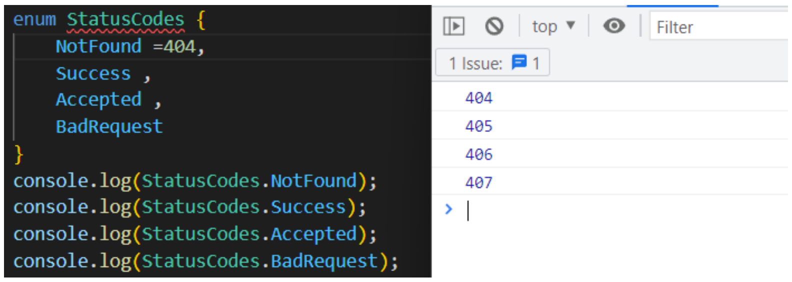
Code(without initialize):



```
enum StatusCodes {
    NotFound ,
    Success ,
    Accepted ,
    BadRequest
}
console.log(StatusCodes.NotFound);
console.log(StatusCodes.Success);
console.log(StatusCodes.Accepted);
console.log(StatusCodes.BadRequest);
```

```
0
1
2
3
```

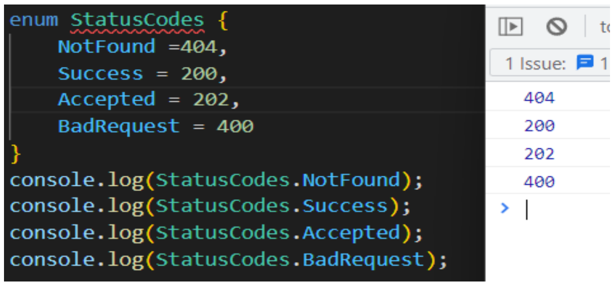
Code(with initialize first one):



```
enum StatusCodes {
    NotFound =404,
    Success ,
    Accepted ,
    BadRequest
}
console.log(StatusCodes.NotFound);
console.log(StatusCodes.Success);
console.log(StatusCodes.Accepted);
console.log(StatusCodes.BadRequest);
```

```
404
405
406
407
```

Code(with initialize each one):



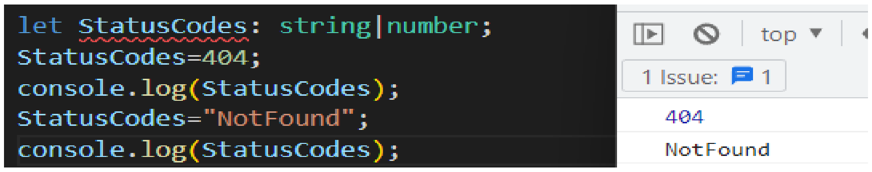
```
enum StatusCodes {
    NotFound =404,
    Success = 200,
    Accepted = 202,
    BadRequest = 400
}
console.log(StatusCodes.NotFound);
console.log(StatusCodes.Success);
console.log(StatusCodes.Accepted);
console.log(StatusCodes.BadRequest);
```

```
404
200
202
400
```

Unions:

Union types are used when a value can be more than a single type. Such as when a property would be string or number.

Code:



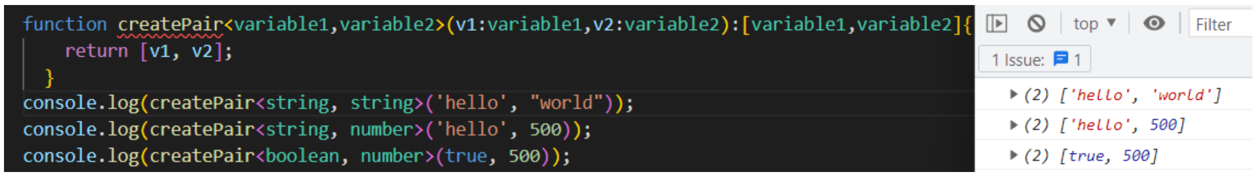
```
let StatusCodes: string|number;
StatusCodes=404;
console.log(StatusCodes);
StatusCodes="NotFound";
console.log(StatusCodes);
```

```
1 Issue: 1
404
NotFound
```

Generics:

Generics allow creating 'type variables' which can be used to create classes, functions & type aliases that don't need to explicitly define the types that they use.

Code:



```
function createPair<variable1,variable2>(v1:variable1,v2:variable2):[variable1,variable2]{
    return [v1, v2];
 }
console.log(createPair<string, string>('hello', "world"));
console.log(createPair<string, number>('hello', 500));
console.log(createPair<boolean, number>(true, 500));
```

```
1 Issue: 1
▸ (2) ['hello', 'world']
▸ (2) ['hello', 500]
▸ (2) [true, 500]
```