The Ultimate TypeScript Cheat Sheet [2022]

TypeScript is an object-oriented, compiled, and strongly typed language. A Javascript superset, it features many of Javascript's features, plus a few more. As a result, all existing JavaScript programs are also TypeScript programs.

The TypeScript code is compatible with any browser, OS, or environment that can run JavaScript. As a result, the language is increasing in popularity – many tech giants, like Meta, Google, and Microsoft are leveraging TypeScript for its robust features.

Learning TypeScript opens up many job opportunities with lucrative salaries. Interested in learning the TypeScript list of commands or want a quick reference to any TypeScript syntax? We've got you covered with this comprehensive TypeScript cheat sheet.

Our TypeScript React cheat sheet will cover TypeScript and React TypeScript types. But first, let's explore how TypeScript works and why it's superior to Javascript.

Setting up TypeScript

You'll need the following tools to install TypeScript:

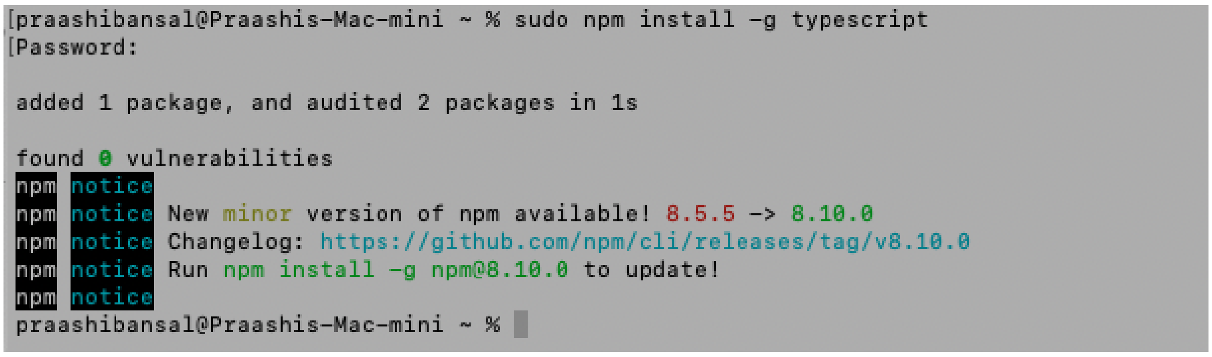
- **Node.js**: the environment where you'll run the TypeScript compiler – no technical knowledge needed!
- **TypeScript compiler:** module that converts TypeScript code into JavaScript.

- **Visual Studio Code or VS code**: code editor to write TypeScript code. VS Code is preferred over other options because it allows you to use the Live Server extension to speed up the development process.

Let's get started:

1. Install Node.js's latest version of Node.js. Complete the installation process. You can verify it by executing the "node -v" command on your terminal.
2. After that, execute the following command:

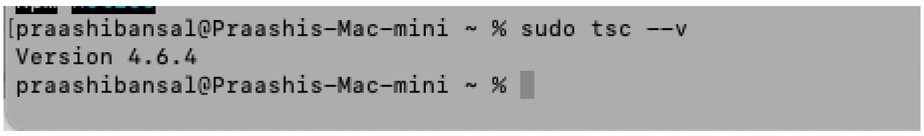
```
npm install -g TypeScript
```



```
[praashibansal@Praashis-Mac-mini ~ % sudo npm install -g typescript
[Password:

added 1 package, and audited 2 packages in 1s

found 0 vulnerabilities
npm notice
npm notice New minor version of npm available! 8.5.5 -> 8.10.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v8.10.0
npm notice Run npm install -g npm@8.10.0 to update!
npm notice
praashibansal@Praashis-Mac-mini ~ %
```

3. Check the installed version using the following command:

```
tsc --v
```



```
[praashibansal@Praashis-Mac-mini ~ % sudo tsc --v
Version 4.6.4
praashibansal@Praashis-Mac-mini ~ %
```

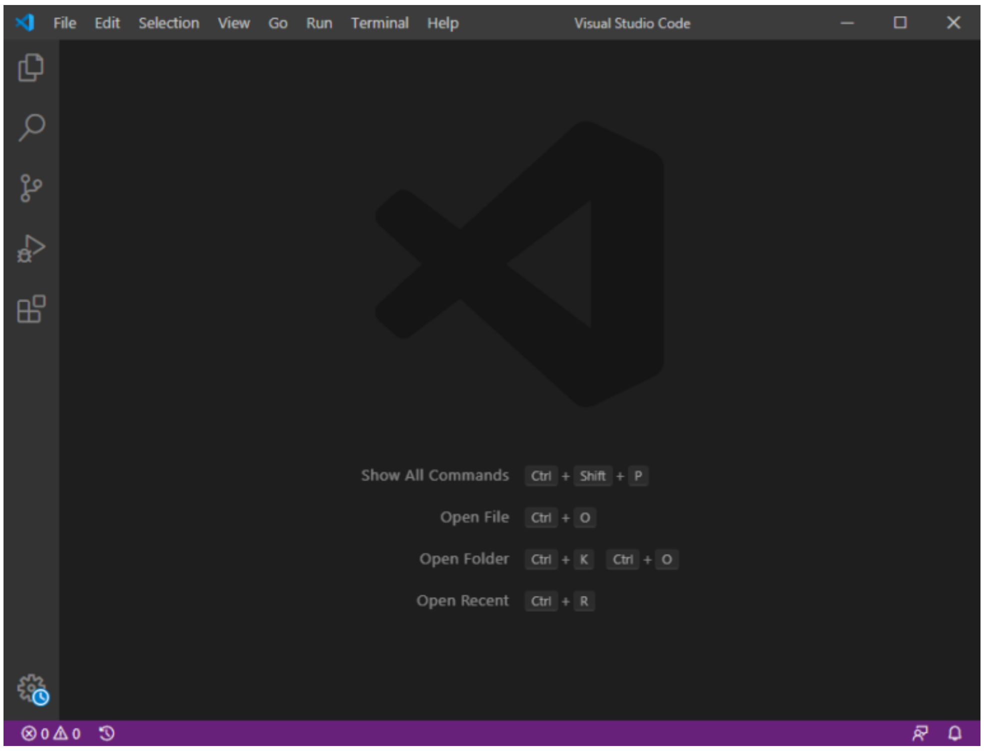
4. Add the following path to the PATH variable.

"C:\Users\<user>\AppData\Roaming\npm"
Note: for this cheat sheet, we are using Version 4.0.2.

5. Install the "ts-node" module globally using the following command:

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

6. Install VS Code and download the latest version as per your required platform. After the installation process finishes, launch the VS Code.



7. To install the live Server extension, go to the Extensions tab, search for liver Server and click the install button.

All set up? Now, let's get into coding.

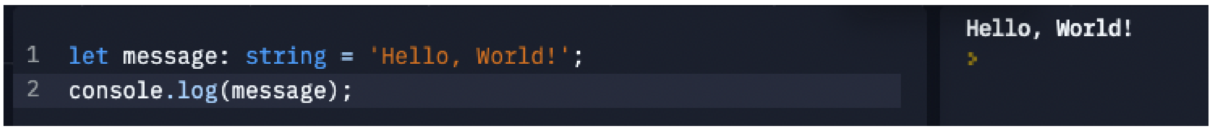
Basic TypeScript Example

Here's how we can create a basic TypeScript example.

1. Create a folder to store the code, named "helloworld."

2. Now, to write code, open the folder in the VS Code editor. After that, create a TypeScript file named "app.ts." The file app.ts will have the following code.

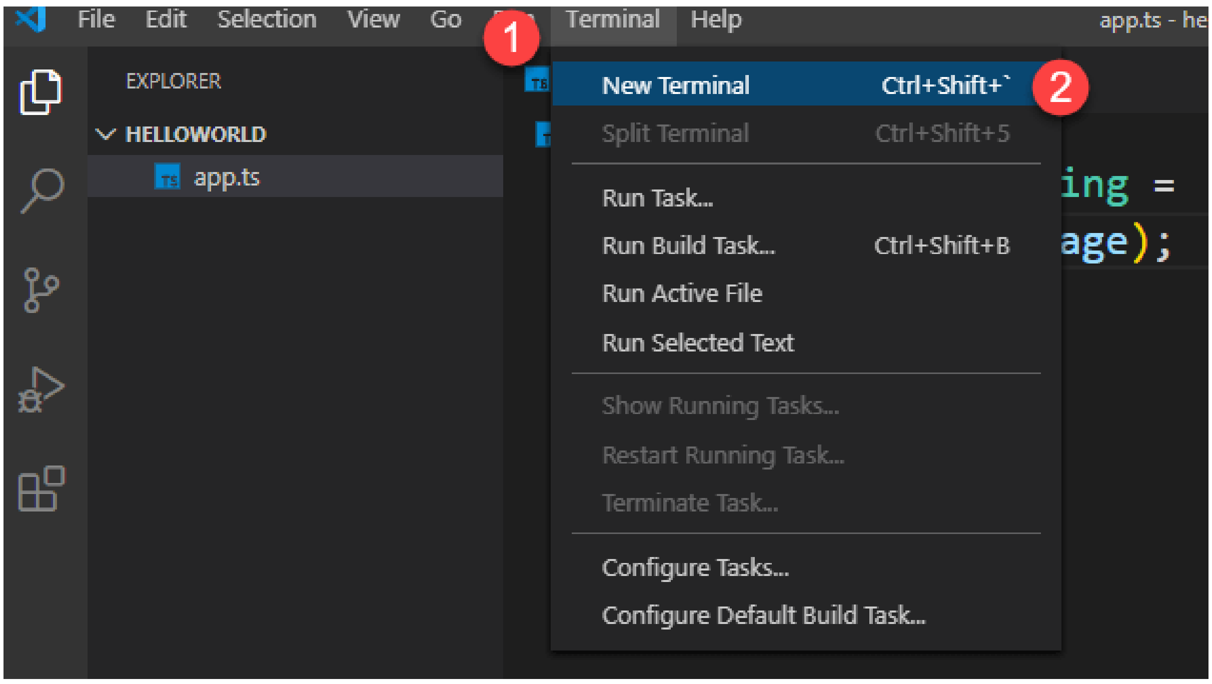
```
let message: string = 'Hello, User!';
console.log(message);
```



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

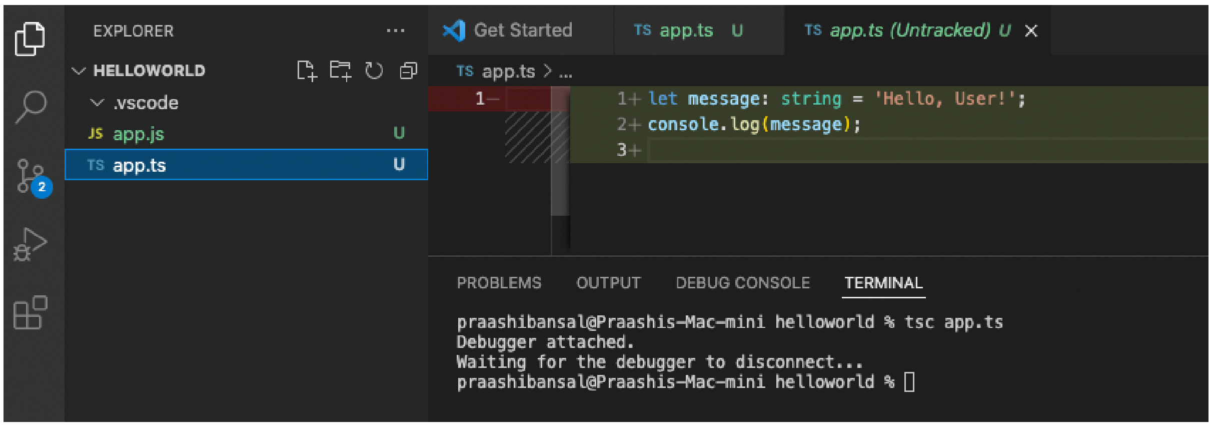
```
Hello, World!
>
```

3. Launch the terminal from the VS Code using the keyboard shortcut "Ctrl+"` or go to the terminal tab on the VS Code and select "new terminal:"



4, Now, compile the “app.ts” file using the following command on the terminal:

```
tsc app.ts
```

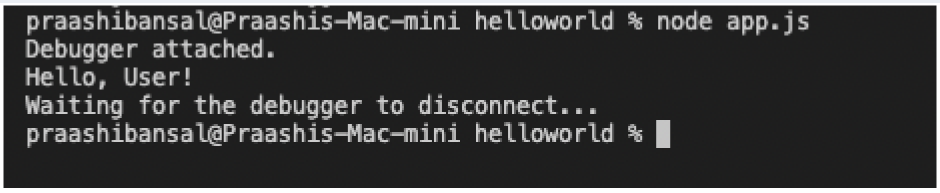


```
let message: string = 'Hello, User!';
console.log(message);
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
Debugger attached.
Waiting for the debugger to disconnect...
praashibansal@Praashis-Mac-mini helloworld % 
```

5. The “app.js” file has been created and is listed below the helloworld folder. Execute the app.js file by using the below command:

```
node app.js
```



```
praashibansal@Praashis-Mac-mini helloworld % node app.js
Debugger attached.
Hello, User!
Waiting for the debugger to disconnect...
praashibansal@Praashis-Mac-mini helloworld % 
```

Here's your basic TypeScript example. Now, let’s look at how to run a TypeScript program in a web browser!

Running TypeScript program in a web browser.

1. Create the index.html file and include the app.js file:

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

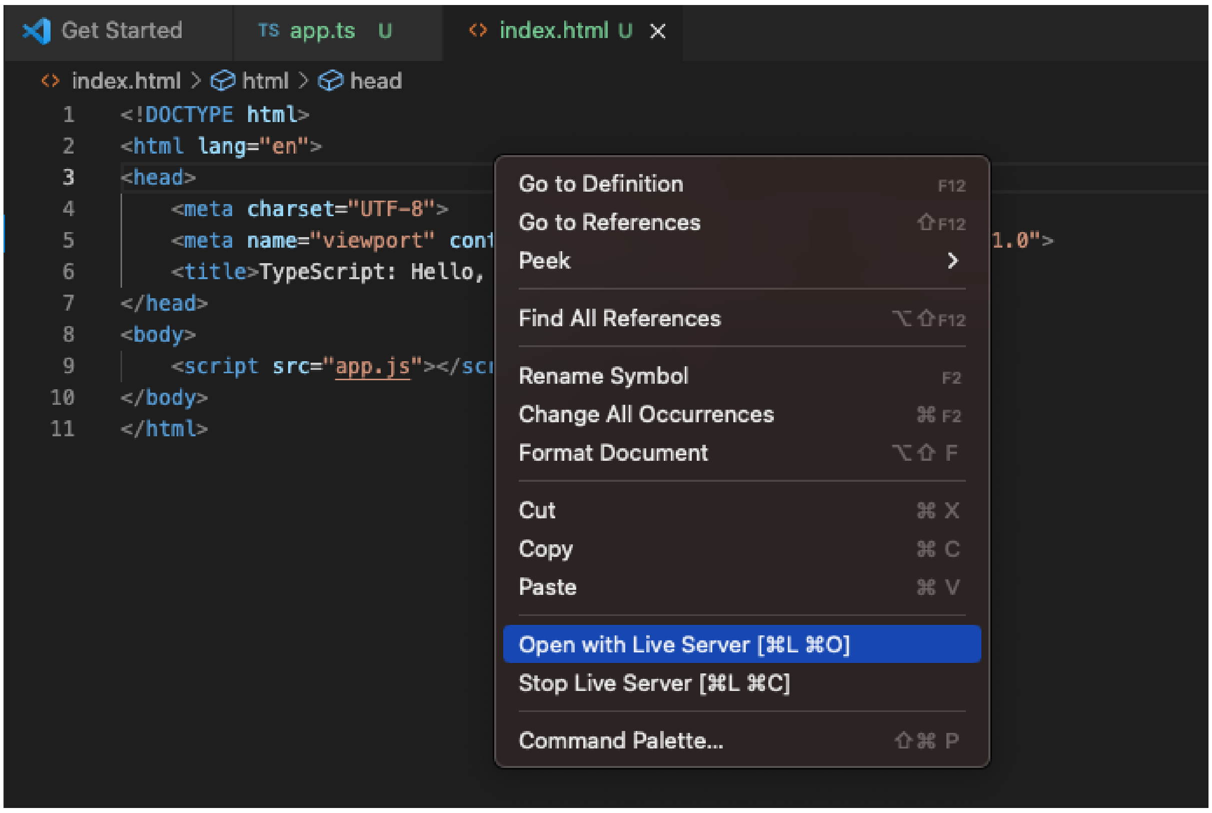
2. Change the app.ts code:

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

3. Compile the app.ts file, using the following code:

```
tsc app.ts
```

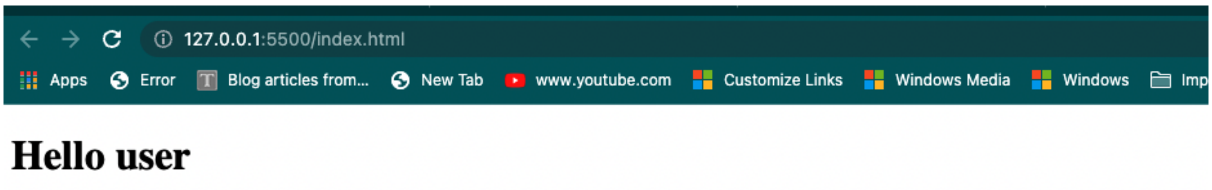
4. Right-click the index.html code and open with Live server:


Get Started | TS app.ts U | index.html U ×

index.html > html > head

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" conti                                  1.0">
    <title>TypeScript: Hello,
</head>
<body>
    <script src="app.js"></sci
</body>
</html>
```

5. The following will be the output.



Hello user

TypeScript Cheat Sheet

1. Basic Types

Type helps you refer to the different properties and functions of a value. The value refers to anything you can assign to a variable, such as a number, string, array, object, and function. For example, “Sam,” specifies the value is a string type and thus will have the properties of a string.

Suppose, the “Sam” value has a property called *length* that will return the number of characters present in the value.

```
console.log("Sam".length); // 5
```

```
Code language: JavaScript (javascript)
```

In addition to characters, it can also have many methods like match(), indexOf(), and toLocaleUpperCase().

For example:

```
console.log('Sam'.toLocaleUpperCase()); // SAM
Code language: JavaScript (javascript)
```

So, a type is a label describing a value's different properties and methods, and each value has a type. There are two different types of TypeScript type:

- **Primitive types:** string, number, boolean, null, undefined, symbol.
- **Object types:** functions, arrays, classes, etc.

Type Annotations in TypeScript

Annotations in TypeScript specify the types for various identifiers explicitly. These identifiers include variables, functions, objects, etc.

Syntax:

```
:Type
```

Use it after the identifier name.

Once you annotate an identifier with a type, you can use it only with that type. If you use that identifier as a different type, the TypeScript compiler will display an error.

Variables and Constants

You can use the following syntax to specify the annotation type for variables and constants.

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

The following example specifies the number annotation for a variable:

```
let counter: number;
Code language: JavaScript (javascript)
```

Keep in mind you can only assign a number to the counter variable:

```
counter = 1;
```

If you try to assign a string to the counter variable, you might encounter the following error:

```
let counter: number;
counter = 'Hello'; // compile error
```

You can use the following syntax for both types to annotate and initialize the variable:

```
let counter: number = 1;
```

Arrays

You can annotate the array type like the variables and constants and add square brackets as a suffix.

```
:type[]

let arrayName: type[];
Code language: JavaScript (javascript)
```

This next example helps you declare an array of strings:

```
let names: string[] = ['Jolly', 'Jam', 'Pam', 'Dam', 'Sam'];
```

Objects

Use annotation to define the type of an object.

```
let person: {
   name: string;
   age: number
};
```

```
person = {
   name: 'John',
   age: 25
}; // valid
```

Here, we have creates an object ‘person’ with two properties: name (String) and age (Number)

Function Arguments & Return Types

Here we shall discuss annotations with function arguments and return types:

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

For the return type, assign a function that only accepts and return to the specified variable.

```
greeting = function (name: string) {
    return `Hi ${name}`;
};
```

In the following code, the function we assigned to ‘greeting’ does not match the function type. Hence, it results in an error.

```
greeting = function () {
    console.log('Hello');
};
```

TypeScript Number

To declare a variable with a floating-point value, you can use the following syntax:

```
let price: number;
```

You can also initialize the variable to a number:

```
let price = 9.95;
```

Like any other language, TypeScript also supports the number literals for decimal, hexadecimal, binary, and octal literals.

Decimal Numbers

```
let counter: number = 0;
let x: number = 150,
    y: number = 240;
```

Binary Numbers

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

Use 0 or 1 after 0b or 0B must be 0 or 1.

Hexadecimal Numbers

They begin with a zero, followed by a lowercase or uppercase letter X. (0x or 0X). The digits following the 0x should fall in the range (0123456789ABCDEF).

For example:

```
let hexadecimal: number = 0XA;
Code language: JavaScript (javascript)
```

Big Integers

With big integers, you can represent whole numbers larger than 253. You must specify the “n” character at the end of the Big integer literal.

For example:

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

TypeScript String

We can leverage the double quotes (") or single quotes (') to represent string literals.

```
let firstName: string = 'Linda';
let title: string = "Web Developer";
Code language: JavaScript (javascript)
```

To represent characters, TypeScript employs the backtick ('). Leveraging the template strings, we can create multi-line strings.

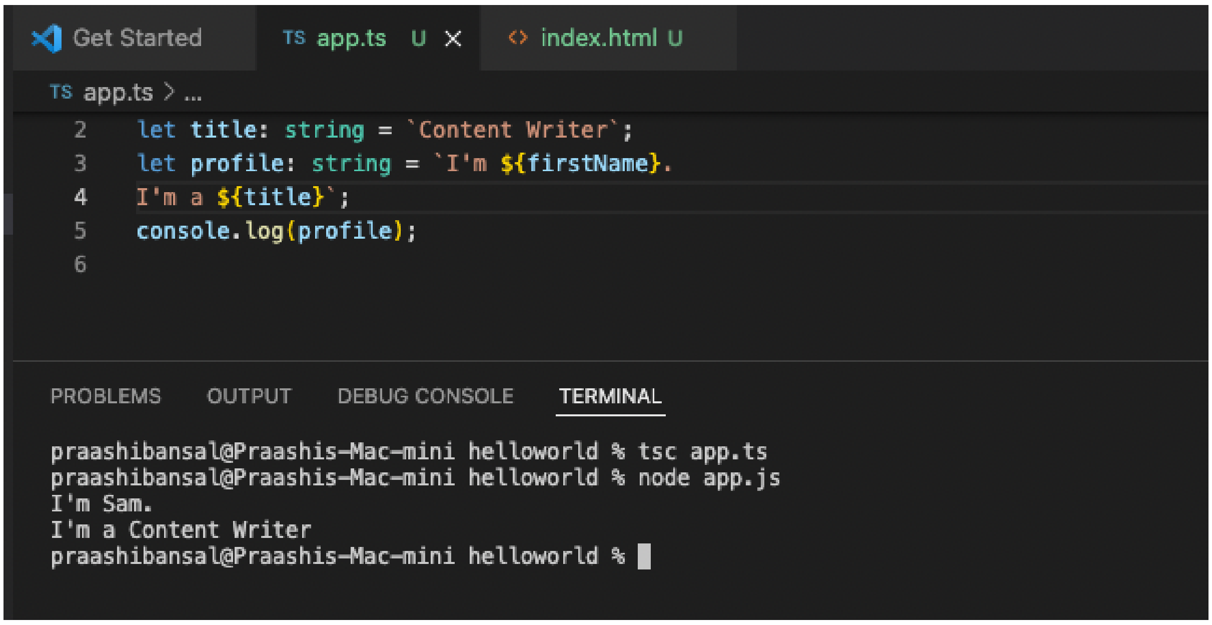
The following example shows how to use the backtick (') to create a multi-line string.

```
let description = `Welcome to Hackr.io
Get tutorials you need
and master`;
Code language: JavaScript (javascript)
```

To incorporate variables into Strings, you need to leverage String.

For example-

```
let firstName: string = `Sam`;
let title: string = `Content Writer`;
let profile: string = `I'm ${firstName}.
I'm a ${title}`;
console.log(profile);
```



```
let title: string = `Content Writer`;
let profile: string = `I'm ${firstName}.
I'm a ${title}`;
console.log(profile);
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
I'm Sam.
I'm a Content Writer
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Boolean

The TypeScript boolean, a primitive type, allows you to use two values: true and false.

For example:

```
let pending: boolean;
pending = true;
// after a while
// ..
pending = false;
Code language: JavaScript (javascript)
```

The Boolean type has the letter B in uppercase, making it different from the boolean type. Our recommendation? Avoid using the Boolean type.

For example:

```
let pending: boolean;
pending = true;
// after a while
// ..
pending = false;
Code language: JavaScript (javascript)
```

JavaScript comes with a Boolean type which we call non-primitive boxed objects. The Boolean type is different from the boolean type, as the former one has the capital B.

TypeScript Object Type

The TypeScript object type represents all primitive type values.

Primitive types in TypeScript:

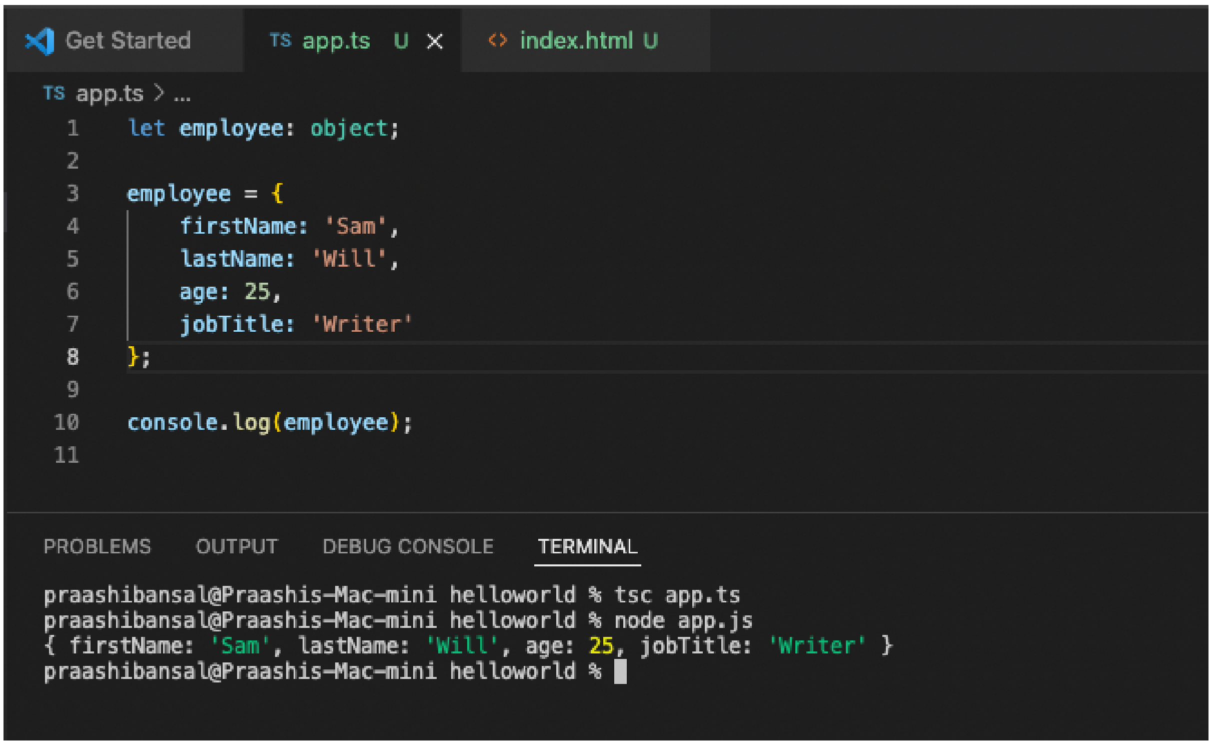
- number
- bigint
- string
- boolean
- null
- undefined
- symbol

The following example specifies how to declare a variable that holds an object.

For example:

```
let employee: object;
```

```
employee = {
   firstName: 'Sam',
   lastName: 'Will',
   age: 25,
   jobTitle: 'Writer'
};
console.log(employee);
```

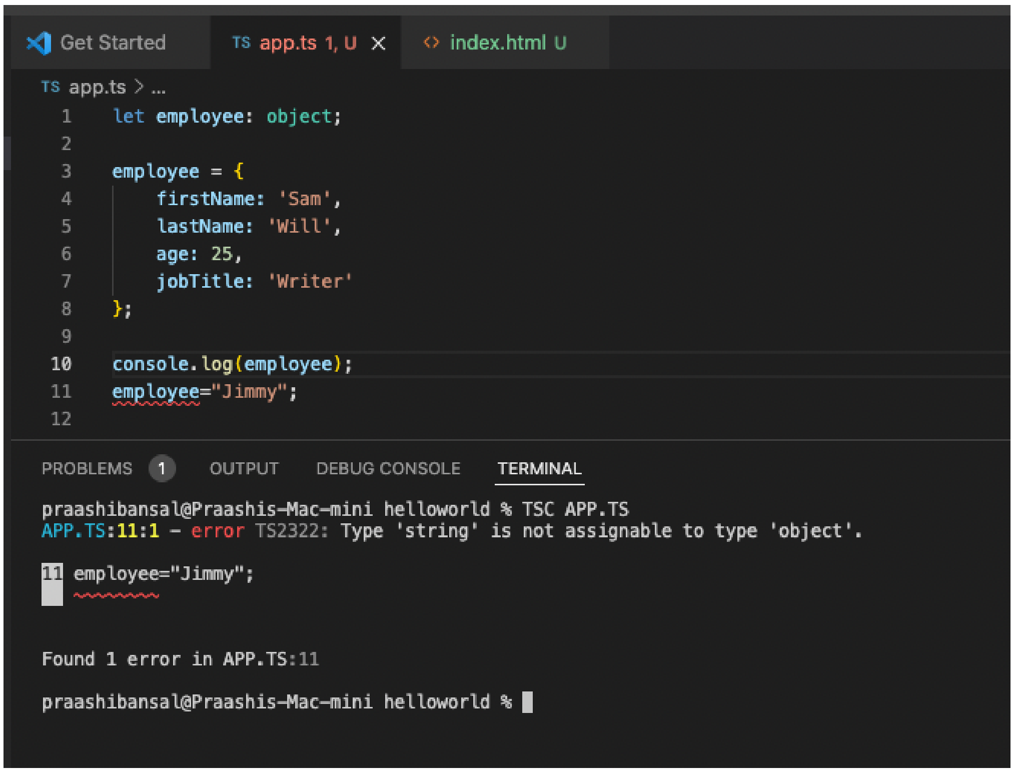


```
let employee: object;

employee = {
    firstName: 'Sam',
    lastName: 'Will',
    age: 25,
    jobTitle: 'Writer'
};

console.log(employee);
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
{ firstName: 'Sam', lastName: 'Will', age: 25, jobTitle: 'Writer' }
praashibansal@Praashis-Mac-mini helloworld %
```

If you want to reassign a primitive value to the employee object, you will receive an error:

```
employee = "Jimmy";
Code language: JavaScript (javascript)
```

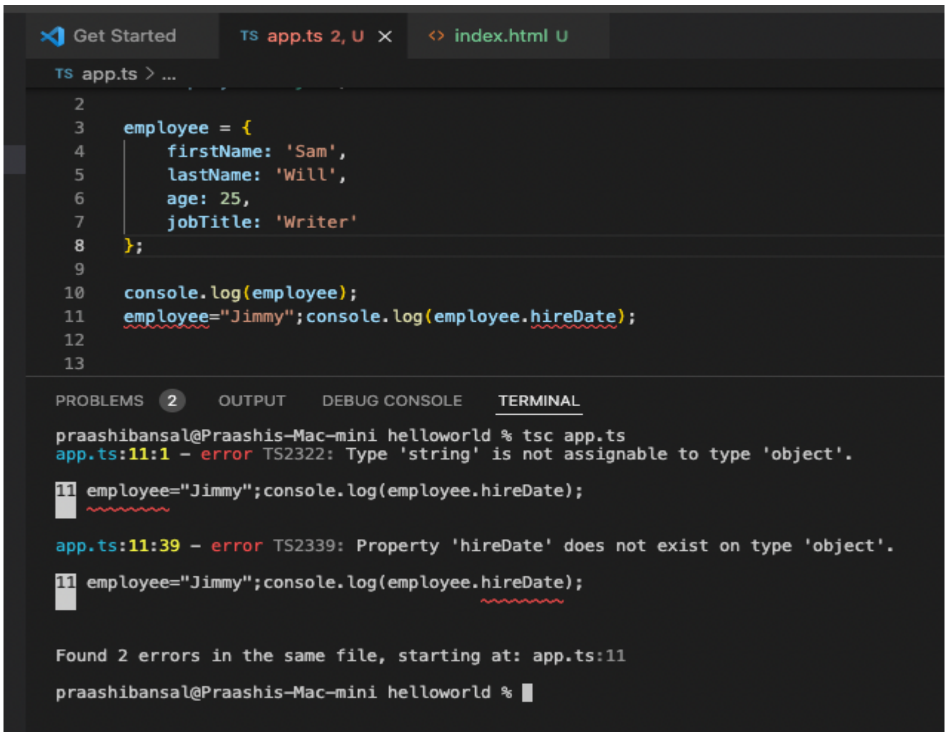


```
let employee: object;

employee = {
    firstName: 'Sam',
    lastName: 'Will',
    age: 25,
    jobTitle: 'Writer'
};

console.log(employee);
employee="Jimmy";
```

```
praashibansal@Praashis-Mac-mini helloworld % TSC APP.TS
APP.TS:11:1 - error TS2322: Type 'string' is not assignable to type 'object'.

11 employee="Jimmy";
   ~~~~~~~~


Found 1 error in APP.TS:11

praashibansal@Praashis-Mac-mini helloworld %
```

Here ‘employee’ is an object type having a fixed set of properties. You may encounter an error if you access a property out of the scope of “employee.”

```
console.log(employee.hireDate);
Code language: CSS (css)
```



```

employee = {
    firstName: 'Sam',
    lastName: 'Will',
    age: 25,
    jobTitle: 'Writer'
};

console.log(employee);
employee="Jimmy";console.log(employee.hireDate);


```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:11:1 - error TS2322: Type 'string' is not assignable to type 'object'.

11 employee="Jimmy";console.log(employee.hireDate);
   ~~~~~~~~

app.ts:11:39 - error TS2339: Property 'hireDate' does not exist on type 'object'.

11 employee="Jimmy";console.log(employee.hireDate);
                                         ~~~~~~~~


Found 2 errors in the same file, starting at: app.ts:11

praashibansal@Praashis-Mac-mini helloworld %
```

The following syntax lets you define the employee object properties explicitly:

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

Assign the employee object to a literal object:

```
employee = {
    firstName: 'Jimmy',
    lastName: 'Will',
    age: 30,
    jobTitle: 'Writer'
};
```

Or you can use the combined syntax in the same statement:

```
let employee: {
    firstName: string;
    lastName: string;
    age: number;
    jobTitle: string;
} = {
    firstName: 'Jimmy',
    lastName: 'Will',
    age: 30,
    jobTitle: 'Writer'
};
```

Object vs. object

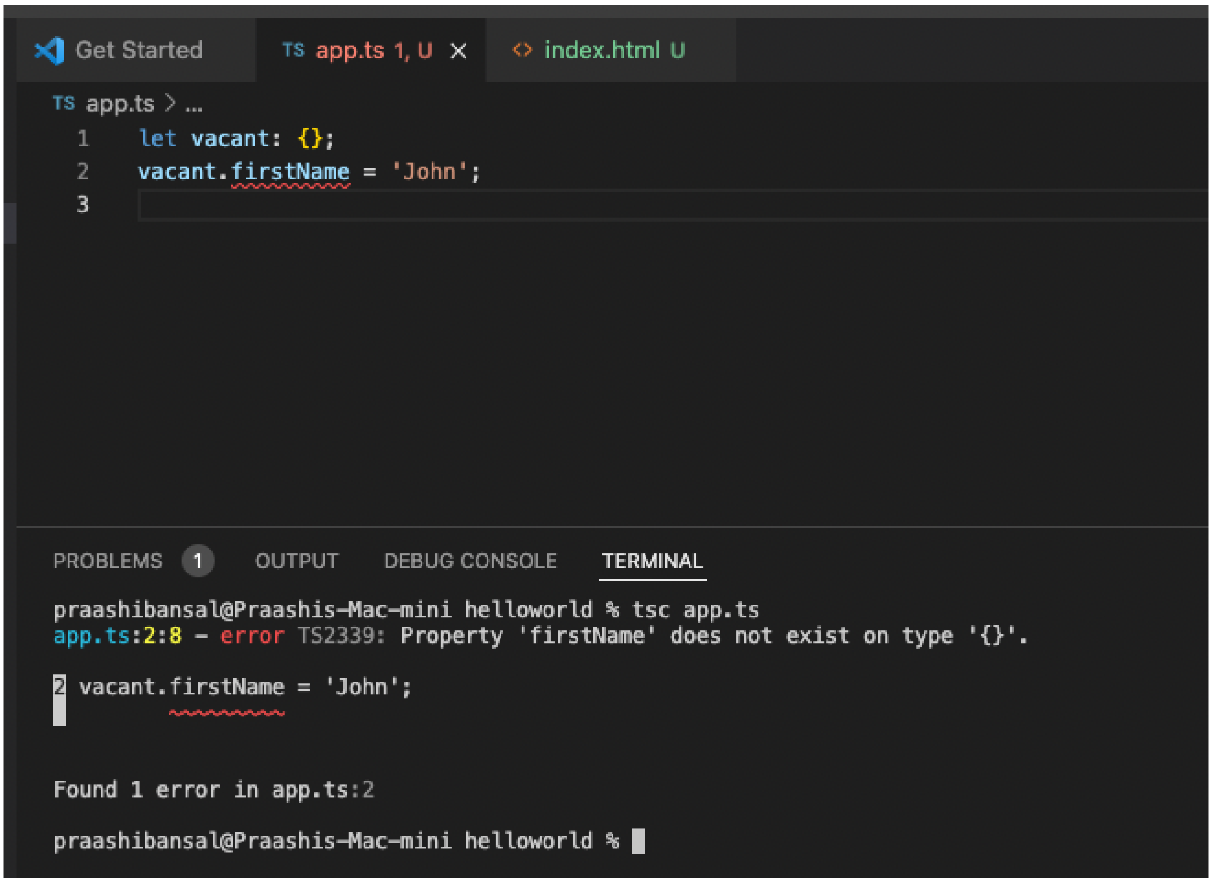
TypeScript supports another type called Object having the letter O in uppercase.

We know that the object type indicates all non-primitive values. Meanwhile, the Object type represents the functionality of all objects.

The Empty Type {}

Similar to the object type, TypeScript has an empty type specified by the empty type {}. It describes an object that has no properties. If you try to access any property using such an object, you will receive the following compile-time error:

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

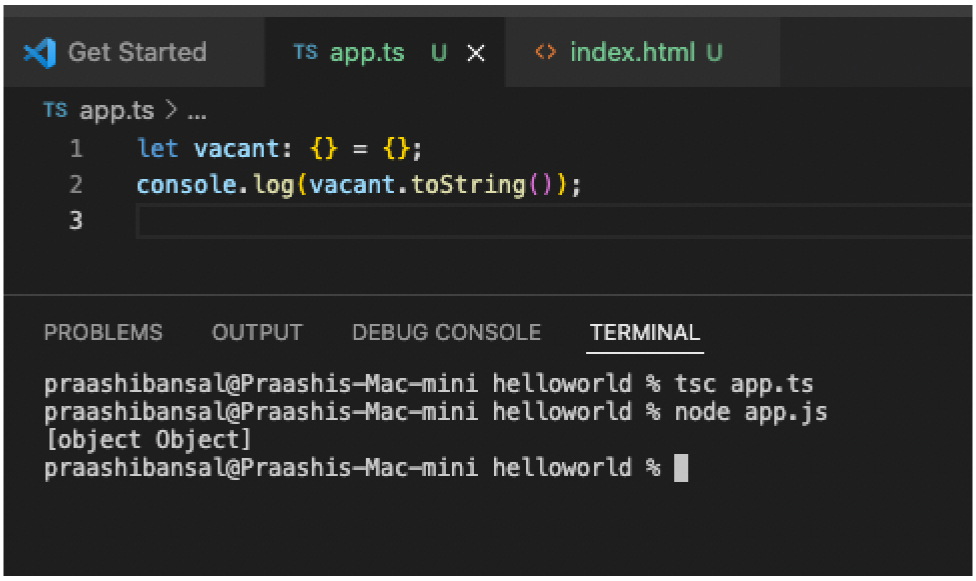


```
let vacant: {};
vacant.firstName = 'John';
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:2:8 - error TS2339: Property 'firstName' does not exist on type '{}'.

2 vacant.firstName = 'John';
         ~~~~~~~~~

Found 1 error in app.ts:2

praashibansal@Praashis-Mac-mini helloworld %
```

But, you can easily access all the properties and methods declared on the Object type. For example:

```
let vacant: {} = {};
console.log(vacant.toString());
```



```
let vacant: {} = {};
console.log(vacant.toString());
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
[object Object]
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Array Type

A TypeScript array is referred to as an ordered list of data.

The following syntax declares an array with values of a specific type:

```
let arrayName: type[];
```

For example, you can use this syntax to declare an array of strings:

```
let skills: string[];
```

Also, you can add more than one string value to the above array:

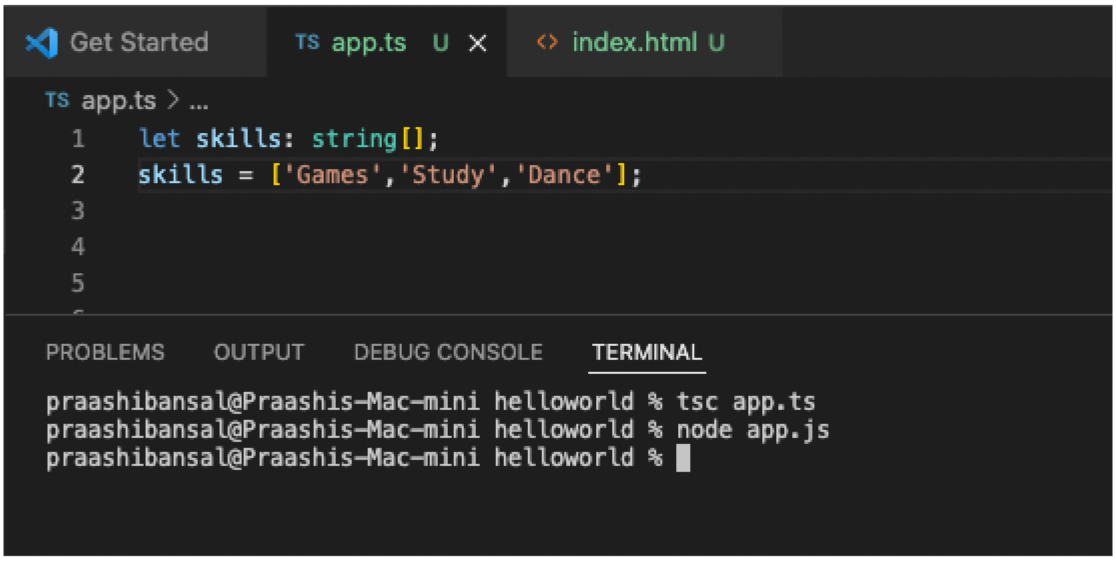
```
skills[0] = "Game";
skills[1] = "Study";
```

Or you can use the push() method to enter new values to the existing array:

```
skills.push('Science');
```

Define an array within one line:

```
let skills = ['Games','Study','Dance'];
```

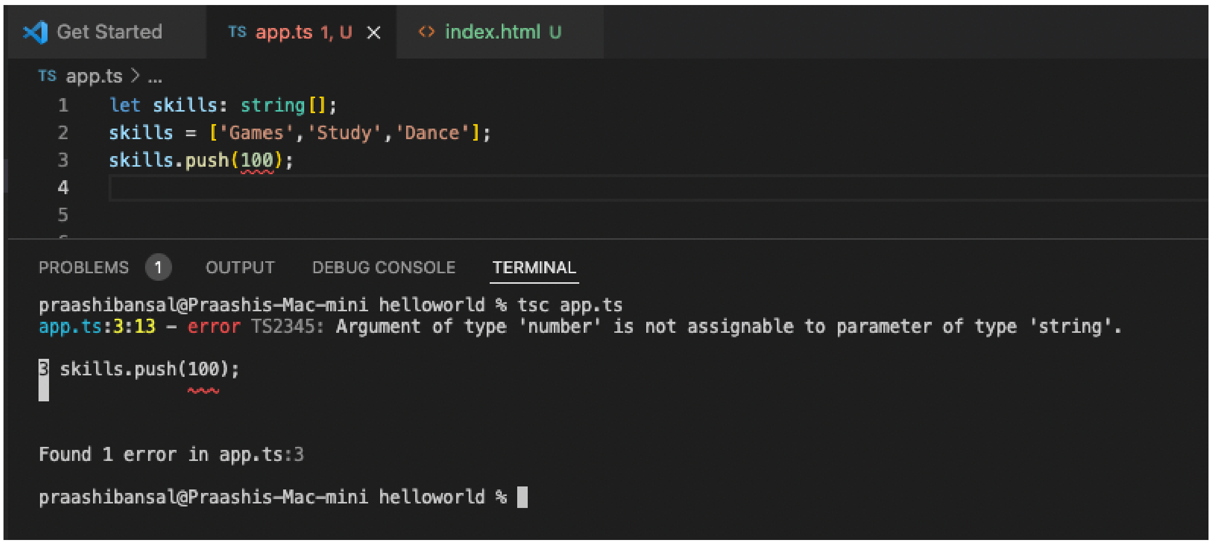


```
let skills: string[];
skills = ['Games','Study','Dance'];
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld %
```

Once you define an array of a specific type, you cannot add any incompatible values to the array, or you'll receive an error message:

```
skills.push(100);
```



```
let skills: string[];
skills = ['Games','Study','Dance'];
skills.push(100);
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:3:13 - error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.

3 skills.push(100);
              ~~~

Found 1 error in app.ts:3

praashibansal@Praashis-Mac-mini helloworld %
```

Now, let us extract the first element of our array:

```
let skill = skills[0];
console.log(typeof(skill));
```

Type inference takes place.

We have extracted the first element of the ‘skill’ array. Later, we assigned it to the variable ‘skill’.

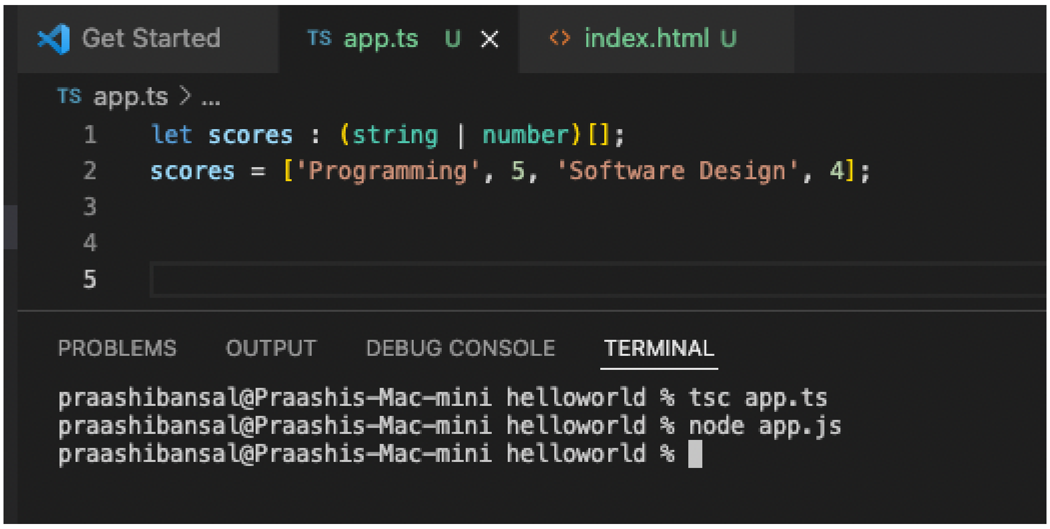
Storing Values of Mixed Types

Let us now discuss how to add values of different types to an array.

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

The above array is array of mixed types, i.e., string | number:

```
let scores : (string | number)[];
scores = ['Programming', 5, 'Software Design', 4];
```



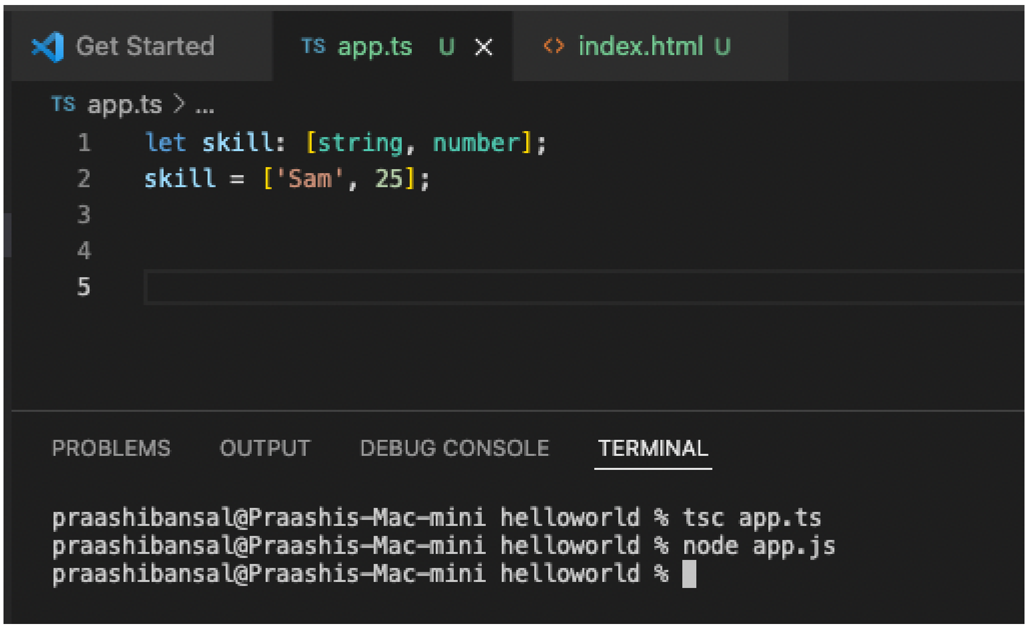
```
let scores : (string | number)[];
scores = ['Programming', 5, 'Software Design', 4];
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Tuple

A tuple works like an array but with some additional considerations. A tuple has a fixed number of elements, none of which have to be the same. For example, you can use a tuple to specify a value as a pair of a string and a number:

```
let skill: [string, number];
skill = ['Sam', 25];
```

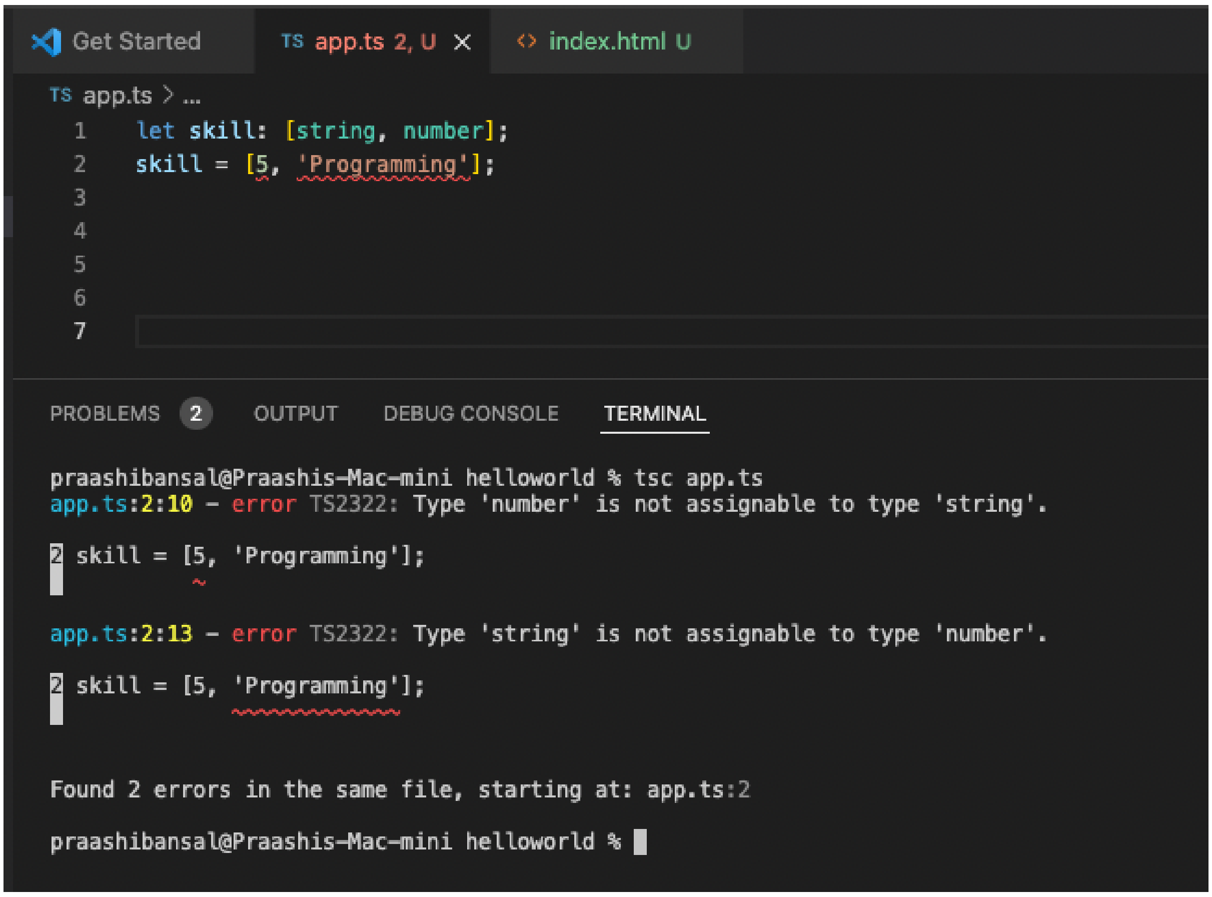


```
let skill: [string, number];
skill = ['Sam', 25];
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld % 
```

Always take care of the order in which you have your values stored. If you mistakenly change the order, you will get an error:

```
let skill: [string, number];
skill = [5, 'Programming'];
```



```
let skill: [string, number];
skill = [5, 'Programming'];
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:2:10 - error TS2322: Type 'number' is not assignable to type 'string'.

2 skill = [5, 'Programming'];
           ~

app.ts:2:13 - error TS2322: Type 'string' is not assignable to type 'number'.

2 skill = [5, 'Programming'];
              ~~~~~~~~~~~~~


Found 2 errors in the same file, starting at: app.ts:2

praashibansal@Praashis-Mac-mini helloworld % 
```

TypeScript Enum

An enum (enumerated type) refers to a group of named constant values. The enum keyword is used to define enum as a prefix to the enum name. Then, define constant values for the enum.

Here's the syntax:

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

Here, the constant1, constant2, etc., are the enum members.

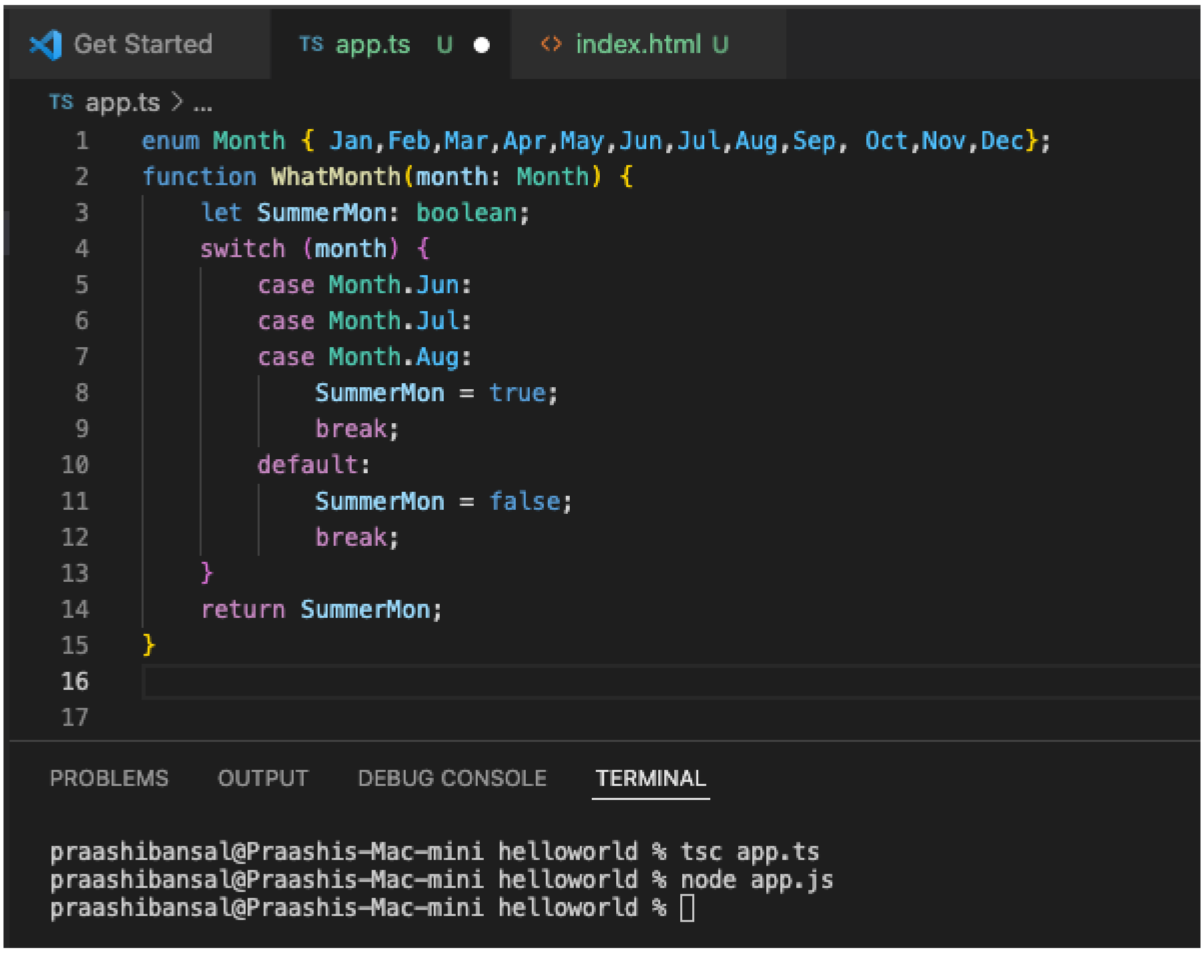
Example:

```
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.

Now, we will declare a function using the Month enum as the parameter.

```
function WhatMonth(month: Month) {
    let SummerMon: boolean;
    switch (month) {
        case Month.Jun:
        case Month.Jul:
        case Month.Aug:
            SummerMon = true;
            break;
        default:
            SummerMon = false;
            break;
    }
    return SummerMon;
}

console.log(SummerMon(Month.Jun));
```



```
enum Month { Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep, Oct,Nov,Dec};
function WhatMonth(month: Month) {
    let SummerMon: boolean;
    switch (month) {
        case Month.Jun:
        case Month.Jul:
        case Month.Aug:
            SummerMon = true;
            break;
        default:
            SummerMon = false;
            break;
    }
    return SummerMon;
}
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld % 
```

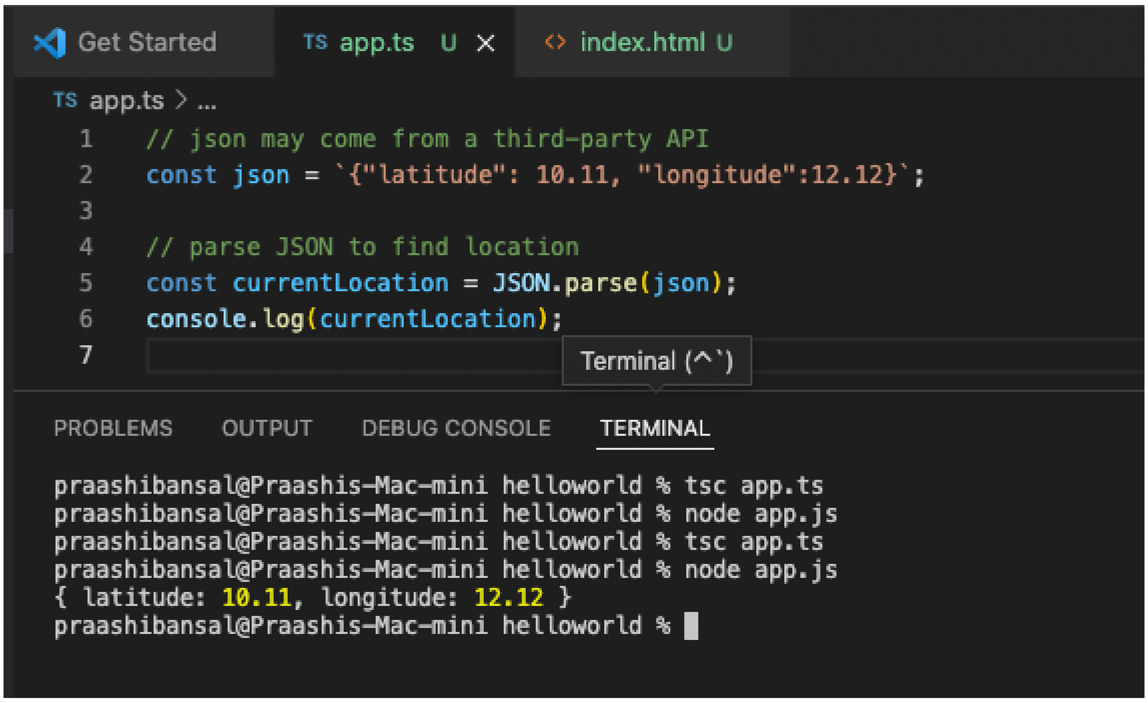
TypeScript Any Type

In some scenarios, you may need to store a value in a variable without knowing its type. A third party API or user input may provide that value. In that case, type checking may be required, and the value may pass through the compile-time check.

To do so, we use the “any” type.

The “any” type lets you assign a value to a variable with any type.

```
// 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);
```

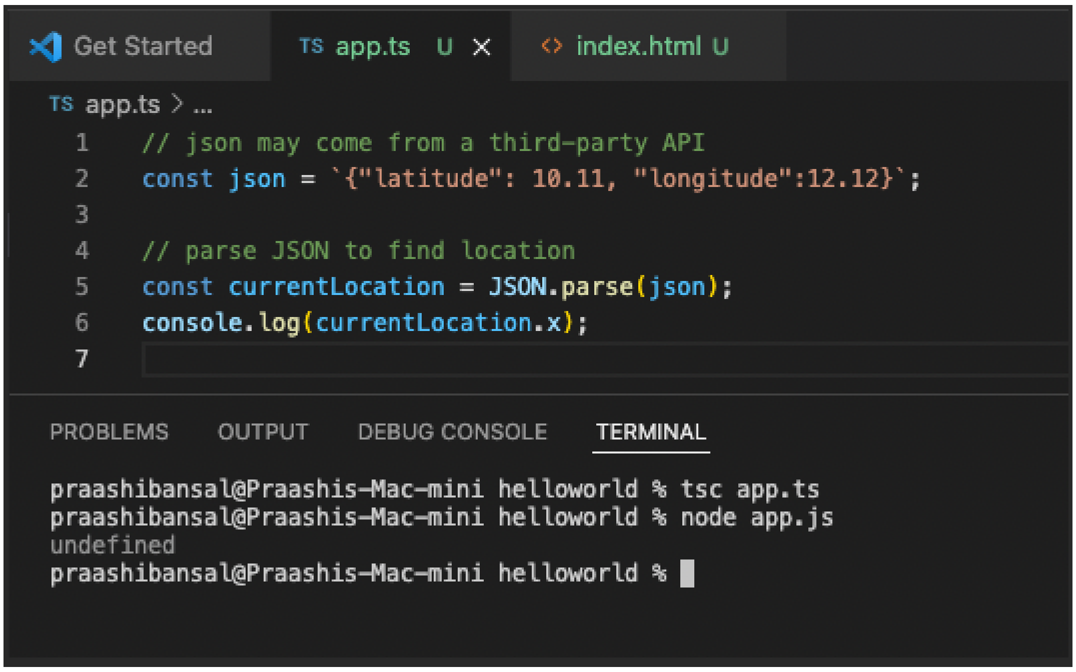


```
// json may come from a third-party API
const json = `{"latitude": 10.11, "longitude":12.12}`;

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

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
{ latitude: 10.11, longitude: 12.12 }
praashibansal@Praashis-Mac-mini helloworld %
```

In this example, the JSON.parse() function returns an object to which we assign the currentLocation variable. However, when you use the currentLocation to access object properties, TypeScript will not perform any check:

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



```
// 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.x);
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
undefined
praashibansal@Praashis-Mac-mini helloworld %
```

There will be no issue while compiling.

If you declare a variable without mentioning a type, TypeScript assumes it as any type, known as type inference. It means that TypeScript will guess the variable type.

TypeScript Void Type

The void type specifies that there is no specific type. It is somehow opposite to "any" type.

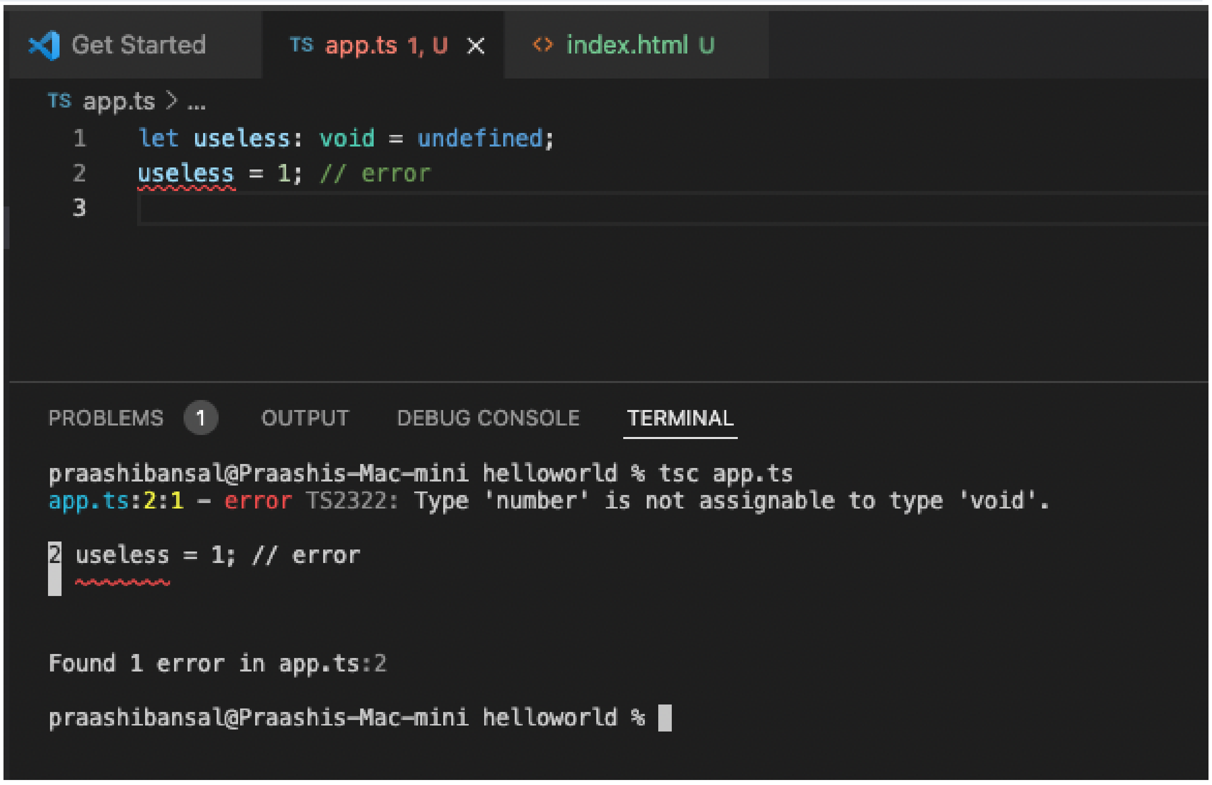
For example:

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

When any function does not require you to return a value, use the void type. It will improve the code's clarity and ensure type-safeness. Also, there is no need to navigate the entire function body to check if it returns anything.

Suppose you use the void type for a variable and assign "undefined." Such a void type is of no use. For example:

```
let useless: void = undefined;
useless = 1; // error
```

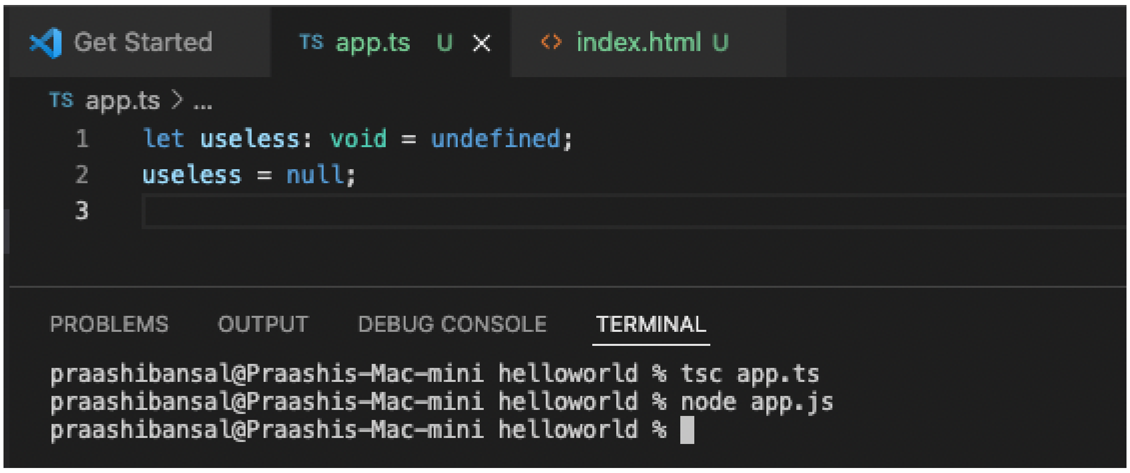


```
let useless: void = undefined;
useless = 1; // error
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:2:1 - error TS2322: Type 'number' is not assignable to type 'void'.

2 useless = 1; // error
  ~~~~~~~


Found 1 error in app.ts:2

praashibansal@Praashis-Mac-mini helloworld %
```

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

useless = null;



```
let useless: void = undefined;
useless = null;
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Never Type

The never type means containing no values, thus you cannot assign any value to a variable with a never type. When the return type of a function throws errors, use the never type.

For example:

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

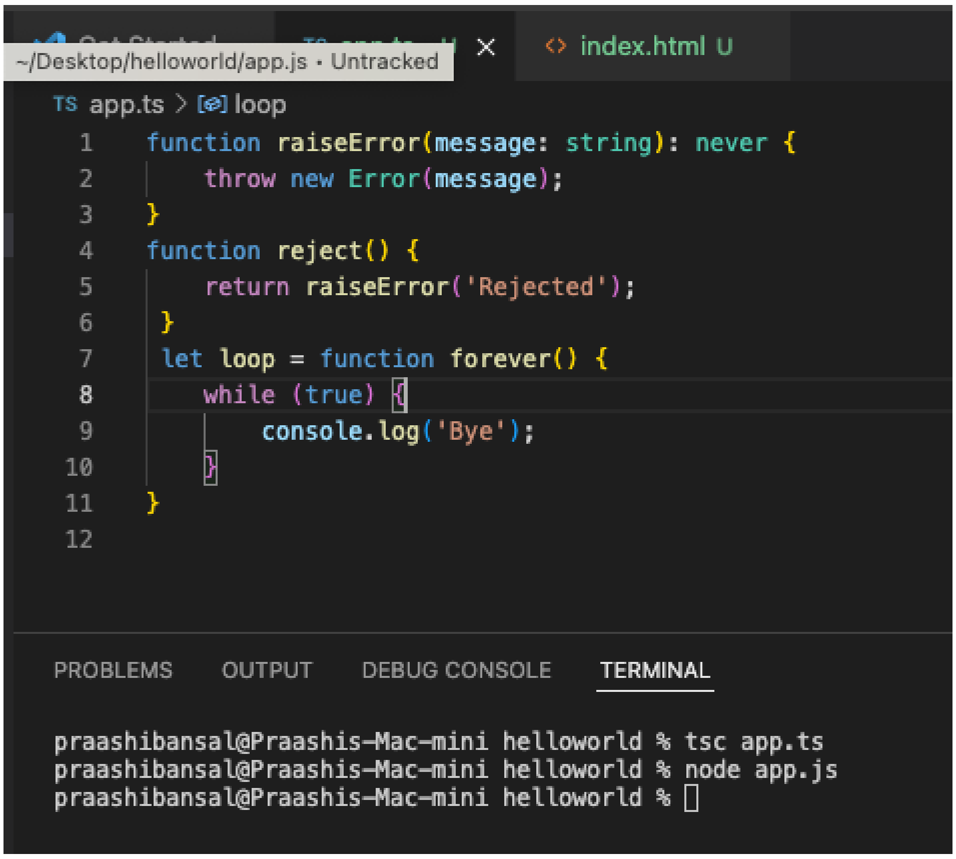
In this next example, the return type of the never type.

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

A function with an expression containing an indefinite loop has the never type.

For example:

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



```
function raiseError(message: string): never {
    throw new Error(message);
}
function reject() {
    return raiseError('Rejected');
 }
 let loop = function forever() {
    while (true) {
        console.log('Bye');
    }
}

```

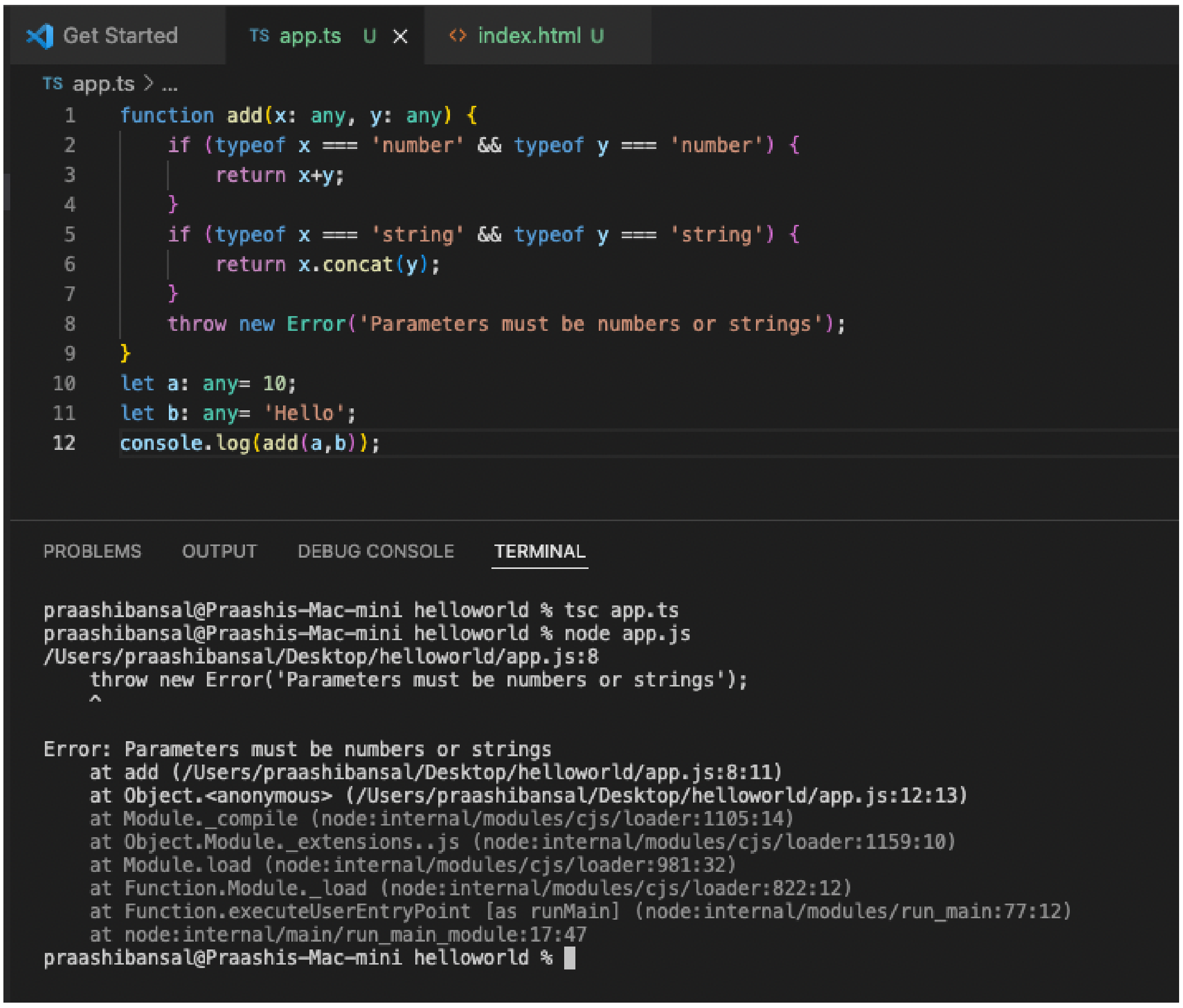
```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld % 
```

TypeScript Union Type

Many times, you may encounter a function that may either require a number or string type parameter.

For example:

```
function add(x: any, y: any) {
    if (typeof x === 'number' && typeof y === 'number') {
        return x+y;
    }
    if (typeof x === 'string' && typeof y === 'string') {
        return x.concat(y);
    }
    throw new Error('Parameters must be numbers or strings');
}
```



```
function add(x: any, y: any) {
    if (typeof x === 'number' && typeof y === 'number') {
        return x+y;
    }
    if (typeof x === 'string' && typeof y === 'string') {
        return x.concat(y);
    }
    throw new Error('Parameters must be numbers or strings');
}
let a: any= 10;
let b: any= 'Hello';
console.log(add(a,b));
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
/Users/praashibansal/Desktop/helloworld/app.js:8
    throw new Error('Parameters must be numbers or strings');
    ^

Error: Parameters must be numbers or strings
    at add (/Users/praashibansal/Desktop/helloworld/app.js:8:11)
    at Object.<anonymous> (/Users/praashibansal/Desktop/helloworld/app.js:12:13)
    at Module._compile (node:internal/modules/cjs/loader:1105:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47
praashibansal@Praashis-Mac-mini helloworld % 
```

We have not specified the type of parameters in the function ‘add()’. Therefore, if the parameters are numbers, add() performs addition. If they are strings, add() concatenate those strings.

But, if the parameters are not numbers as well as string, add() will throw an error.

```
add(true, false);
```

This is where the TypeScript union type comes into the picture. With Union type, you can merge various types into one.

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

```
let result: number | string;
result = 20; // OK
result = 'Bye'; // also OK
result = true; // a boolean value, not OK
```

A union type describes a value that can be one of several types.

TypeScript Type Aliases

Type aliases let you create a new name for an existing type. However, the existing type should be valid.

```
type alias = existingType;
```

Let’s examine an example of type alias below.

Here, we assign the type String to chars.

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

TypeScript String Literal Types

With the help of string literal types, you can define a type that accepts only one specified string literal.

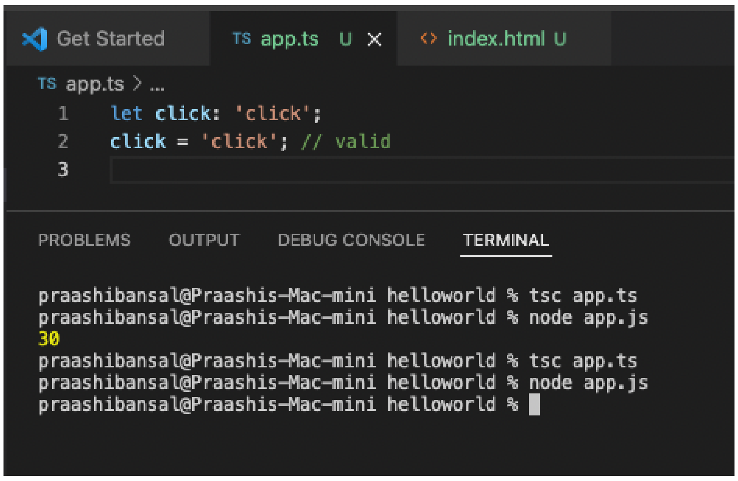
In the below code, we have defined a string literal type that accepts a literal string “click:”

```
let click: 'click';
```

Here, the click is a string literal type that accepts only the string literal “click.”

If you assign ‘click’ to the click, it will be valid:

```
click = 'click'; // valid
```

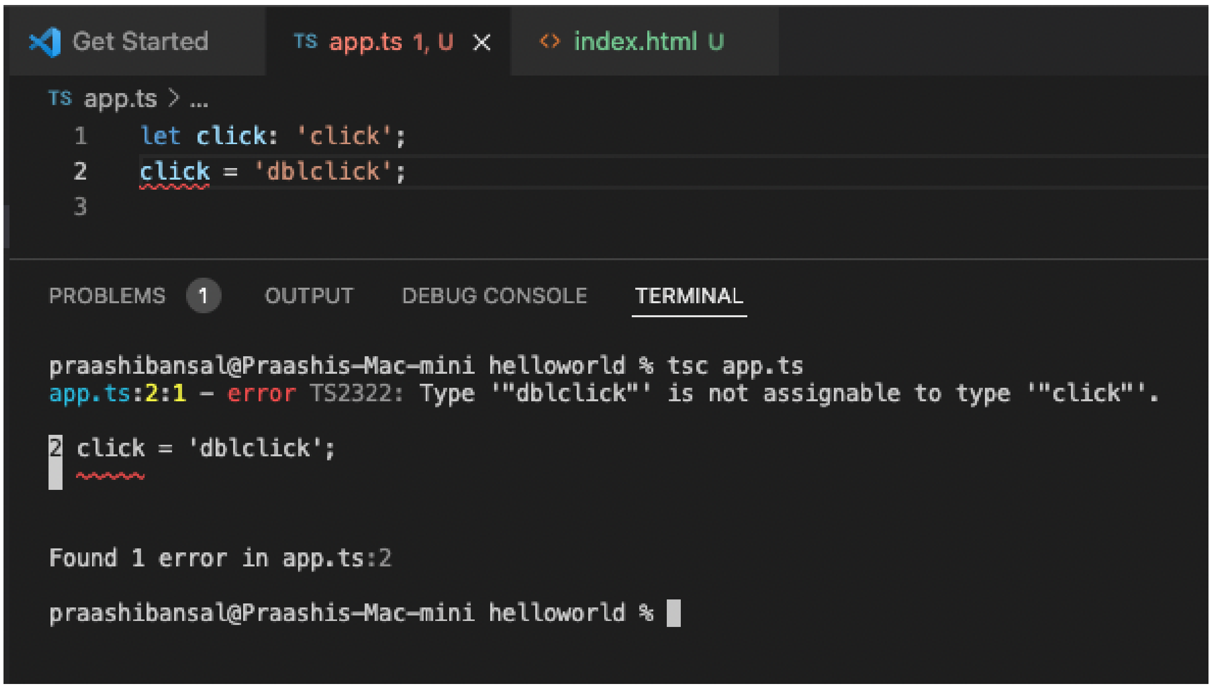


```
let click: 'click';
click = 'click'; // valid
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
30
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld %
```

If you assign another string literal ‘dbclick’ to the click, it gives a compile-time error:

```
click = 'dblclick';
```



```
let click: 'click';
click = 'dblclick';
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:2:1 - error TS2322: Type '"dblclick"' is not assignable to type '"click"'.

2 click = 'dblclick';
  ~~~~~


Found 1 error in app.ts:2

praashibansal@Praashis-Mac-mini helloworld %
```

When you wish to restrict a possible string value in a variable, the string literal type comes in handy.

TypeScript Type Inference

Type inference describes where and how TypeScript infers types without explicitly annotating them. Geneally, we use annotations to explicitly define a type of a variable.

For example:

```
let counter: number;
```

From the above code, TypeScript considers the counter type as a number.

For example:

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

Also, whenever you set the value of a parameter of any function, TypeScript considers its type as the default value type.

For example:

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

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

2. Control Flow Statements

If..else

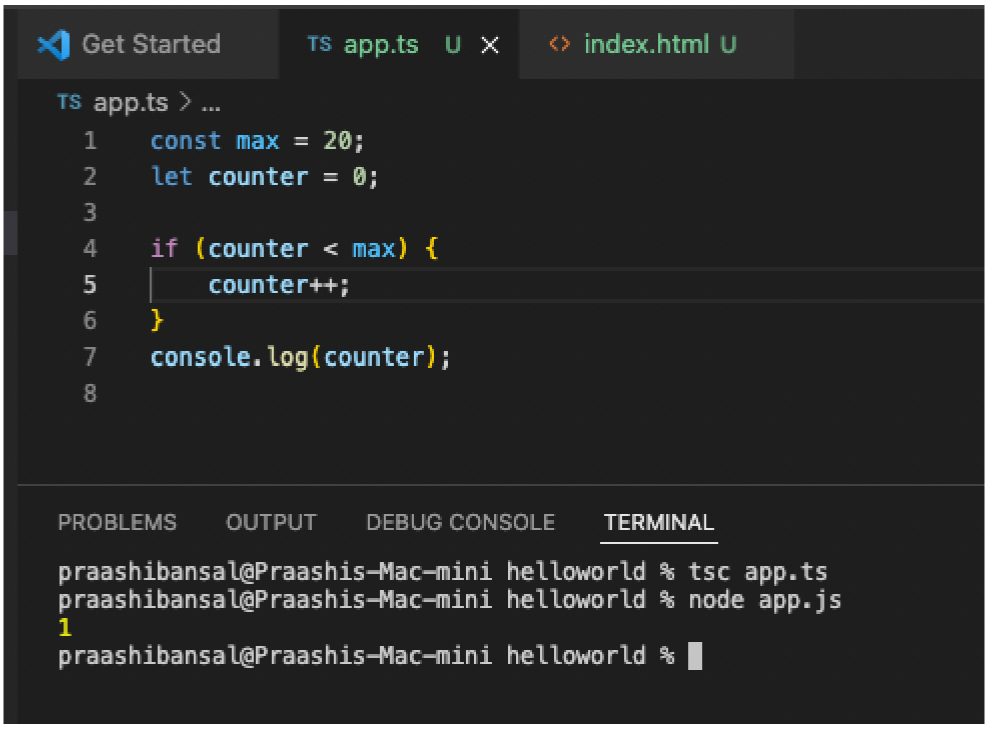
An if statement executes a piece of code based on a condition. In the following syntax, the body inside ‘if’ executes only if the given condition evaluates to true:

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

Example:

```
const max = 20;
let counter = 0;

```

```
if (counter < max) {
    counter++;
}
console.log(counter);
```



Get Started | app.ts U | index.html U

app.ts > ...

```
const max = 20;
let counter = 0;

if (counter < max) {
    counter++;
}
console.log(counter);
```

PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
1
praashibansal@Praashis-Mac-mini helloworld %
```

In the above code, the value of counter increases only if its value is less than max.

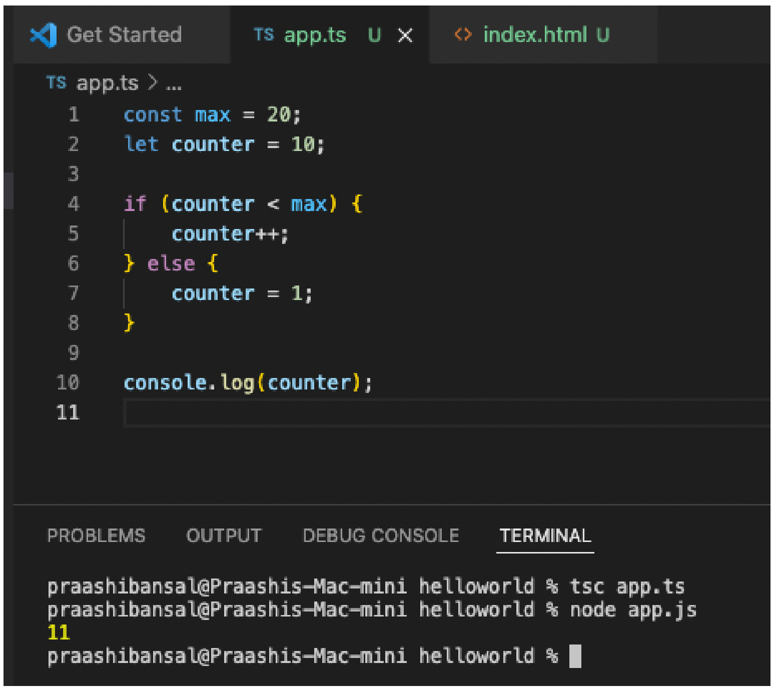
TypeScript if…else statement

If the condition in ‘if’ is false, the control moves to executing the body inside ‘else’.

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

For example:

```
const max = 20;
let counter = 10;

if (counter < max) {
    counter++;
} else {
    counter = 1;
}

console.log(counter);
```



```
const max = 20;
let counter = 10;

if (counter < max) {
    counter++;
} else {
    counter = 1;
}

console.log(counter);
```

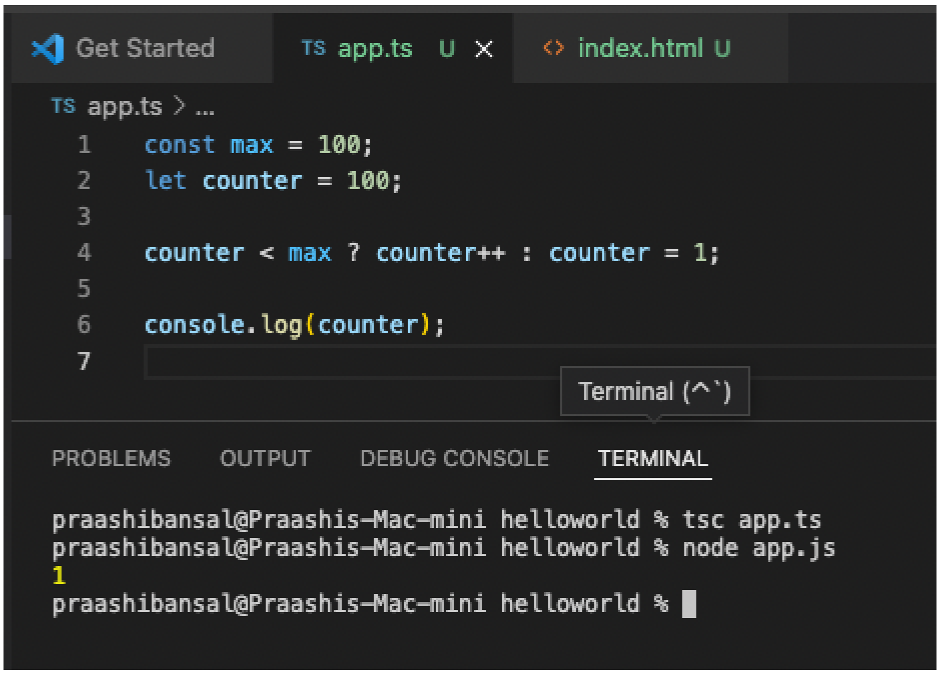
```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
11
praashibansal@Praashis-Mac-mini helloworld %
```

Ternary Operator ?:

The ternary operator (?:) is used to make the code shorter. It is a substitute for if-else In the case of simple conditions.

For example:

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

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

console.log(counter);
```



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

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

console.log(counter);
```

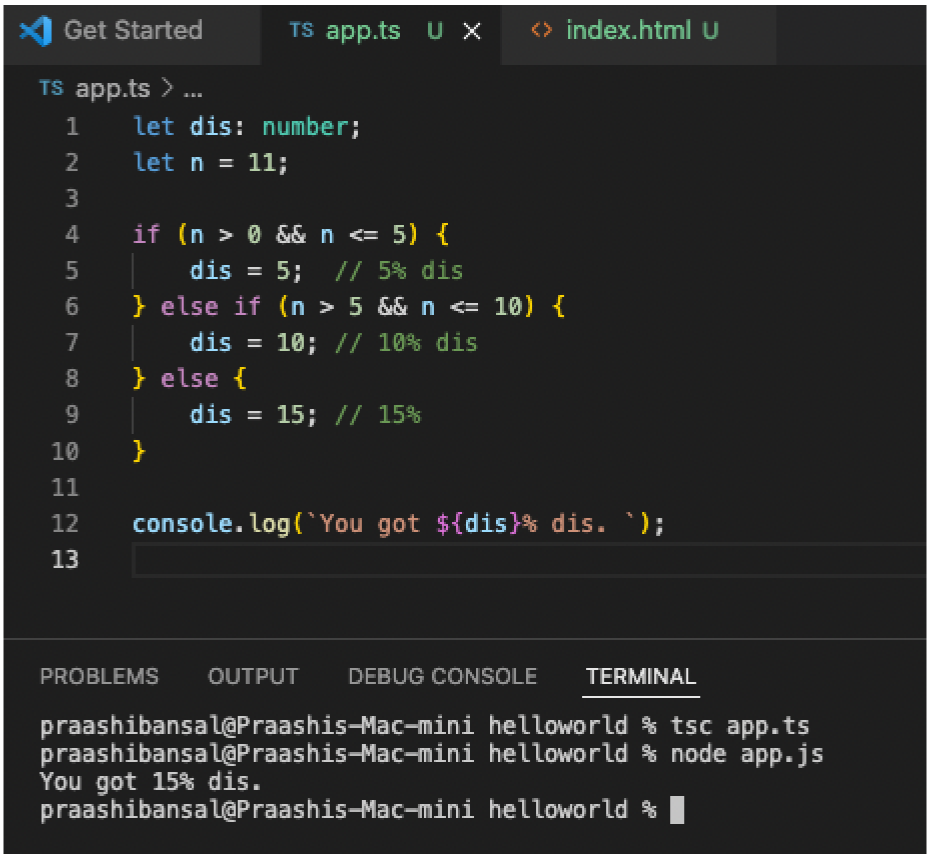
```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
1
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript if…else if…else statement

The if...else if...else statement is ideal when you want to check multiple conditions.

Example:

```typescript
let dis: number;
let n = 11;

if (n > 0 && n <= 5) {
    dis = 5;  // 5% dis
} else if (n > 5 && n <= 10) {
    dis = 10; // 10% dis
} else {
    dis = 15; // 15%
}

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



```typescript
let dis: number;
let n = 11;

if (n > 0 && n <= 5) {
    dis = 5;  // 5% dis
} else if (n > 5 && n <= 10) {
    dis = 10; // 10% dis
} else {
    dis = 15; // 15%
}

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

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
You got 15% dis.
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Switch Case

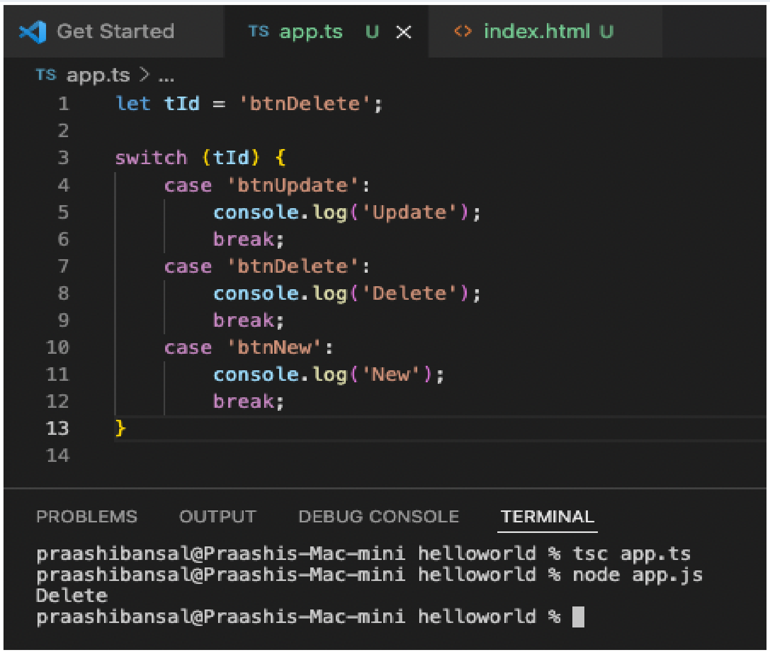
The switch case is useful when you want to check the value of expression against multiple values and execute code on the match found.

Syntax:

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

Example:

```
let tId = 'btnDelete';

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



```
let tId = 'btnDelete';

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

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
Delete
praashibansal@Praashis-Mac-mini helloworld % 
```

TypeScript For

Below is the syntax for using the “for” loop statements.

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

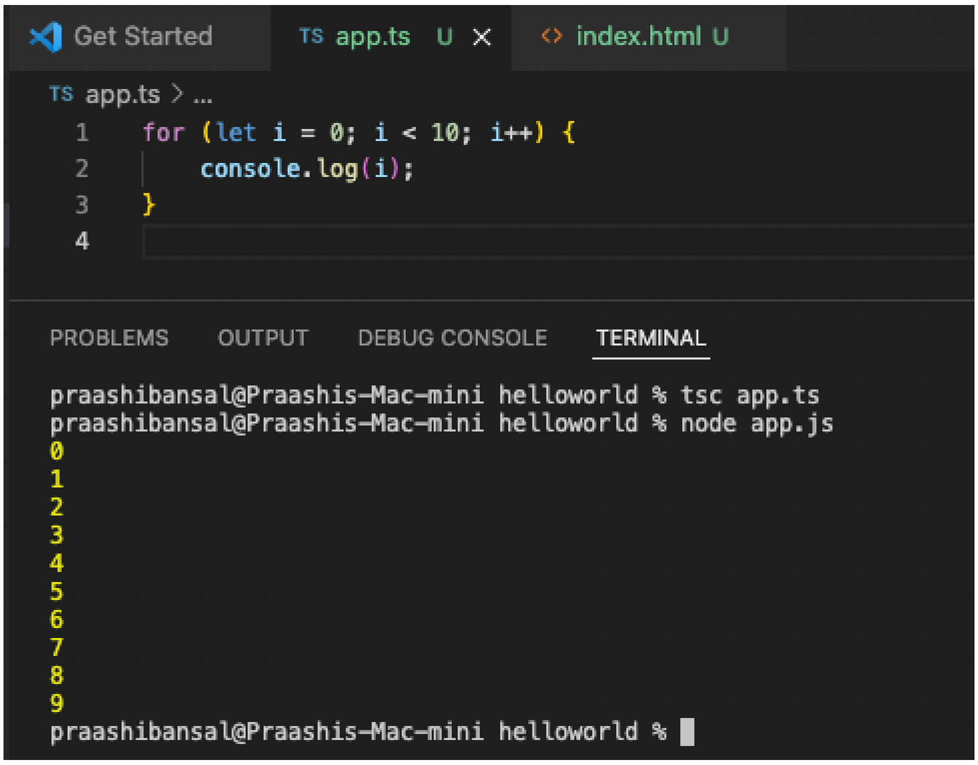
In the for loop, there are three optional expressions separated by semicolons (;) and enclosed in parentheses.

- Initialization

- Condition
- Expression

All these three expressions are optional, meaning you can use the for loop statement:

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

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



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

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
0
1
2
3
4
5
6
7
8
9
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript While

Using the while statement, you can create a loop that will execute a block of code as long as a condition states true.

Below is the syntax for the while loop.

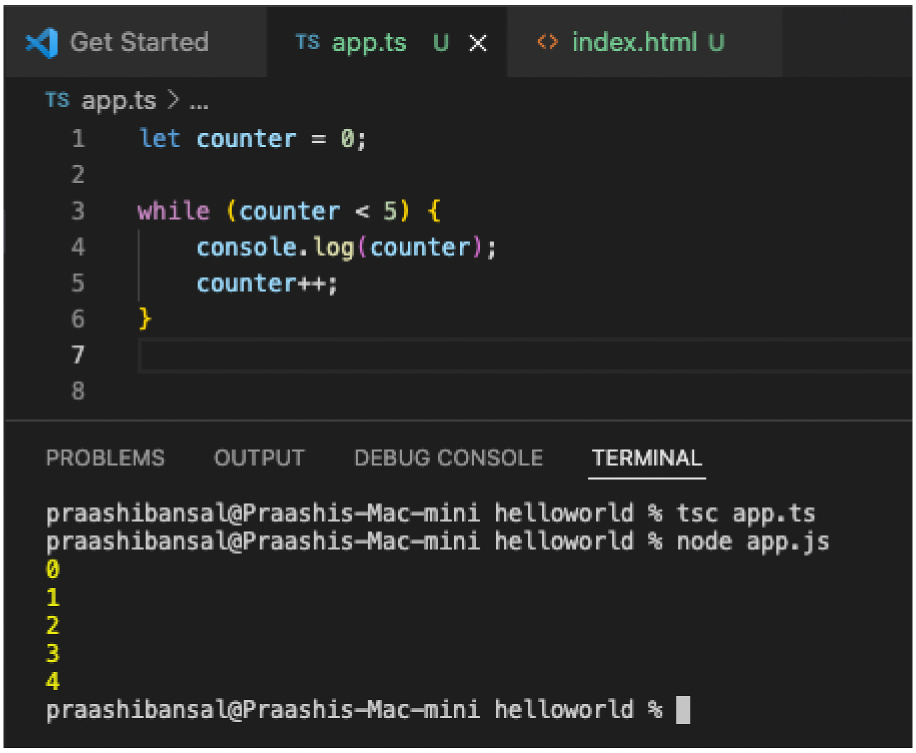
```
while(condition) {
    // do something
}
```

To break the loop immaturely based on another condition, you need to use the break statement:

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

    if(anotherCondition)
        break;
}
```

For example:

```
let counter = 0;

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



```
let counter = 0;

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

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
0
1
2
3
4
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Do..While

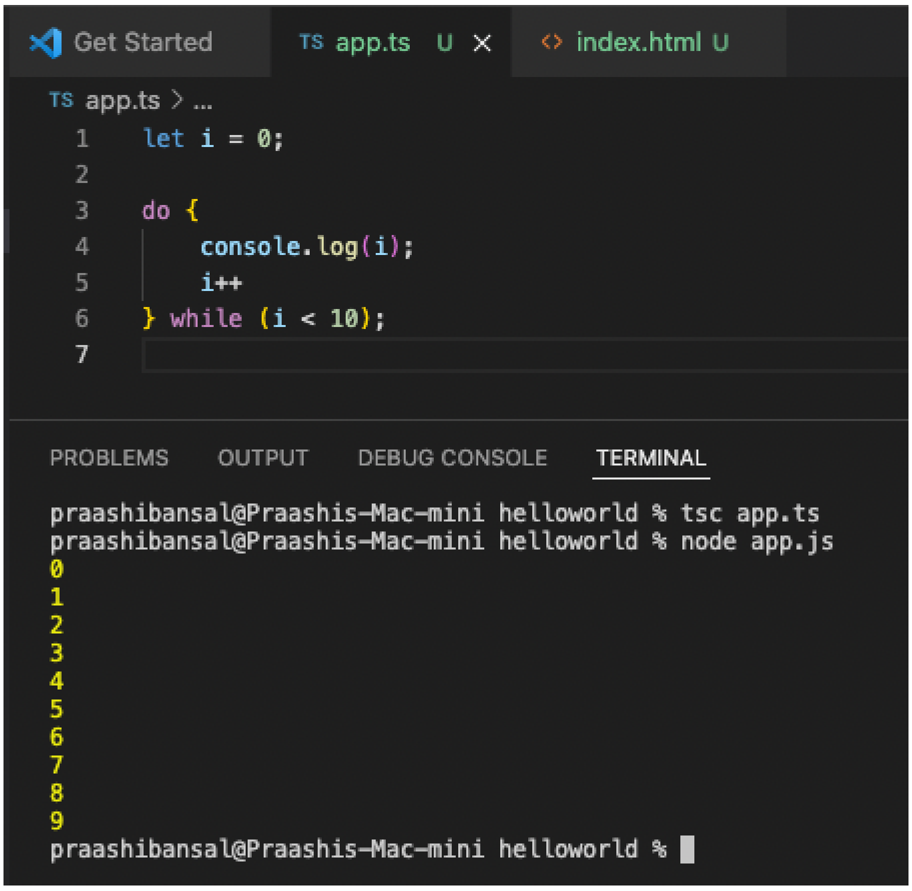
The following shows the syntax of the do...while statement.

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

This runs the code block till the condition evaluates to false. This statement always executes its loop body at least once, as the condition will be executed at the end of the code.

For example:

```
let i = 0;

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



```
let i = 0;

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

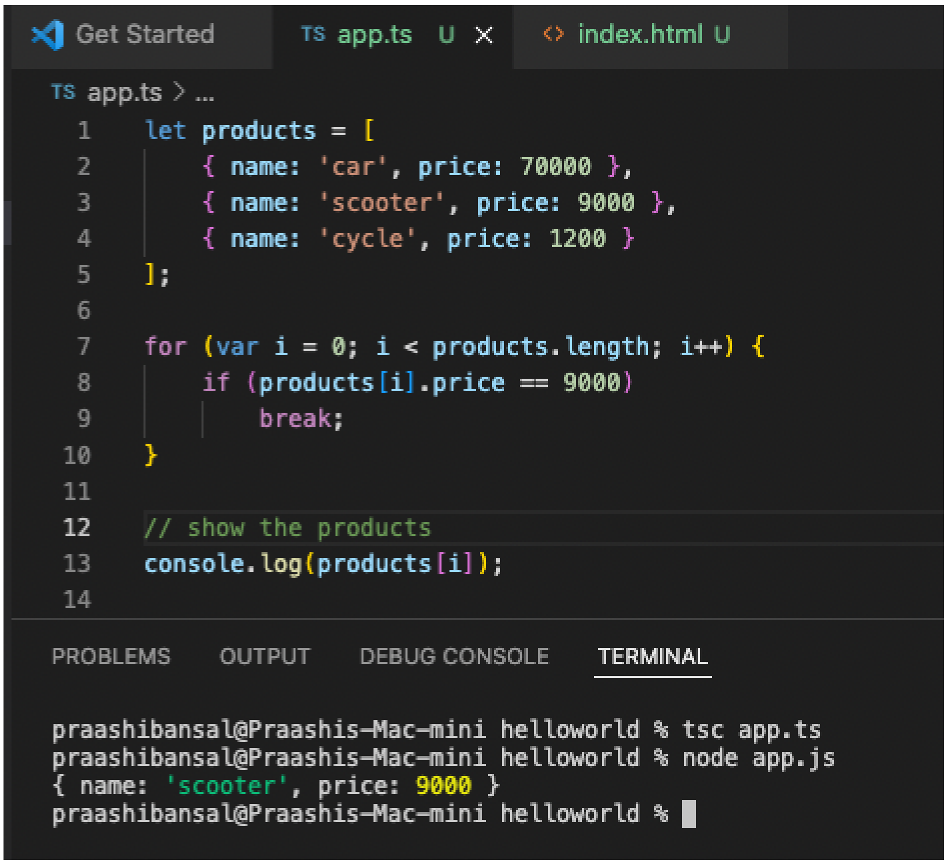
```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
0
1
2
3
4
5
6
7
8
9
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Break

The break statement is used to terminate any loop and pass the program flow to the next immediate statement. The for, while, and do...while statements support the use of the break statement.

For example:

```
let products = [
    { name: 'car', price: 70000 },
    { name: 'scooter', price: 9000 },
    { name: 'cycle', price: 1200 }
];

for (var i = 0; i < products.length; i++) {
    if (products[i].price == 9000)
        break;
}

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



```
let products = [
    { name: 'car', price: 70000 },
    { name: 'scooter', price: 9000 },
    { name: 'cycle', price: 1200 }
];

for (var i = 0; i < products.length; i++) {
    if (products[i].price == 9000)
        break;
}

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

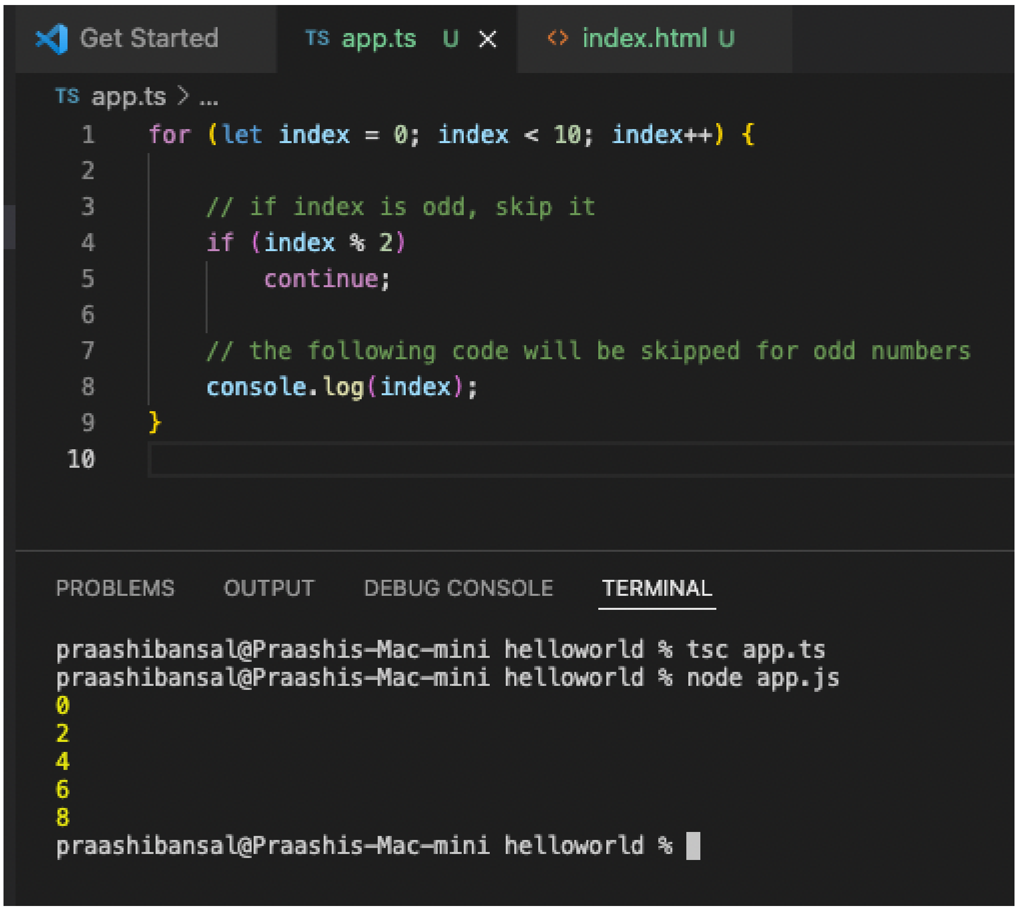
```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
{ name: 'scooter', price: 9000 }
praashibansal@Praashis-Mac-mini helloworld %
```

TypeScript Continue

This statement helps you control loops, such as a for loop, a while loop, or a do...while loop. It skips to the end of the loop and continues the next iteration.

For example, you can apply the continue statement in the for loop:

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

    // if index is odd, skip it
    if (index % 2)
        continue;

    // the following code will be skipped for odd numbers
    console.log(index);
}
```



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

    // if index is odd, skip it
    if (index % 2)
        continue;

    // the following code will be skipped for odd numbers
    console.log(index);
}

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
0
2
4
6
8
praashibansal@Praashis-Mac-mini helloworld %
```

3. Functions

A TypeScript function is any block of code that executes a specific task. To declare a function, use the keyword “function.”

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

You can also use type annotations in the function parameters return value of a function.

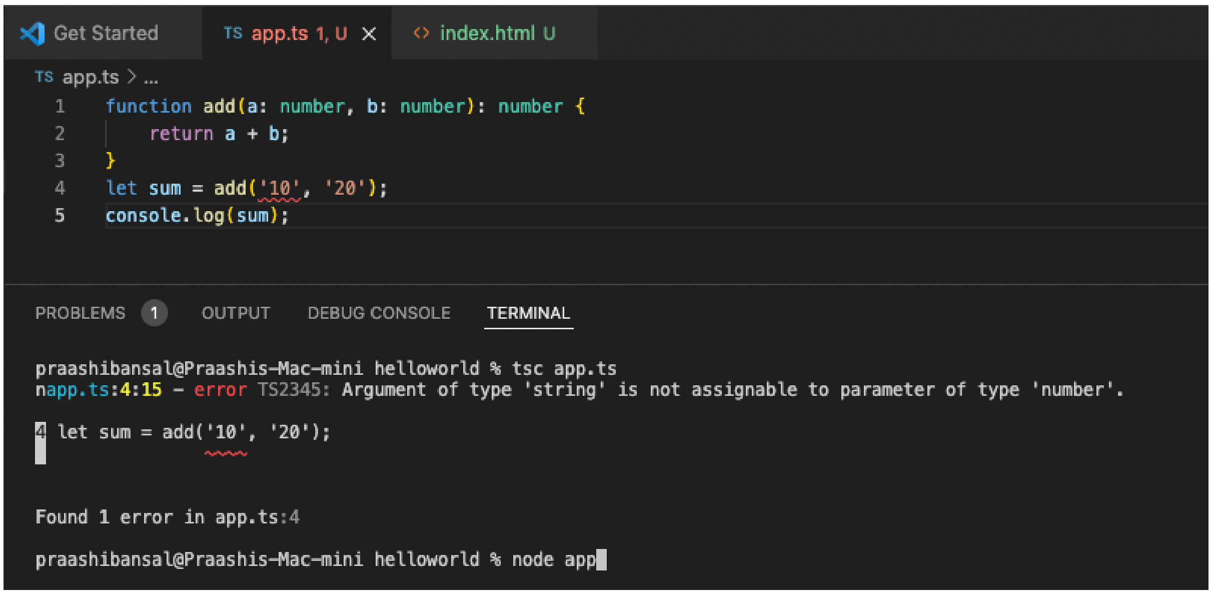
For example:

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

In the above example, the add() function will take two parameters with the number type.

Whenever you call the add() function in your program, the TypeScript compiler verifies each argument whether they are numbers or not. If you pass any other type of clothes numbers, it throws an error:

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



Get Started | TS app.ts 1, U | index.html U

```
function add(a: number, b: number): number {
    return a + b;
}
let sum = add('10', '20');
console.log(sum);
```

PROBLEMS 1 | OUTPUT | DEBUG CONSOLE | TERMINAL

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
napp.ts:4:15 - error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.

4 let sum = add('10', '20');
                ~~~~

Found 1 error in app.ts:4

praashibansal@Praashis-Mac-mini helloworld % node app
```

The return type is indicated by the :number that follows the parentheses. In this scenario, the add() function will return a value of the number type. The compiler will examine each return statement to ensure the return value is compatible with the function's return type if it exists.

You can specify void type if a function returns null. The function doesn't return any value, as indicated by the keyword void.

For example:

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

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

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

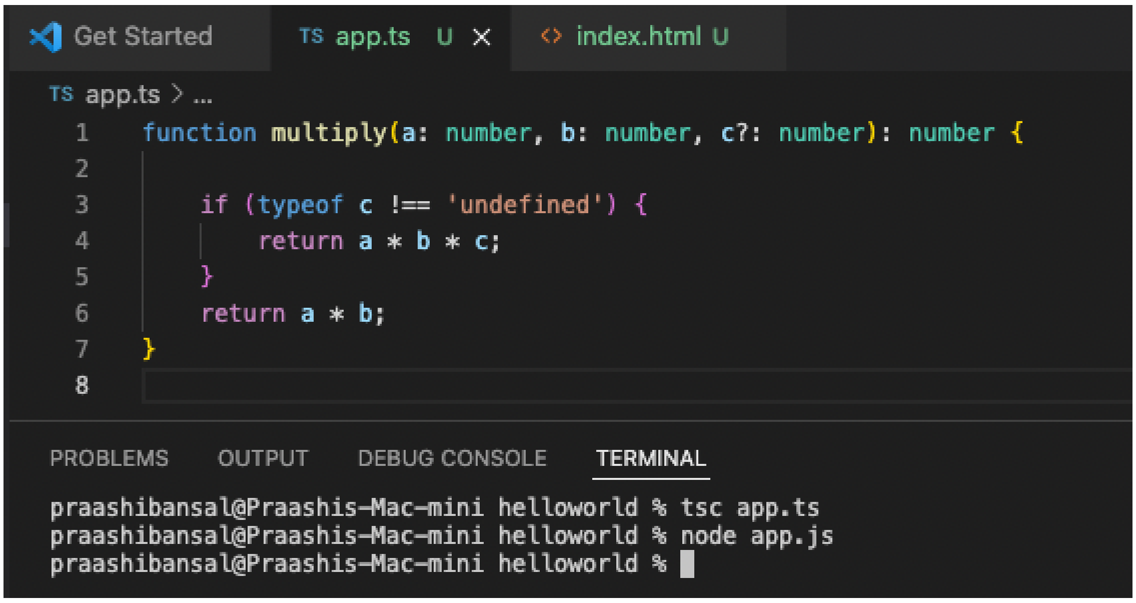
Optional Parameters

Unlike JavaScript, in TypeScript, every function call will be examined by the compiler, and if the number of arguments differs from the number of parameters listed in the function, an error will be produced. Additionally, if the types of the arguments and the types of the function parameters are incompatible, an error will be returned.

You must annotate optional parameters to instruct the compiler not to produce an error whenever you miss any arguments because the compiler will carefully inspect the passing arguments. Use the? symbol after the parameter name to make a function parameter optional.

For example:

```
function multiply(a: number, b: number, c?: number): number {

    if (typeof c !== 'undefined') {
        return a * b * c;
    }
    return a * b;
}
```



```
function multiply(a: number, b: number, c?: number): number {

    if (typeof c !== 'undefined') {
        return a * b * c;
    }
    return a * b;
}

```

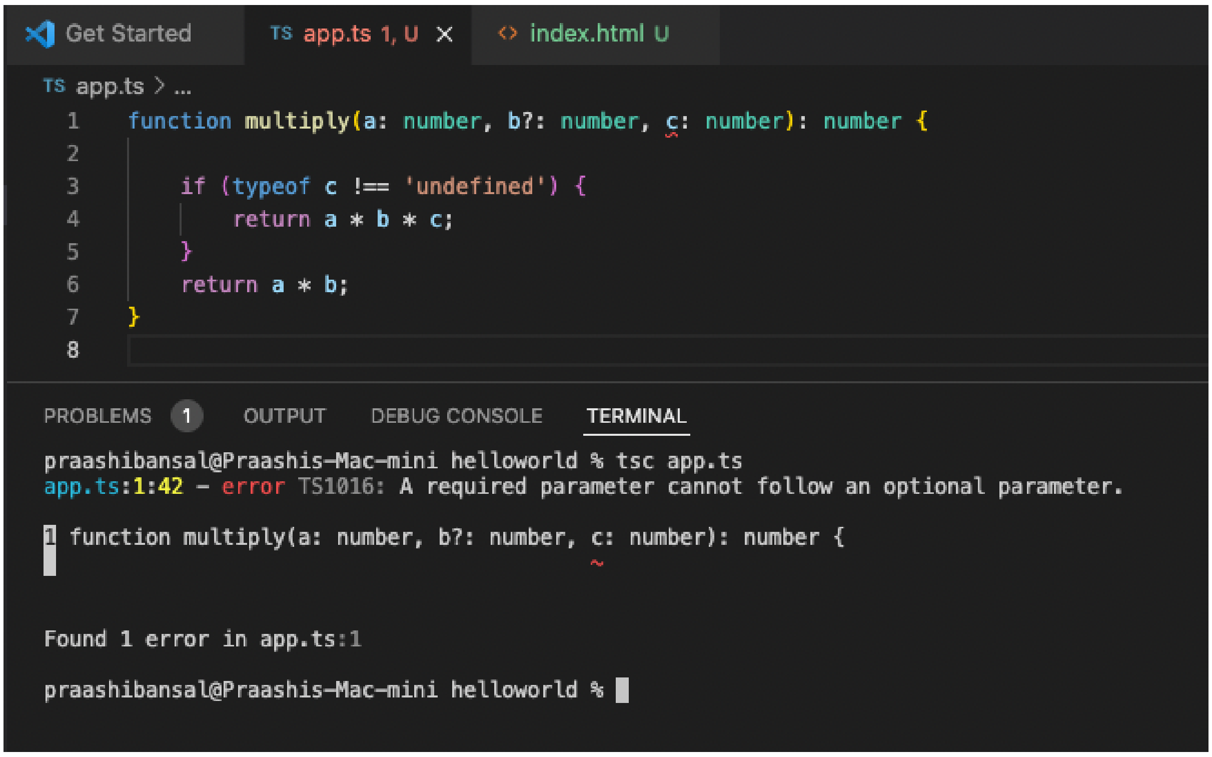
```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld %
```

You need to use the ? after the c parameter. Then, you need to check if the argument is passed to the function by using the expression typeof c !== 'undefined'.

The list of required parameters follows the list of the optional parameters.

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

```
function multiply(a: number, b?: number, c: number): number {

    if (typeof c !== 'undefined') {
        return a * b * c;
    }
    return a * b;
}
```



```
function multiply(a: number, b?: number, c: number): number {

    if (typeof c !== 'undefined') {
        return a * b * c;
    }
    return a * b;
}

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:1:42 - error TS1016: A required parameter cannot follow an optional parameter.

1 function multiply(a: number, b?: number, c: number): number {
                                           ~


Found 1 error in app.ts:1

praashibansal@Praashis-Mac-mini helloworld %
```

Default Parameters

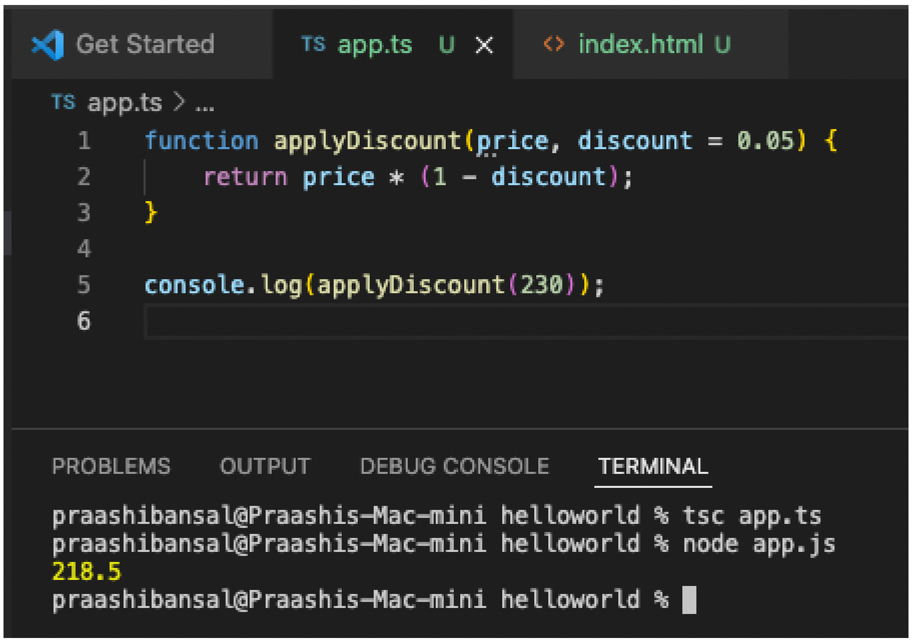
JavaScript has supported the default parameters since ES2015 (or ES6) with the following syntax:

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

While calling the function, it takes the default initialized values if you don’t pass any arguments or pass undefined arguments.

For example:

```
function applyDiscount(price, discount = 0.05) {
    return price * (1 - discount);
}

console.log(applyDiscount(230));
```



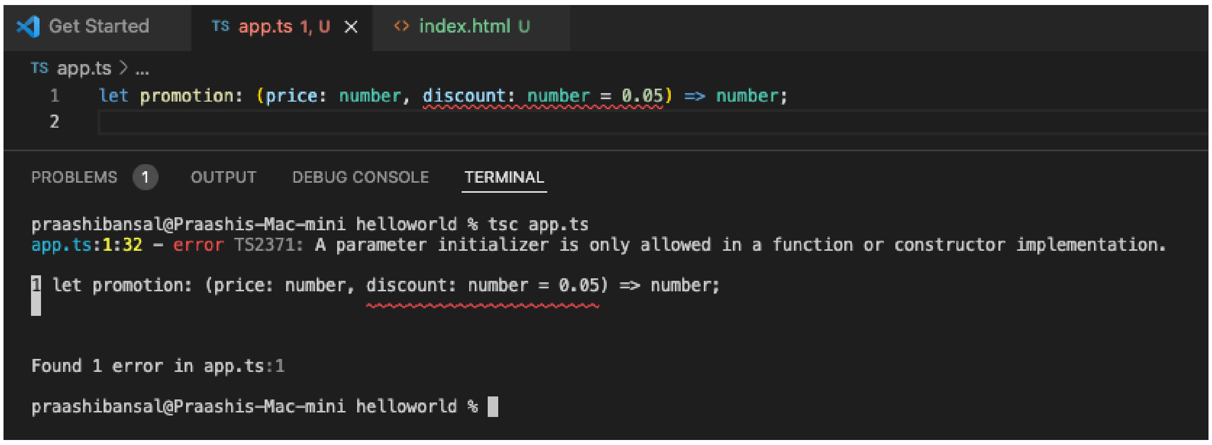
```
function applyDiscount(price, discount = 0.05) {
    return price * (1 - discount);
}

console.log(applyDiscount(230));
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
218.5
praashibansal@Praashis-Mac-mini helloworld %
```

Here, the discount parameter is the default parameter. The applyDiscount() function uses a default value of 0.05 if you don't pass the discount argument while calling it.

You will get an error if you pass the default parameters in function type definitions.

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



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

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:1:32 - error TS2371: A parameter initializer is only allowed in a function or constructor implementation.

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


Found 1 error in app.ts:1

praashibansal@Praashis-Mac-mini helloworld %
```

Function Overloading

With function overloading, you can create the relationship between a function's parameter types and result types.

For example:

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

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

The first function returns the sum of two numbers, whereas the later function concatenates two strings.

4. Class

TypeScript Class

A constructor function and prototype inheritance lets you create a “class.”

For example, we have created a Person class with three properties using the constructor function:

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

To access a person’s details, define a prototype method:

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

Then, create an object for the ‘person’ class and use it.

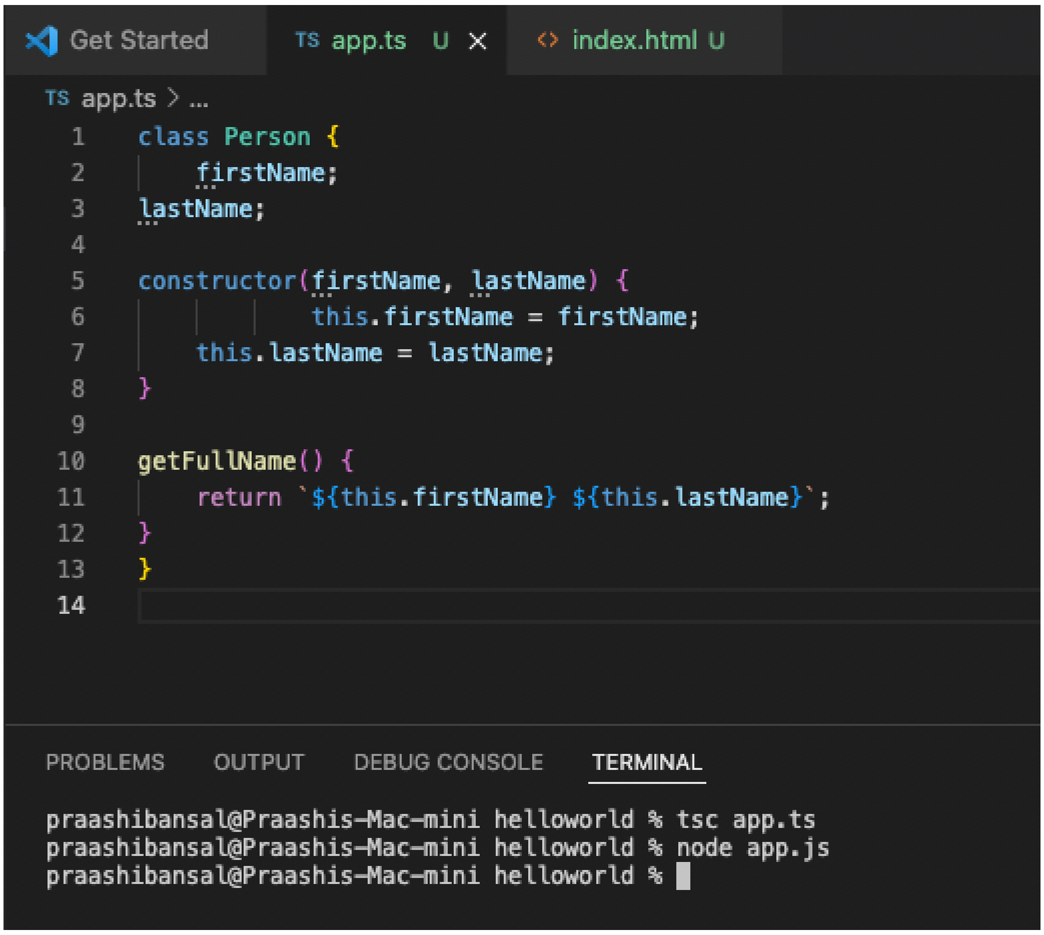
```
let person = new Person('Jimmy','Dwell');
console.log(person.getFullName());
```

But, in ES6, you can define a class for creating constructor function and prototypal inheritance:

```
class Person {
        firstName;
    lastName;

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

Here, we have defined the constructor inside the class. Now, let us add the getFullName() method to the same class:

```
class Person {
        firstName;
    lastName;

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

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

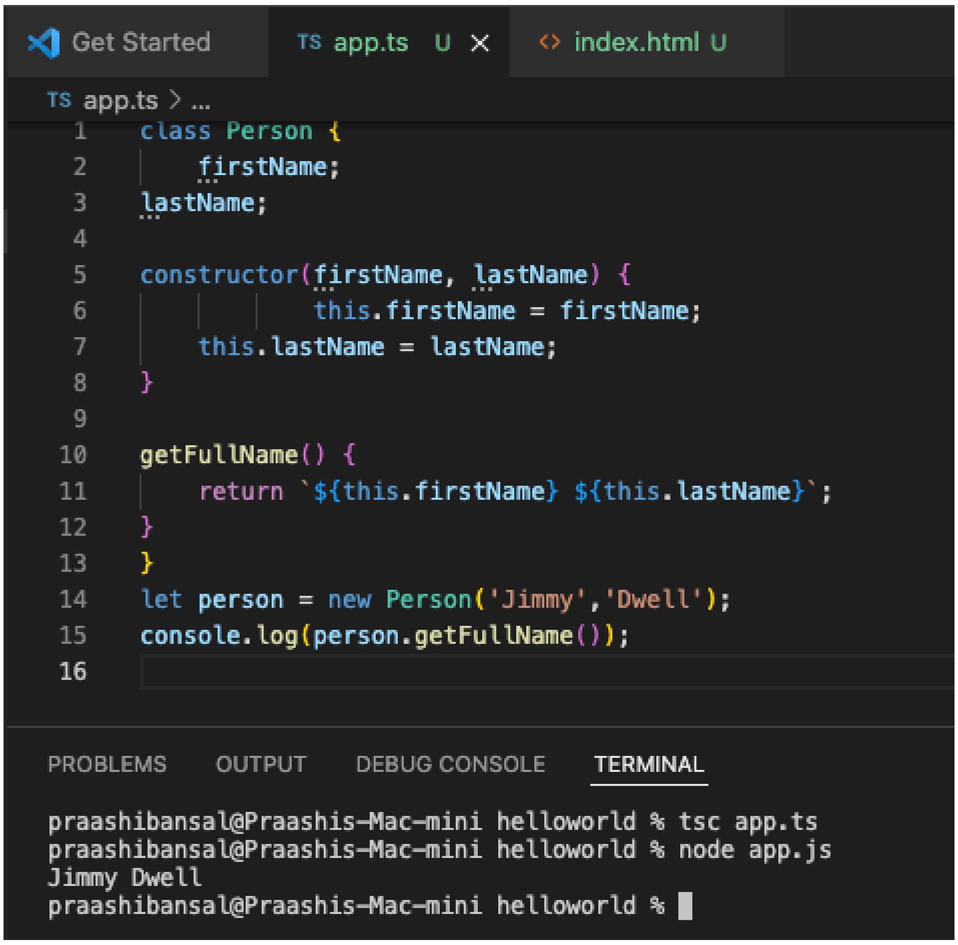


```
class Person {
    firstName;
lastName;

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

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

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld % 
```

The Person constructor function functions the same as the ‘Person’ class:

```
let person = new Person('Jimmy','Dwell');
console.log(person.getFullName());
```

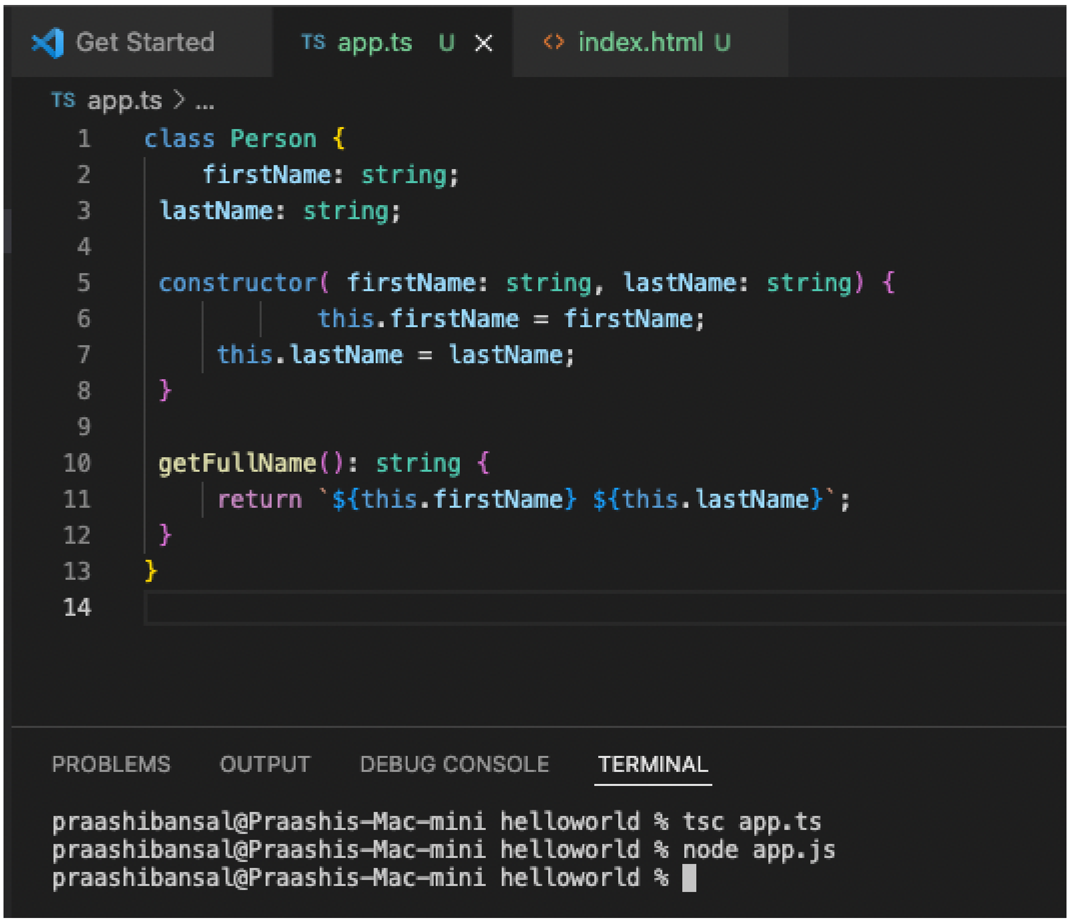


```
class Person {
    firstName;
lastName;

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

getFullName() {
    return `${this.firstName} ${this.lastName}`;
}
}
let person = new Person('Jimmy','Dwell');
console.log(person.getFullName());

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
Jimmy Dwell
praashibansal@Praashis-Mac-mini helloworld % 
```

The following code adds type annotations to the class's properties and methods:

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

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

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



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

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

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

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld %
```

The TypeScript compiler will carry out the type checks after annotating the types to properties, constructors, and methods.

TypeScript Access Modifiers

The visibility of a class's properties and methods is determined by access modifiers. Private, protected, and public are the three categories. The access is logically controlled by TypeScript during compilation rather than runtime.

The Private Modifier

When you use the private modifier for any property or method in a class, you can access them within the same class but not outside the class.

```
For example:

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

The Public Modifier

It is the default access modifier for all methods and properties. It lets you access properties and methods of a class from any location in a program.

Example:

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

The Protected Modifier

When you define the properties and methods as protected, they are accessible only within the same class and its subclasses. If you try to access protected properties or methods from other locations, you get an error.

```
class Person {

    protected ssn: string;

    // other code
}
```

TypeScript Readonly

TypeScript supports the readonly modifier, which marks the properties of a class immutable. You can declare the readonly property in the property declaration or in the constructor of the same class.

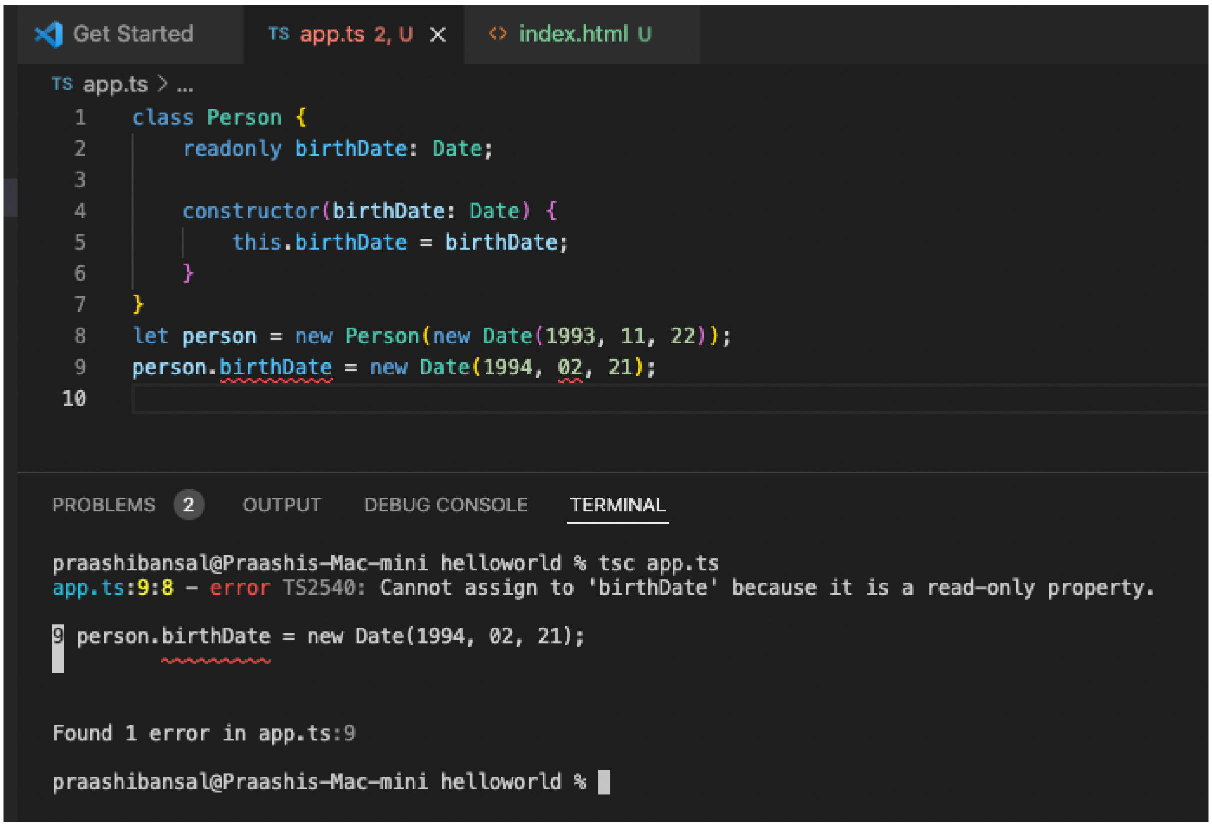
For example:

```
class Person {
    readonly birthDate: Date;

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

In this example, we have initialised the birthdate property as a readonly property in the Person class constructor.

If you reassign the birthDate property, you will get an error:

```
let person = new Person(new Date(1993, 11, 22));
person.birthDate = new Date(1994, 02, 21);
```



```
class Person {
    readonly birthDate: Date;

    constructor(birthDate: Date) {
        this.birthDate = birthDate;
    }
}
let person = new Person(new Date(1993, 11, 22));
person.birthDate = new Date(1994, 02, 21);
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:9:8 - error TS2540: Cannot assign to 'birthDate' because it is a read-only property.

9 person.birthDate = new Date(1994, 02, 21);
         ~~~~~~~~~

Found 1 error in app.ts:9

praashibansal@Praashis-Mac-mini helloworld %
```

Getters and Setters

Consider the following code, where the user inputs the age of a person.

```
class Person {
    public age: number;
    }

person.age = inputAge;
```

The inputAge can take any valid number. However, we will apply a condition for age:

```
if( inputAge > 0 && inputAge < 50 ) {
    person.age = inputAge;
}
```

It is pretty daunting to apply the checks everywhere in the code. This is where getters and setters come into the picture. They allow you to control the access to the properties of a class.

```
class Person {
    private _age: number;
     public get age() {
        return this._age;
    }

    public set age(theAge: number) {
        if (theAge <= 0 || theAge >= 50) {
            throw new Error('The age is invalid');
        }
        this._age = theAge;
    }

    public getFullName(): string {
        return `${this._age}`;
    }
}
```

TypeScript Inheritance

A class can use the properties of its parent class through inheritance. The child class must inherit the parent class in order to do that. The term "child class" refers to the class that will inherit another class, whereas "parent class" refers to the other class.

For example:

```
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}.`;
    }
}
```

If you want a child class to inherit the parent class, you need to use the extends keyword as any other programming language:

```
class Employee extends Person {
    //..code
}
```

Constructor

In the previous section's example, you can see that the person class comes with a constructor initializing the properties, such as firstName and lastName properties.

After that, initialize these properties in the constructor of the Employee class. This class calls the constructor of the person class to extend it.

Use super() to access the parent class's constructor.

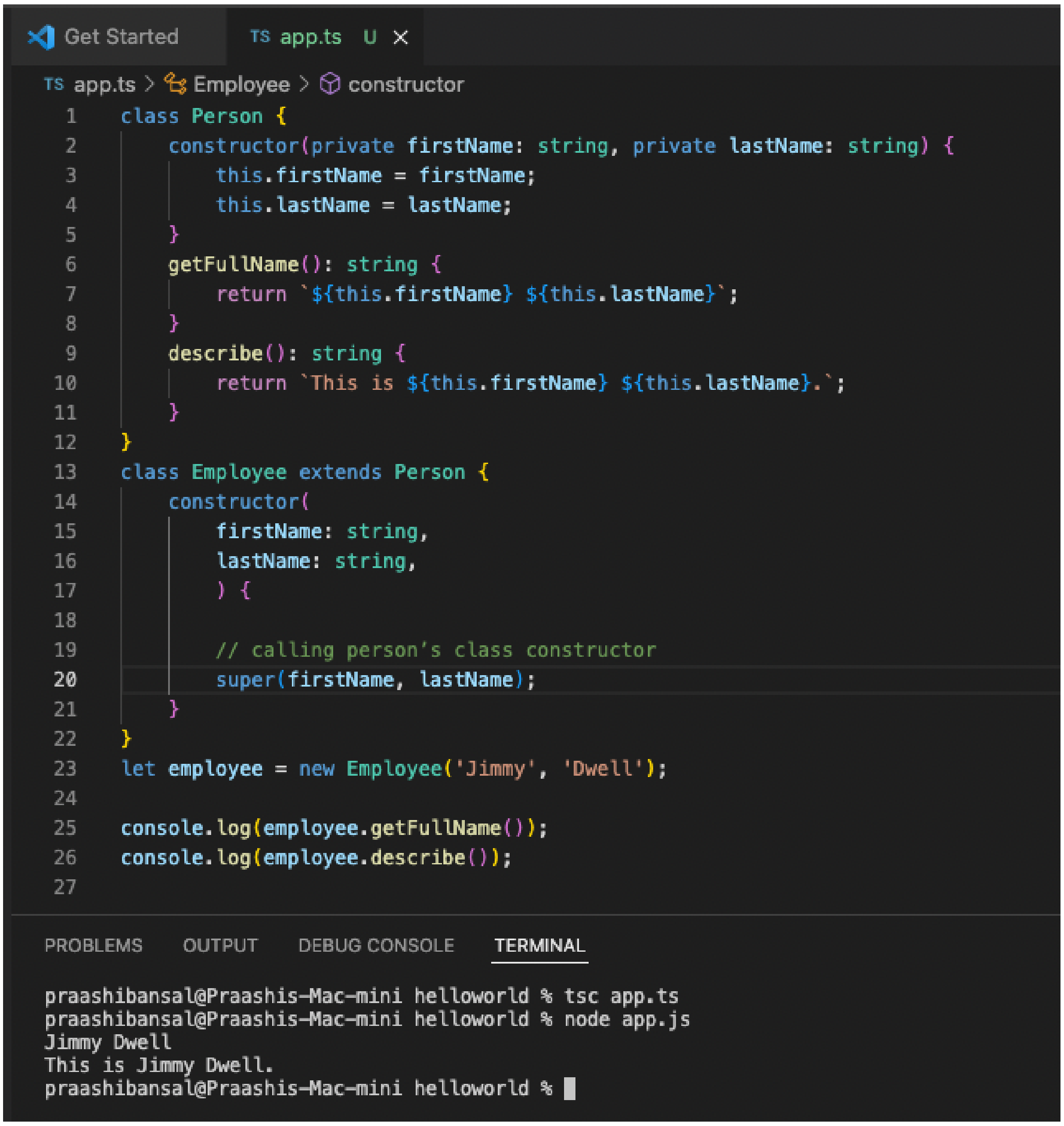
```
class Employee extends Person {
    constructor(
        firstName: string,
        lastName: string,
       ) {

        // calling person's class constructor
        super(firstName, lastName);
    }
}
```

The below code will create the instance for the employee class:

```
let employee = new Employee('Jimmy','Dwell');
```

With the instance of the employee class, you can access the persons class methods:

```
let employee = new Employee('Jimmy', 'Dwell');

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


Get Started
TS app.ts U
TS app.ts > Employee > constructor
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
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}.`;
    }
}
class Employee extends Person {
    constructor(
        firstName: string,
        lastName: string,
        ) {

        // calling person's class constructor
        super(firstName, lastName);
    }
}
let employee = new Employee('Jimmy', 'Dwell');

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

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
Jimmy Dwell
This is Jimmy Dwell.
praashibansal@Praashis-Mac-mini helloworld % 
```

Static Properties

Static properties are shared among all the classes, requiring you to use a static keyword. Also, if you want to access the static property, you must mention the classname or Propertyname:

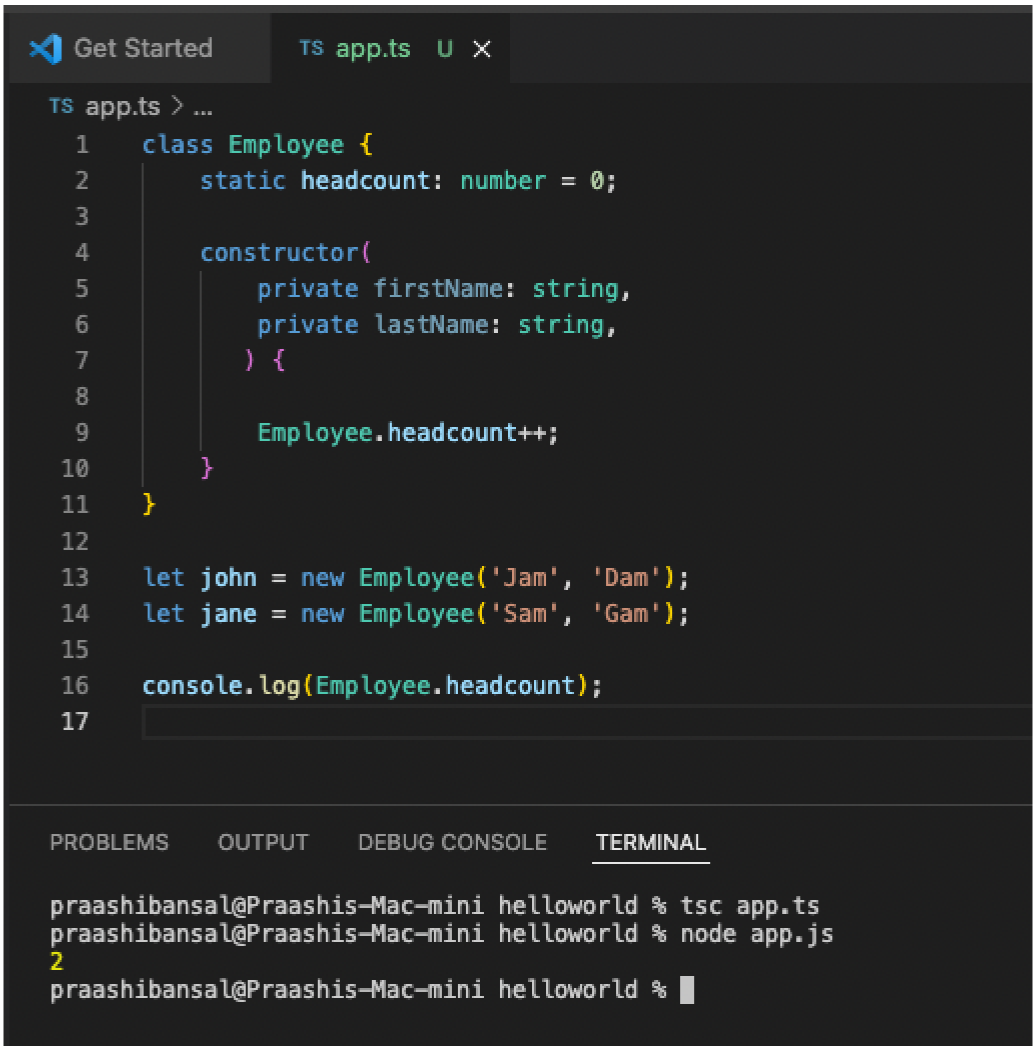
```
class Employee {
    static headcount: number = 0;

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

        Employee.headcount++;
    }
}
```

In the above example, we have mentioned the headcount as static initialized to zero. Whenever the object is created the headcount gets increased by one.

Now, we will create two employee objects:

```
let john = new Employee('Jam', 'Dam');
let jane = new Employee('Sam', 'Gam');

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

The output will be 2.



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

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

        Employee.headcount++;
    }
}

let john = new Employee('Jam', 'Dam');
let jane = new Employee('Sam', 'Gam');

console.log(Employee.headcount);

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
2
praashibansal@Praashis-Mac-mini helloworld %
```

Static Methods

Static methods are similar to a static property, as it is also shared across class instances. To specify a static method, you again have to use the static keyword before the method name.

For example:

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

    constructor(
        private firstName: string,
        private lastName: string,
       ) {
        Employee.headcount++;
    }

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

Abstract Class

The abstract class defines the common behaviors for derived classes to extend. The abstract keyword is used to declare the abstract classes.

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

More interestingly, abstract classes contain methods with no implementation. The classes that extend the abstract class contain the implementation of those methods.

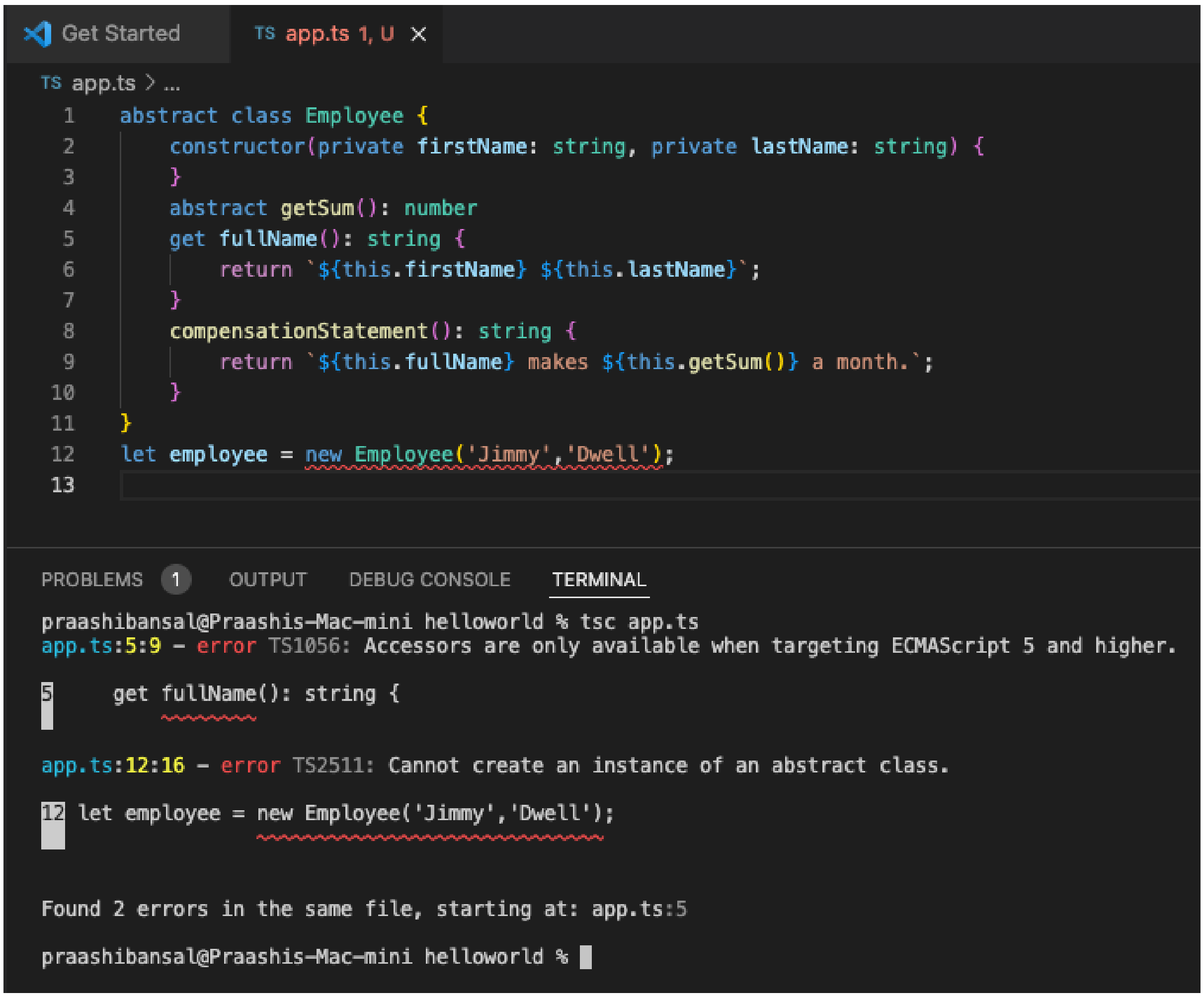
The following code has 'Employee' as abstract class and 'getSum' as the abstract method:

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

You cannot create the object of the 'Employee' class as it is an abstract class.

```
let employee = new Employee('Jimmy','Dwell');
```

You will get an error.



```
abstract class Employee {
    constructor(private firstName: string, private lastName: string) {
    }
    abstract getSum(): number
    get fullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
    compensationStatement(): string {
        return `${this.fullName} makes ${this.getSum()} a month.`;
    }
}
let employee = new Employee('Jimmy','Dwell');
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
app.ts:5:9 - error TS1056: Accessors are only available when targeting ECMAScript 5 and higher.

5     get fullName(): string {
          ~~~~~~~~

app.ts:12:16 - error TS2511: Cannot create an instance of an abstract class.

12 let employee = new Employee('Jimmy','Dwell');
                  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


Found 2 errors in the same file, starting at: app.ts:5

praashibansal@Praashis-Mac-mini helloworld % 
```

The Emp class is inheriting the Employee class:

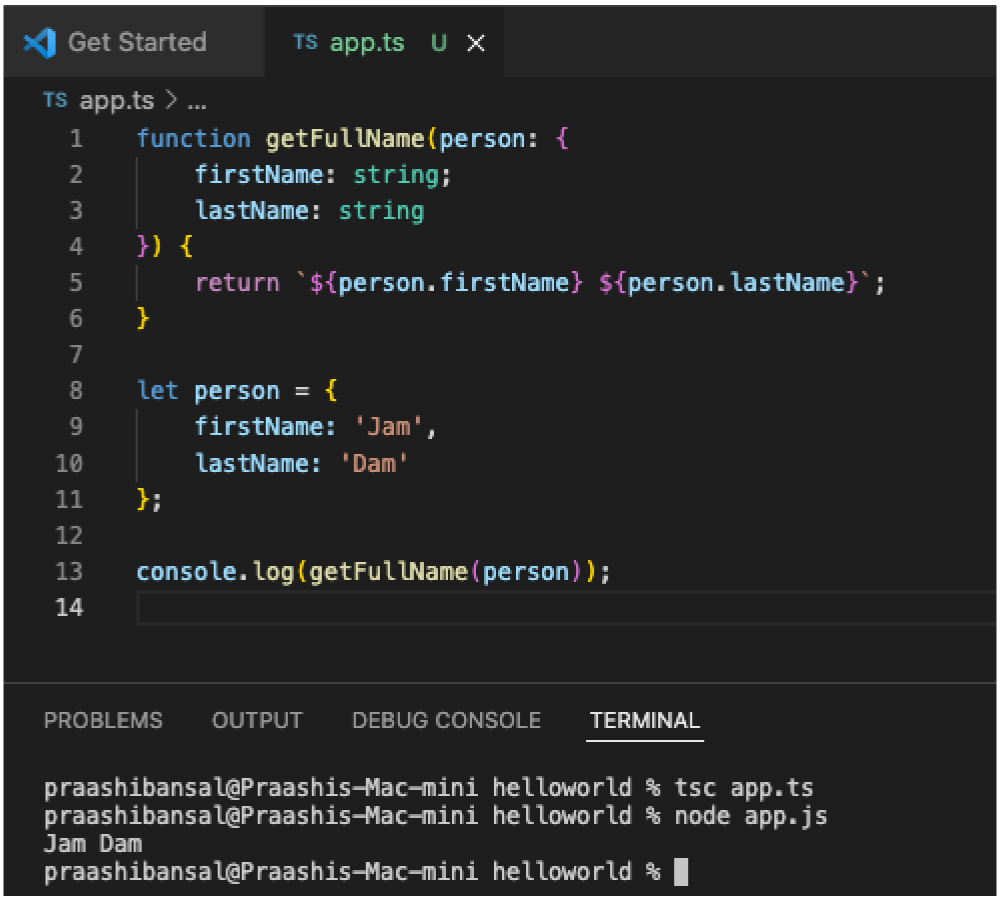
```
class Emp extends Employee {
    constructor(firstName: string, lastName: string, private
salary: number) {
        super(firstName, lastName);
    }
    getSum(): number {
        return this.salary;
    }
}
let jim = new FullTimeEmployee('Jim', 'Dim', 3400);
console.log(john.compensationStatement());
```

5. Interfaces

Interfaces

If you want to define contracts within your TypeScript code, you can use the interfaces. It lets you use the explicit names for type checking. For example:

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

let person = {
    firstName: 'Jam',
    lastName: 'Dam'
};

console.log(getFullName(person));
```



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

let person = {
    firstName: 'Jam',
    lastName: 'Dam'
};

console.log(getFullName(person));
```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
Jam Dam
praashibansal@Praashis-Mac-mini helloworld %
```

The compiler will check the passed arguments for the getFullName function. As mentioned, the passed arguments should be string.

If any of them don’t match, it results in an error. But, as you see due to type annotation, the code has become complex and difficult to read. Thus, we use interfaces to overcome this readability issue.

We have created an interface named ‘Person’ with two properties:

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

You can use the above interface to define other classes or methods.

Optional Properties

With interface, you can also have optional properties. An optional property is specified by the question mark (?) at the end of the property name within the declaration part:

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

In the above example, the middle name is declared as the optional parameter. If the user does not pass the argument for this variable, there will be no compilation error.

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

Interfaces Extending One Interface

For this concept, we are using an interface called person containing two methods called male()and female():

```
interface Person {
    male(g: string): boolean
    female(g: string): boolean
}
```

Suppose this interface has already been implemented by several classes. Perhaps we want to add a method girl to the ‘Person’ interface.

```
girl(g: string): void
```

But, it will break the current code. To avoid that, we will create a new interface and extend it to the ‘Person’ interface:

```
interface family extends Person {
    girl(g: string): boolean
}

We use the extend keyword to extend one interface to another:

interface A {
    a(): void
}

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

6. Advanced Types

TypeScript Intersection Types

If you want to create a new type by combining several existing types, you can use intersection types in TypeScript. The newly created type will have the features of the existing one. To combine the types, you need to use the & operator:

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

The new typeAB will have the features of both A and B. Also, the union type (|) assures the variable can have a value of either type A or B.

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

To explain this concept, we use three interfaces: BusinessPartner, Identity, and Contact.

```
interface BusinessPartner {
    name: string;
    credit: number;
}

interface Identity {
    id: number;
    name: string;
}

interface Contact {
    email: string;
    phone: string;
}
```

Below are the two different intersection types:

type Employee = Identity & Contact; //containing the properties of both Identity and Contact type. For example:

```
let e: Employee = {
    id: 10,
    name: 'ping pong',
    email: 'ping.pong@example.com',
    phone: '(408)-897-5684'
};
```

type Customer = BusinessPartner & Contact; // containing all the properties of the BusinessPartner and Contact type. For example:

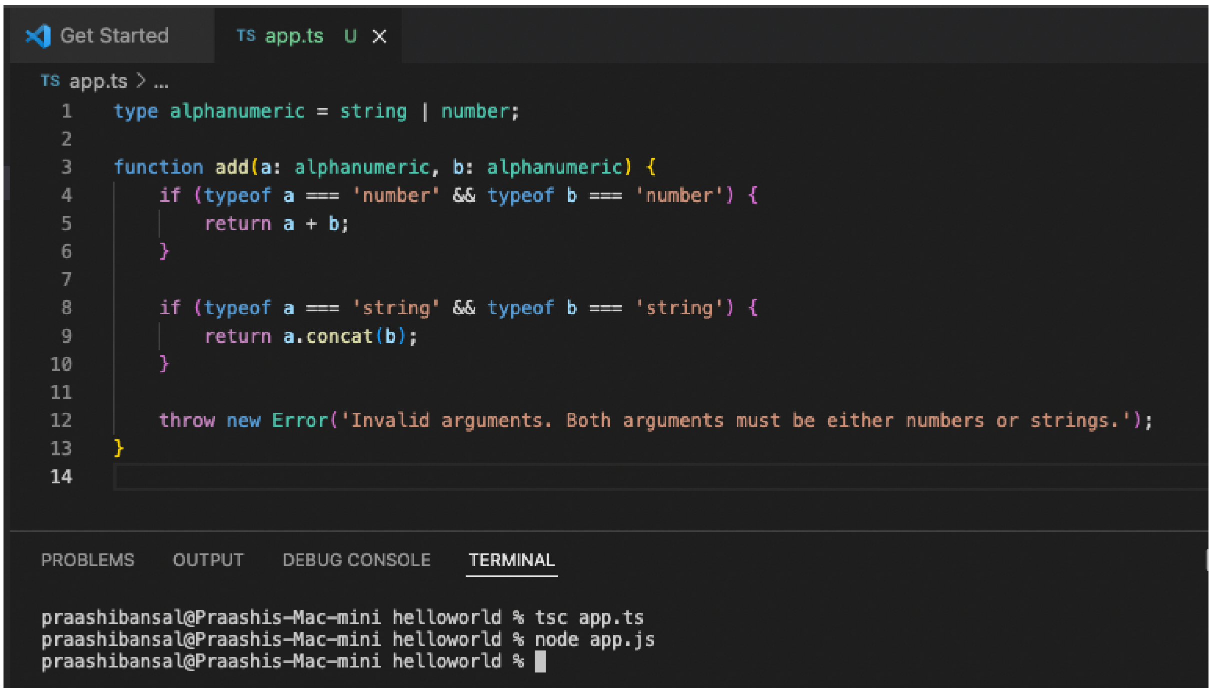
```
type Customer = BusinessPartner & Contact;

let c: Customer = {
    name: 'ABC Inc.',
    credit: 10000,
    email:'hello@abcinc.com',
    phone: '(408)-897-5735'
};
```

TypeScript Type Guard

If you want to narrow down the types of variables within a conditional block, you can use Type guards.

Typeof

```
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.');
}
```



```
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.');
}

```

```
praashibansal@Praashis-Mac-mini helloworld % tsc app.ts
praashibansal@Praashis-Mac-mini helloworld % node app.js
praashibansal@Praashis-Mac-mini helloworld % 
```

The above function checks if both the argument types are numbers and are using the typeof operator. It computes the sum of both the arguments if they use typeof.

The same is for the string type argument. However, it will concatenate two arguments rather than performing the sum.

If the arguments are neither numbers nor strings, you will get an error.

7. Generics

To create reusable and generalised forms of functions, classes, and interfaces, TypeScript offers generics.

Below is the example of a generic function returning the random element from the R type array:

```
function getRandomElement<R>(items: R[]): R {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

The above function will use the R type array that captures the type provided while calling the function. Also, the function has a return type of R. The getRandomElement() function works as a generic as it can work with any data type. You can use any letter rather than R.

Calling a Generic Function

The getRandomElement() can also work with arrays:

```
let numbers = [1, 5, 7, 4, 2, 9];
let randomEle = getRandomElement<number>(numbers);
console.log(randomEle);
```

The above code passes the numbers to the getRandomElement() function as the R type.

Here, we are using the type inference for the argument, so the TypeScript compiler can set the value of R automatically depending on the type of argument you will pass.

```
let numbers = [1, 5, 7, 4, 2, 9];
let randomEle = getRandomElement(numbers);
console.log(randomEle);
```

Here, we have not specified the return type of the function, it will be done by the compiler by looking at the arguments passed. The function getRandomElement() is now type-safe as well. However, if you assign a string variable, an error will occur.

```
let numbers = [1, 5, 7, 4, 2, 9];
let returnElem: string;
returnElem = getRandomElement(numbers);
```

TypeScript Generic Classes

A generic class consists of a generic type parameter list. The class name follows the parameter list enclosed within the angle brackets <>.

```
class className<T>{
    //...
}
```

It is possible to have multiple generic types in a parameter list.

For example:

```
class className<K,T>{
    //...
}
```

TypeScript Generic Interface

You can also create a generic interface just like we did with the classes. A generic interface consists of a generic type parameter list. The interface name follows the parameter list enclosed within the angle brackets <>.

```
interface interfaceName<R> {
    // ...
}
```

All the members of the interface can see the type parameter R. There can be more types in the type parameter list.

For example:

```
interface interfaceName<U,V> {
    // ...
}
```

8. TypeScript Modules

Creating Modules

You can create modules in TypeScript. Below we create Check.ts with an declared interface Check:

```
export interface Check {
    isValid(s: string): boolean
}
```

We have used the export keyword before the interface so other modules can use this interface. Otherwise, the check interface will be private to the Check.ts module.

Export Statements

To export declaration from a module, you can use the export statement:

```
interface Check {
    isValid(s: string): boolean
}

export { Check };
```

You can also rename the declarations for the module:

```
interface Check {
    isValid(s: string): boolean
}
export { Check as StringCheck };
```

Another module can use the Check interface as the StringCheck interface.

Importing a New Module

Use the import statement within your code to use a new module. In the following code, we have created a new module CheckSum using the Check.ts module:

```
import { Check } from './Check';

class CheckSum implements Check {
    isValid(s: string): boolean {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(s);
    }
}

export { CheckSum };
```

While importing the module, you can rename it:

```
import { Check as StringCheck } from './Check';

Inside the CheckSum module, you use the Check interface as the
StringCheck interface instead:

import { Check as StringCheck } from './Check';

class CheckSum implements StringCheck {
    isValid(s: string): boolean {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(s);
    }
}

export { CheckSum };
```

Importing Everything from a Module

To import everything from a module, you can use the following syntax:

```
import * from 'module_name';
```

Re-Exports

Below, we are creating a new module called CheckSal.ts using the Check.ts module:

```
import { Check } from './Check';

class CheckSal implements Check {
    isValid(s: string): boolean {
        const numberRegexp = /^[0-9]+$/;
        return s.length === 5 && numberRegexp.test(s);
    }
}

export { CheckSal };
```

TypeScript React Cheatsheet

In this react TypeScript cheatsheet we will learn react TypeScript types. To import React, use the following commands:

```
import * as React from "react";
import * as ReactDOM from "react-dom";
```

Function Components

You can use them as the normal functions that take a props argument and return a JSX element:

```
// Declaring type of props
type AppProps = {
  message: string;
};

// Declare a Function Component; return type is inferred.
const App = ({ message }: AppProps) => <div>{message}</div>;

// Annotate the return type
const App = ({ message }: AppProps): JSX.Element =>
<div>{message}</div>;

// Inline the type declaration
const App = ({ message }: { message: string }) =>
<div>{message}</div>;
```

Conclusion

In this React TypeScript cheat sheet, we have covered every aspect with simple syntax and examples with running outputs from VS Code editor.

TypeScript is a simple and interactive language like any other object-oriented programming language that allows you to work with objects, classes, modules, interfaces, inheritance, and more. You can keep this TypeScript cheat sheet handy to prepare for your upcoming interview.