

TypeScript

TypeScript is an open-source language which builds on JavaScript, one of the world's most used tools, by adding static type definitions

Types provide a way to describe the shape of an object, providing better documentation, and allowing TypeScript to validate that your code is working correctly.

All valid JavaScript code is also TypeScript code. You might get type-checking errors, but that won't stop you from running the resulting JavaScript. While you can go for stricter behavior, that means you're still in control.

TypeScript code is transformed into JavaScript code via the TypeScript compiler or [Babel](#). This JavaScript is clean, simple code which runs anywhere JavaScript runs: In a browser, on Node.JS or in your apps.

Installation & setup

TypeScript is available in Node.js library. So, install node.js first . Then install TypeScript software from Node library using npm tool

npm install -g typescript

Data types

Boolean

Number

String

Array

object

Tuple

Any

Void

Variable declaration

Syntax:

```
let var-name : datatype;
```

Ex:

```
let n:number=10;  
let b:Boolean=true;  
let s:string="hello"  
let data:numbers[]={ 10,20,30}  
let emp:object = { name:"john",age:20}
```

Note:

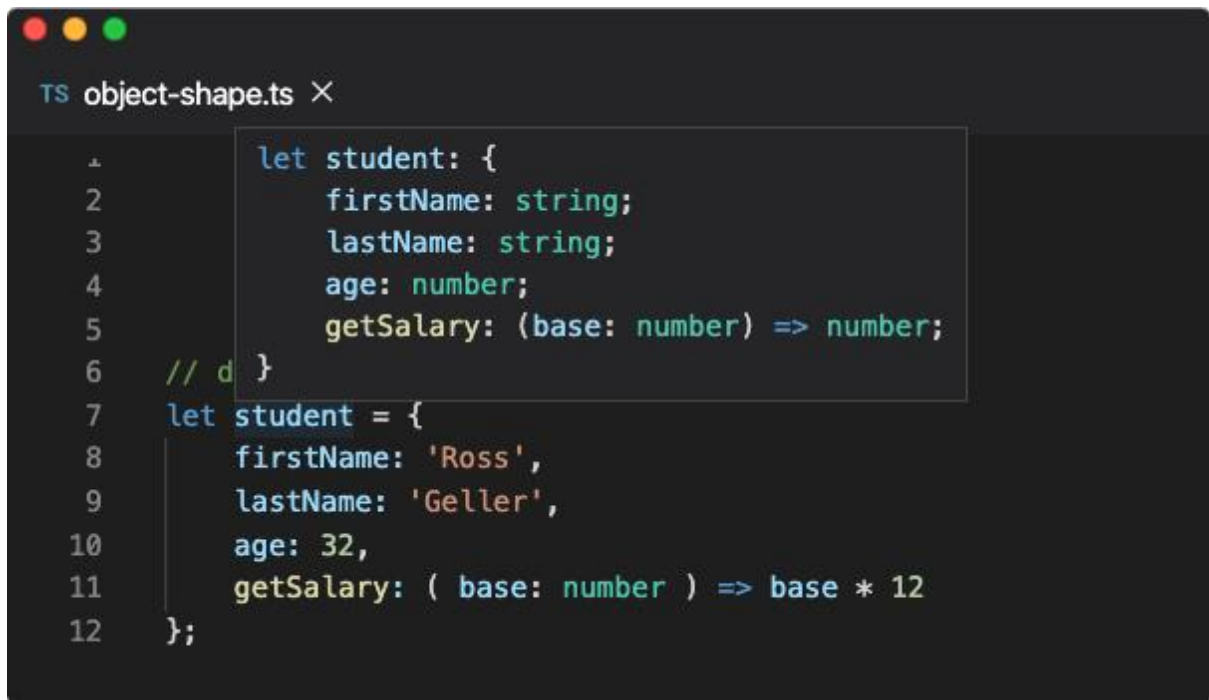
Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

```
// Declare a tuple type  
let x: [string, number];  
  
// Initialize it  
  
x = ["hello", 10]; // OK
```

Interfaces

Interfaces can be used to create new type and also to provide a powerful way of defining contracts within your code as well as contracts with code outside of your project.

When we define an object with properties (*keys*) and values, TypeScript creates an **implicit interface** by looking at the property names and data type of their values in the object. This happens because of the **type inference**.

A screenshot of a code editor window titled "TS object-shape.ts". The code defines a variable 'student' with a specific shape. A tooltip is visible over the opening curly brace of the object literal, showing the inferred type: `let student: {
 firstName: string;
 lastName: string;
 age: number;
 getSalary: (base: number) => number;
}`. The main code in the editor is:

```
1  let student: {  
2    firstName: string;  
3    lastName: string;  
4    age: number;  
5    getSalary: (base: number) => number;  
6  }  
7  // d }  
8  let student = {  
9    firstName: 'Ross',  
10   lastName: 'Geller',  
11   age: 32,  
12   getSalary: ( base: number ) => base * 12  
};
```

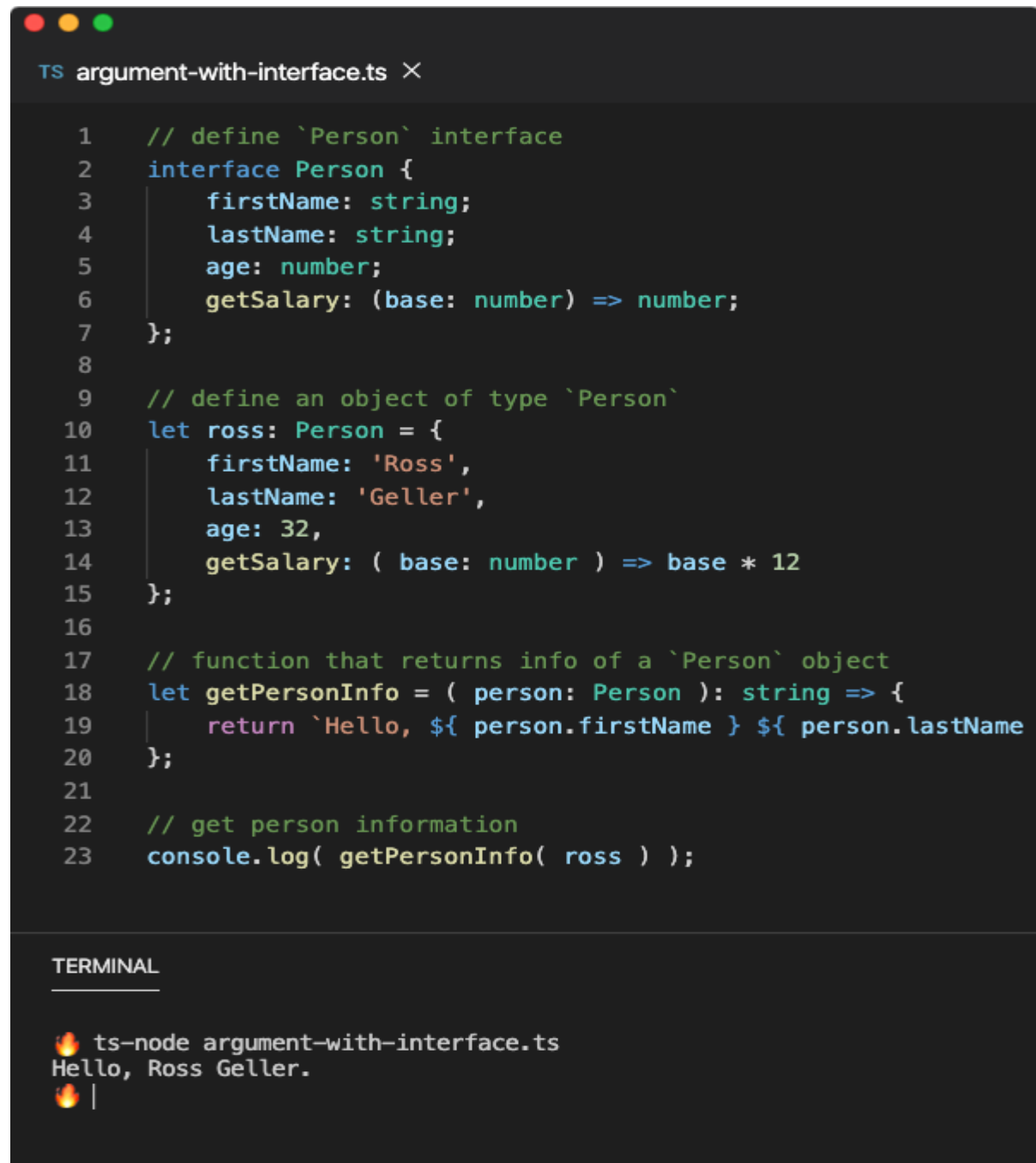
In the above example, we have created an object `student` with `firstName`, `lastName`, `age` and `getSalary` fields and assigned some initial values. Using this information, TypeScript creates an implicit interface type for `student`.

An interface is just like an object but it only contains the information about object properties and their types. We can also create an interface type and give it a name so that we can use it to annotate object values but here, this interface doesn't have a name since it was created implicitly.

Declaration of an Interface

An interface can be declared with properties and method declarations without definitions.

Ex:



```
TS argument-with-interface.ts ×

1  // define `Person` interface
2  interface Person {
3      firstName: string;
4      lastName: string;
5      age: number;
6      getSalary: (base: number) => number;
7  };
8
9  // define an object of type `Person`
10 let ross: Person = {
11     firstName: 'Ross',
12     lastName: 'Geller',
13     age: 32,
14     getSalary: ( base: number ) => base * 12
15 };
16
17 // function that returns info of a `Person` object
18 let getPersonInfo = ( person: Person ): string => {
19     return `Hello, ${ person.firstName } ${ person.lastName }`;
20 };
21
22 // get person information
23 console.log( getPersonInfo( ross ) );
```

TERMINAL

```
🔥 ts-node argument-with-interface.ts
Hello, Ross Geller.
🔥 |
```

In the example above, we have defined an interface `Person` that describes the shape of an object, but this time, we have a name we can use to refer to this type. We have used this type to annotate `ross` variable as well as the `person` argument of the `getPersonInfo` function. This will inform TypeScript to validate these entities against the shape of `Person`.

Readonly properties in interface:

Some properties should only be modifiable when an object is first created. You can specify this by putting `readonly` before the name of the property:

```
interface Point {  
    readonly x: number;  
    readonly y: number;  
}
```

Note: readonly vs const

The easiest way to remember whether to use `readonly` or `const` is to ask whether you're using it on a variable or a property. Variables use `const` whereas properties use `readonly`.

Functions

Functions are the fundamental building block of any application in JavaScript. TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.

Just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function.

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Ex:

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

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

Optional & Default parameters:

In TypeScript, every parameter is assumed to be required by the function. when the function is called, the compiler will check that the user has provided a value for each parameter. The number of arguments given to a function has to match the number of parameters the function expects.

Optional parameters can be created by adding "?" after parameter. Then compiler wont consider it as a required parameter.

Ex:

```
function buildName(firstName: string, lastName?: string) {
    if (lastName) return firstName + " " + lastName;
    else return firstName;
}
```

```
let result1 = buildName("Bob"); // works correctly now
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
Expected 1-2 arguments, but got 3.
let result3 = buildName("Bob", "Adams"); // ah, just right
```

Default parameters:

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place. These are called default-initialized parameters.

Ex:

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}
```

```
let result1 = buildName("Bob"); // works correctly now, returns "Bob
                                //Smith"
let result2 = buildName("Bob", undefined); // still works, also returns
                                            //"Bob Smith"
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many
```

```
Expected 1-2 arguments, but got 3. //parameters  
Expected 1-2 arguments, but got 3.  
let result4 = buildName("Bob", "Adams"); // ah, just right
```

lasses

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers can build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Ex:

```
class Greeter {  
    greeting: string;  
  
    constructor(message: string) {  
        this.greeting = message;  
    }  
  
    greet() {  
        return "Hello, " + this.greeting;  
    }  
}  
  
let greeter = new Greeter("world");
```

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take a look at an example:

```
class Animal {  
    move(distanceInMeters: number = 0) {  
        console.log(`Animal moved ${distanceInMeters}m.`);  
    }  
}  
  
class Dog extends Animal {  
    bark() {  
        console.log("Woof! Woof!");  
    }  
}  
  
const dog = new Dog();  
dog.bark();  
dog.move(10);  
dog.bark();
```

Public, private, and protected modifiers

Access modifiers are used to decide visible scope of class contents.

When we do not use any modifiers, it is public by default. When a class member is "public", it can be accessible from anywhere.

When a class member is applied with "private" modifier, that member is visible (accessible) only within that class, but not from outside.

The protected modifier acts much like the private modifier with the exception that members declared protected can also be accessed within deriving classes.

Ex:

```
class Person {  
  protected name: string;  
  constructor(name: string) {  
    this.name = name;  
  }  
}  
  
class Employee extends Person {  
  private department: string;  
  
  constructor(name: string, department: string) {  
    super(name);  
    this.department = department;  
  }  
  
  public getElevatorPitch() {  
    return `Hello, my name is ${this.name} and I work in ${this.department}.`;  
  }  
}  
  
let howard = new Employee("Howard", "Sales");  
console.log(howard.getElevatorPitch());  
console.log(howard.name);
```

Property 'name' is protected and only accessible within class 'Person' and its subclasses.