CONTENTS

Tutorial Series: How To Code in TypeScript

5/9 How To Use Classes in Type...
6/9 How To Use Decorators in T...

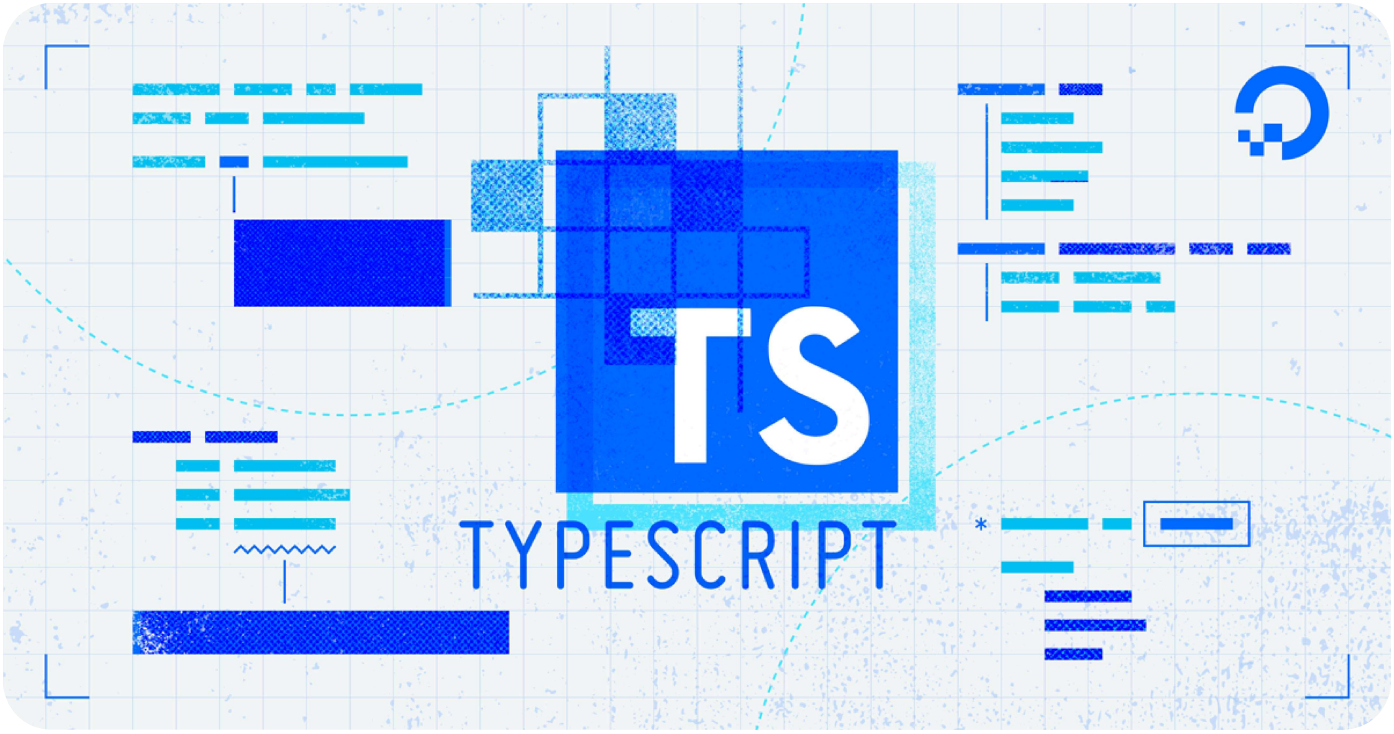
// TUTORIAL //

How To Use Classes in TypeScript

Published on July 9, 2021

Development TypeScript JavaScript

By Jonathan Cardoso
Software Engineer - Lindy.ai



The author selected the COVID-19 Relief Fund to receive a donation as part of the Write for DOnations program.

Introduction

Classes are a common abstraction used in object-oriented programming (OOP) languages to describe data structures known as objects. These objects may contain an initial state and implement behaviors bound to that particular object instance. In 2015, ECMAScript 6 introduced a new syntax to JavaScript to create classes that internally uses the prototype features of the language. TypeScript has full support for that syntax and also adds features on top of it, like member visibility, abstract classes, generic classes, arrow function methods, and a few others.

This tutorial will go through the syntax used to create classes, the different features available, and how classes are treated in TypeScript during the compile-time type-check. It will lead you through examples with different code samples, which you can follow along with in your own TypeScript environment.

Prerequisites

To follow this tutorial:

- A environment in which you can execute TypeScript programs to follow along with the examples. To set this up on your local machine, you will need the following:

  - Both Node and npm (or yarn) installed in order to run a development environment that handles TypeScript-related packages. This tutorial was tested with Node.js version 14.3.0 and npm version 6.14.5. To install on macOS or Ubuntu 18.04, follow the steps in How to Install Node.js and Create a Local Development Environment on macOS or the **Installing Using a PPA** section of How To Install Node.js on Ubuntu 18.04. This also works if you are using the Windows Subsystem for Linux (WSL).
  - Additionally, you will need the TypeScript Compiler (`tsc`) installed on your machine. To do this, refer to the official TypeScript website.
- If you do not wish to create a TypeScript environment on your local machine, you can use the official TypeScript Playground to follow along.
- You will need sufficient knowledge of JavaScript, especially ES6+ syntax, such as destructuring, rest operators, and imports/exports. If you need more information on these topics, reading our How To Code in JavaScript series is recommended.
- This tutorial will reference aspects of text editors that support TypeScript and show in-line errors. This is not necessary to use TypeScript but does take more advantage of TypeScript features. To gain the benefit of these, you can use a text editor like Visual Studio Code, which has full support for TypeScript out of the box. You can also try out these benefits in the TypeScript Playground.

All examples shown in this tutorial were created using TypeScript version 4.3.2.

Creating Classes in TypeScript

In this section, you will run through examples of the syntax used to create classes in TypeScript. While you will cover some of the fundamental aspects of creating classes with TypeScript, the syntax is mostly the same used to create classes with JavaScript. Because of this, this tutorial will focus on some of the distinguishing features available in TypeScript.

You can create a class declaration by using the `class` keyword, followed by the class name and then a `{}` pair block, as shown in the following code:

```
class Person {

}
```

Copy

This snippet creates a new class named `Person`. You can then create a new *instance* of the `Person` class by using the `new` keyword followed by the name of your class and then an empty parameter list (which may be omitted), as shown in the following highlighted code:

```
class Person {

}

const personInstance = new Person();
```

Copy

You can think of the class itself as a blueprint for creating objects with the given shape, while instances are the objects themselves, created from this blueprint.

When working with classes, most of the time you will need to create a `constructor` function. A `constructor` is a method that runs every time a new instance of the class is created. This can be used to initialize values in the class.

Introduce a constructor to your `Person` class:

```
class Person {
  constructor() {
    console.log("Constructor called");
  }
}

const personInstance = new Person();
```

Copy

This constructor will log `Constructor called` to the console when `personInstance` is created.

Constructors are similar to normal functions in the way that they accept parameters. Those parameters are passed to the constructor when you create a new instance of your class. Currently, you are not passing any parameters to the constructor, as shown by the empty parameter list `()` when creating the instance of your class.

Next, introduce a new parameter called `name` of type `string`:

```
class Person {
  constructor( name: string ) {
    console.log(`Constructor called with name=${name}`);
  }
}

const personInstance = new Person( "Jane" );
```

Copy

In the highlighted code, you added a parameter called `name` of type `string` to your class constructor. Then, when creating a new instance of the `Person` class, you are also setting the value of that parameter, in this case to the string `"Jane"`. Finally, you changed the `console.log` to print the argument to the screen.

If you were to run this code, you would receive the following output in the terminal:

```
Output
Constructor called with name=Jane
```

The parameter in the constructor is not optional here. This means that when you instantiate the class, you must pass the `name` parameter to the constructor. If you do not pass the `name` parameter to the constructor, like in the following example:

```
const unknownPerson = new Person;
```

Copy

The TypeScript Compiler will give the error `2554`:

```
Output
Expected 1 arguments, but got 0. (2554)
filename.ts(4, 15): An argument for 'name' was not provided.
```

Now that you have declared a class in TypeScript, you will move on to manipulating those classes by adding properties.

Adding Class Properties

One of the most useful aspects of classes is their ability to hold data that is internal to each instance created from the class. This is done using *properties*.

TypeScript has a few safety checks that differentiate this process from JavaScript classes, including a requirement to initialize properties to avoid them being `undefined`. In this section, you will add new properties to your class to illustrate these safety checks.

With TypeScript, you usually have to declare the property first in the body of the class and give it a type. For example, add a `name` property to your `Person` class:

```
class Person {
  name: string;

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

Copy

In this example, you declare the property `name` with type `string` in addition to setting the property in the `constructor`.

Note: In TypeScript, you can also declare the *visibility* of properties in a class to determine where the data can be accessed. In the `name: string` declaration, the visibility is not declared, which means that the property uses the default `public` status that is accessible anywhere. If you wanted to control the visibility explicitly, you would put declare this with the property. This will be covered more in depth later in the tutorial.

You are also able to give a default value to a property. As an example, add a new property called `instantiatedAt` that will be set to the time the class instance was instantiated:

```
class Person {
  name: string;
  instantiatedAt = new Date();

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

Copy

This uses the `Date` object to set an initial date for the creation of the instance. This code works because the code for the default value is executed when the class constructor is called, which would be equivalent to setting the value on the constructor, as shown in the following:

```
class Person {
  name: string;
  instantiatedAt: Date;

  constructor(name: string) {
    this.name = name;
    this.instantiatedAt = new Date();
```

Copy

```
  }
}
```

By declaring the default value in the body of the class, you do not need to set the value in the constructor.

Note that if you set a type for a property in a class, you must also initialize that property to a value of that type. To illustrate this, declare a class property but do not provide an initializer to it, like in the following code:

```
class Person {
  name: string;
  instantiatedAt: Date;

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

Copy

`instantiatedAt` is assigned a type of `Date`, so must always be a `Date` object. But since there is no initialization, the property becomes `undefined` when the class is instantiated. Because of this, the TypeScript Compiler is going to show the error `2564`:

```
Output
Property 'instantiatedAt' has no initializer and is not definitely assigned in the const
```

This is an additional TypeScript safety check to ensure that the correct properties are present upon class instantiation.

TypeScript also has a shortcut for writing properties that have the same name as the parameters passed to the constructor. This shortcut is called *parameter properties*.

In the previous example, you set the `name` property to the value of the `name` parameter passed to the class constructor. This may become tiresome to write if you add more fields to your class. For example, add a new field called `age` of type `number` to your `Person` class and also add it to the constructor:

```
class Person {
  name: string;
  age: number;
```

Copy

```
  instantiatedAt = new Date();

  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
}
```

While this works, TypeScript can reduce such boilerplate code with parameter properties, or properties set in the parameters for the constructor:

```
class Person {
  instantiatedAt = new Date();

  constructor(
    public name: string,
    public age: number
  ) {}
}
```

Copy

In this snippet, you removed the `name` and `age` property declarations from the class body and moved them to be inside the parameters list of the constructor. When you do that, you are telling TypeScript that those constructor parameters are also properties of that class. This way you do not need to set the property of the class to the value of the parameter received in the constructor, as you did before.

Note: Notice the visibility modifier `public` has been explicitly stated in the code. This modifier must be included when setting parameter properties, and will not automatically default to `public` visibility.

If you take a look at the compiled JavaScript emitted by the TypeScript Compiler, this code compiles to the following JavaScript code:

```
"use strict";
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
    this.instantiatedAt = new Date();
  }
}
```

Copy

This is the same JavaScript code that the original example compiles to.

Now that you have tried out setting properties on TypeScript classes, you can move on to extending classes into new classes with class inheritance.

Class Inheritance in TypeScript

TypeScript offers the full capability of JavaScript's class inheritance, with two main additions: *interfaces* and *abstract classes*. An interface is a structure that describes and enforces the shape of a class or an object, like providing type-checking for more complex pieces of data. You can implement an interface in a class to make sure that it has a specific public shape. Abstract classes are classes that serve as the basis for other classes, but cannot be instantiated themselves. Both of these are implemented via class inheritance.

In this section, you will run through some examples of how interfaces and abstract classes can be used to build and create type checks for classes.

Implementing Interfaces

Interfaces are useful to specify a set of behaviors that all implementations of that interface must possess. Interfaces are created by using the `interface` keyword followed by the name of the interface, and then the interface body. As an example, create a `Logger` interface that could be used to log important data about how your program is running:

```
interface Logger {}
```

Copy

Next, add four methods to your interface:

```
interface Logger {
  debug(message: string, metadata?: Record<string, unknown>): void;
  info(message: string, metadata?: Record<string, unknown>): void;
  warning(message: string, metadata?: Record<string, unknown>): void;
  error(message: string, metadata?: Record<string, unknown>): void;
}
```

Copy

As shown in this code block, when creating the methods in your interface, you do not add any implementation to them, just their type information. In this case, you have four methods: `debug`, `info`, `warning`, and `error`. All of them share the same type signature: They receive parameters, a `message` of type `string` and an optional `metadata` parameter of type `Record<string, unknown>`. They all return the `void` type.

All classes implementing this interface must have the corresponding parameters and return types for each of these methods. Implement the interface in a class named `ConsoleLogger`, which logs all messages using `console` methods:

```
class ConsoleLogger implements Logger {
  debug(message: string, metadata?: Record<string, unknown>) {
    console.info(`[DEBUG] ${message}`, metadata);
  }
  info(message: string, metadata?: Record<string, unknown>) {
    console.info(message, metadata);
  }
  warning(message: string, metadata?: Record<string, unknown>) {
    console.warn(message, metadata);
  }
  error(message: string, metadata?: Record<string, unknown>) {
    console.error(message, metadata);
  }
}
```

Copy

Notice that when creating your interface, you are using a new keyword called `implements` to specify the list of interfaces your class implements. You can implement multiple interfaces by adding them as a comma-separated list of interface identifiers after the `implements` keyword. For example, if you had another interface called `Clearable`:

```
interface Clearable {
  clear(): void;
}
```

Copy

You could implement it in the `ConsoleLogger` class by adding the following highlighted code:

```
class ConsoleLogger implements Logger , Clearable {
  clear() {
    console.clear();
  }
  debug(message: string, metadata?: Record<string, unknown>) {
    console.info(`[DEBUG] ${message}`, metadata);
  }
  info(message: string, metadata?: Record<string, unknown>) {
    console.info(message, metadata);
  }
  w(message: string, metadata?: Record<string, unknown>) {
    console.warn(message, metadata);
  }
```

Copy

```
  error(message: string, metadata?: Record<string, unknown>) {
    console.error(message, metadata);
  }
}
```

Notice that you also have to add the `clear` method to make sure the class adheres to the new interface.

If you did not provide the implementation for one of the members required by any of the interfaces, like the `debug` method from the `Logger` interface, the TypeScript compiler would give you the error `2420`:

```
Output
Class 'ConsoleLogger' incorrectly implements interface 'Logger'.
  Property 'debug' is missing in type 'ConsoleLogger' but required in type 'Logger'. (24
```

The TypeScript Compiler would also show an error if your implementation did not match the one expected by the interface you are implementing. For example, if you changed the type of the `message` parameter in the `debug` method from `string` to `number`, you would receive error `2416`:

```
Output
Property 'debug' in type 'ConsoleLogger' is not assignable to the same property in base
  Type '(message: number, metadata?: Record<string, unknown> | undefined) => void' is no
    Types of parameters 'message' and 'message' are incompatible.
      Type 'string' is not assignable to type 'number'. (2416)
```

Building on Abstract Classes

Abstract classes are similar to normal classes, with two major differences: They cannot be directly instantiated and they may contain *abstract members*. Abstract members are members that must be implemented in inheriting classes. They do not have an implementation in the abstract class itself. This is useful because you can have some common functionality in the base abstract class, and more specific implementations in the inheriting classes. When you mark a class as abstract, you are saying that this class has missing functionality that should be implemented in inheriting classes.

To create an abstract class, you add the `abstract` keyword before the `class` keyword, like in the highlighted code:

```
abstract class AbstractClassName {

}
```

Copy

Next, you can create members in your abstract class, some that may have an implementation and others that will not. Ones without implementation are marked as `abstract` and must then be implemented in the classes that extend from your abstract class.

For example, imagine you are working in a Node.js environment and you are creating your own `Stream` implementation. For that, you are going to have an abstract class called `Stream` with two abstract methods, `read` and `write`:

```
declare class Buffer {
  from(array: any[]): Buffer;
  copy(target: Buffer, offset?: number): void;
}

abstract class Stream {

  abstract read(count: number): Buffer;

  abstract write(data: Buffer): void;
}
```

Copy

The `Buffer` object here is a class available in Node.js that is used to store binary data. The `declare class Buffer` statement at the top allows the code to compile in a TypeScript environment without the Node.js type declarations, like TypeScript Playground.

In this example, the `read` method counts bytes from the internal data structure and returns a `Buffer` object, and `write` writes all the contents of the `Buffer` instance to the stream. Both of these methods are abstract, and can only be implemented in classes extended from `Stream`.

You can then create additional methods that do have an implementation. This way any class extending from your `Stream` abstract class would receive those methods automatically. One such example would be a `copy` method:

```
declare class Buffer {
  from(array: any[]): Buffer;
  copy(target: Buffer, offset?: number): void;
}
```

Copy

```
abstract class Stream {

  abstract read(count: number): Buffer;

  abstract write(data: Buffer): void;

  copy(count: number, targetBuffer: Buffer, targetBufferOffset: number) {
    const data = this.read(count);
    data.copy(targetBuffer, targetBufferOffset);
  }
}
```

This `copy` method copies the result from reading the bytes from the stream to the `targetBuffer`, starting at `targetBufferOffset`.

If you then create an implementation for your `Stream` abstract class, like a `FileStream` class, the `copy` method would be readily available, without having to duplicate it in your `FileStream` class:

```
declare class Buffer {
  from(array: any[]): Buffer;
  copy(target: Buffer, offset?: number): void;
}

abstract class Stream {

  abstract read(count: number): Buffer;

  abstract write(data: Buffer): void;

  copy(count: number, targetBuffer: Buffer, targetBufferOffset: number) {
    const data = this.read(count);
    data.copy(targetBuffer, targetBufferOffset);
  }
}

class FileStream extends Stream {
  read(count: number): Buffer {
    // implementation here
    return new Buffer();
  }

  write(data: Buffer) {
    // implementation here
  }
```

Copy

```
}

const fileStream = new FileStream();
```

In this example, the `fileStream` instance automatically has the `copy` method available on it. The `FileStream` class also had to implement a `read` and a `write` method explicitly to adhere to the `Stream` abstract class.

If you had forgotten to implement one of the abstract members of the abstract class you are extending from, like not adding the `write` implementation in your `FileStream` class, the TypeScript compiler would give error `2515`:

```
Output
Non-abstract class 'FileStream' does not implement inherited abstract member 'write' fro
```

The TypeScript compiler would also display an error if you implemented any of the members incorrectly, like changing the type of the first parameter of the `write` method to be of type `string` instead of `Buffer`:

```
Output
Property 'write' in type 'FileStream' is not assignable to the same property in base typ
  Type '(data: string) => void' is not assignable to type '(data: Buffer) => void'.
    Types of parameters 'data' and 'data' are incompatible.
      Type 'Buffer' is not assignable to type 'string'. (2416)
```

With abstract classes and interfaces, you are able to put together more complex type-checking for your classes to ensure that classes extended from base classes inherit the correct functionality. Next, you will run through examples of how method and property visibility work in TypeScript.

Class Members Visibility

TypeScript augments the available JavaScript class syntax by allowing you to specify the visibility of the members of a class. In this case, *visibility* refers to how code outside of an instantiated class can interact with a member inside the class.

Class members in TypeScript may have three possible visibility modifiers: `public`, `protected`, and `private`. `public` members may be accessed outside of the class instance,

where as `private` ones cannot. `protected` occupies a middle ground between the two, where members can be accessed by instances of the class or subclasses based on that class.

In this section, you are going to examine the available visibility modifiers and learn what they mean.

`public`

This is the default visibility of class members in TypeScript. When you do not add the visibility modifier to a class member, it is the same as setting it to `public`. Public class members may be accessed anywhere, without any restrictions.

To illustrate this, return to your `Person` class from earlier:

```
class Person {
  public instantiatedAt = new Date();

  constructor(
    name: string,
    age: number
  ) {}
}
```

Copy

This tutorial mentioned that the two properties `name` and `age` had `public` visibility by default. To declare type visibility explicitly, add the `public` keyword before the properties and a new `public` method to your class called `getBirthYear`, which retrieves the year of birth for the `Person` instance:

```
class Person {
  constructor(
    public name: string,
    public age: number
  ) {}

  public getBirthYear() {
    return new Date().getFullYear() - this.age;
  }
}
```

Copy

You can use the properties and methods in the global space, outside the class instance:

```
class Person {
  constructor(
    public name: string,
    public age: number
  ) {}

  public getBirthYear() {
    return new Date().getFullYear() - this.age;
  }
}

const jon = new Person("Jon", 35);

console.log(jon.name);
console.log(jon.age);
console.log(jon.getBirthYear());
```

Copy

This code would print the following to the console:

```
Output
Jon
35
1986
```

Notice that you can access all the members of your class.

`protected`

Class members with the `protected` visibility are only allowed to be used inside the class they are declared in or in the subclasses of that class.

Take a look at the following `Employee` class and the `FinanceEmployee` class that is based on it:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}
}

class FinanceEmployee extends Employee {
  getFinanceIdentifier() {
    return `fin-${this.identifier}`;
```

Copy

```
  }
}
```

The highlighted code shows the `identifier` property declared with `protected` visibility. The `this.identifier` code tries to access this property from the `FinanceEmployee` subclass. This code would run without error in TypeScript.

If you tried to use that method from a place that is not inside the class itself, or inside a subclass, like in the following example:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}
}

class FinanceEmployee extends Employee {
  getFinanceIdentifier() {
    return `fin-${this.identifier}`;
  }
}

const financeEmployee = new FinanceEmployee('abc-12345');
financeEmployee.identifier;
```

Copy

The TypeScript compiler would give us the error `2445`:

```
Output
Property 'identifier' is protected and only accessible within class 'Employee' and its s
```

This is because the `identifier` property of the new `financeEmployee` instance cannot be retrieved from the global space. Instead, you would have to use the internal method `getFinanceIdentifier` to return a string that included the `identifier` property:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}
}

class FinanceEmployee extends Employee {
```

Copy

```
  getFinanceIdentifier() {
    return `fin-${this.identifier}`;
  }
}

const financeEmployee = new FinanceEmployee('abc-12345');
console.log(financeEmployee.getFinanceIdentifier())
```

This would log the following to the console:

```
Output
fin-abc-12345
```

`private`

Private members are only accessible inside the class that declares them. This means that not even subclasses have access to it.

Using the previous example, turn the `identifier` property in the `Employee` class into a `private` property:

```
class Employee {
  constructor(
    private identifier: string
  ) {}
}

class FinanceEmployee extends Employee {
  getFinanceIdentifier() {
    return `fin-${this.identifier}`;
  }
}
```

Copy

This code will now cause the TypeScript compiler to show the error `2341`:

```
Output
Property 'identifier' is private and only accessible within class 'Employee'. (2341)
```

This is because you are accessing the property `identifier` in the `FinanceEmployee` subclass, and this is not allowed, as the `identifier` property was declared in the `Employee` class and has its visibility set to `private`.

Remember that TypeScript is compiled to raw JavaScript that by itself does not have any way to specify the visibility of the members of a class. As such, TypeScript has no protection against such usage during runtime. This is a safety check done by the TypeScript compiler only during compilation.

Now that you've tried out visibility modifiers, you can move on to arrow functions as methods in TypeScript classes.

Class Methods as Arrow Functions

In JavaScript, the `this` value that represents a function's context can change depending on how a function is called. This variability can sometimes be confusing in complex pieces of code. When working with TypeScript, you can use a special syntax when creating class methods to avoid `this` being bound to something else other than the class instance. In this section, you will try out this syntax.

Using your `Employee` class, introduce a new method used only to retrieve the employee identifier:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}

  getIdentifier() {
    return this.identifier;
  }
}
```

Copy

This works pretty well if you call the method directly:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}

  getIdentifier() {
    return this.identifier;
  }
}

const employee = new Employee("abc-123");
```

Copy

```
console.log(employee.getIdentifier());
```

This would print the following to the console's output:

```
Output
abc-123
```

However, if you stored the `getIdentifier` instance method somewhere for it to be called later, like in the following code:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}

  getIdentifier() {
    return this.identifier;
  }
}

const employee = new Employee("abc-123");

const obj = {
  getId: employee.getIdentifier
}

console.log(obj.getId());
```

Copy

The value would be inaccessible:

```
Output
undefined
```

This happens because when you call `obj.getId()`, the `this` inside `employee.getIdentifier` is now bound to the `obj` object, and not to the `Employee` instance.

You can avoid this by changing your `getIdentifier` to be an arrow function. Check the highlighted change in the following code:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}

  getIdentifier = () => {
    return this.identifier;
  }
}
...
```

Copy

If you now try to call `obj.getId()` like you did before, the console correctly shows:

```
Output
abc-123
```

This demonstrates how TypeScript allows you to use arrow functions as direct values of class methods. In the next section, you will learn how to enforce classes with TypeScript's type-checking.

Using Classes as Types

So far this tutorial has covered how to create classes and use them directly. In this section, you will use classes as types when working with TypeScript.

Classes are both a type and a value in TypeScript, and as such, can be used both ways. To use a class as a type, you use the class name in any place that TypeScript expects a type. For example, given the `Employee` class you created previously:

```
class Employee {
  constructor(
    public identifier: string
  ) {}
}
```

Copy

Imagine you wanted to create a function that prints the identifier of any employee. You could create such a function like this:

```
class Employee {
  constructor(
    public identifier: string
  ) {}
}

function printEmployeeIdentifier(employee: Employee) {
  console.log(employee.identifier);
}
```

Copy

Notice that you are setting the `employee` parameter to be of type `Employee`, which is the exact name of your class.

Classes in TypeScript are compared against other types, including other classes, just like other types are compared in TypeScript: structurally. This means that if you had two different classes that both had the same shape (that is, the same set of members with the same visibility), both can be used interchangeably in places that would expect only one of them.

To illustrate this, imagine you have another class in your application called `Warehouse`:

```
class Warehouse {
  constructor(
    public identifier: string
  ) {}
}
```

Copy

It has the same shape as `Employee`. If you tried to pass an instance of it to `printEmployeeIdentifier`:

```
class Employee {
  constructor(
    public identifier: string
  ) {}
}

class Warehouse {
  constructor(
    public identifier: string
  ) {}
}

function printEmployeeIdentifier(employee: Employee) {
```

Copy

```
  console.log(employee.identifier);
}

const warehouse = new Warehouse("abc");

printEmployeeIdentifier(warehouse);
```

The TypeScript compiler would not complain. You could even use just a normal object instead of the instance of a class. As this may result in a behavior that is not expected by a programmer that is just starting with TypeScript, it is important to keep an eye on these scenarios.

With the basics of using a class as a type out of the way, you can now learn how to check for specific classes, rather than just the shape.

The Type of `this`

Sometimes you will need to reference the type of the current class inside some methods in the class itself. In this section, you will find out how to use `this` to accomplish this.

Imagine you had to add a new method to your `Employee` class called `isSameEmployeeAs`, which would be responsible for checking if another employee instance references the same employee as the current one. One way you could do this would be like the following:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}

  getIdentifier() {
    return this.identifier;
  }

  isSameEmployeeAs(employee: Employee) {
    return this.identifier === employee.identifier;
  }
}
```

Copy

This test will work to compare the `identifier` property of all classes derived from `Employee`. But imagine a scenario in which you do not want specific subclasses of `Employee` to be compared at all. In this case, instead of receiving the boolean value of the comparison, you would want TypeScript to report an error when two different subclasses are compared.

For example, create two new subclasses for employees in the finance and marketing departments:

```
...
class FinanceEmployee extends Employee {
  specialFieldToFinanceEmployee = '';
}

class MarketingEmployee extends Employee {
  specialFieldToMarketingEmployee = '';
}

const finance = new FinanceEmployee("fin-123");
const marketing = new MarketingEmployee("mkt-123");

marketing.isSameEmployeeAs(finance);
```

Here you derive two classes from the `Employee` base class: `FinanceEmployee` and `MarketingEmployee`. Each one has different new fields. You are then creating one instance of each one, and checking if the `marketing` employee is the same as the `finance` employee. Given this scenario, TypeScript should report an error, since subclasses should not be compared at all. This does not happen because you used `Employee` as the type of the `employee` parameter in your `isSameEmployeeAs` method, and all classes derived from `Employee` will pass the type-checking.

To improve this code, you could use a special type available inside classes, which is the `this` type. This type is dynamically set to the type of the current class. This way, when this method is called in a derived class, `this` is set to the type of the derived class.

Change your code to use `this` instead:

```
class Employee {
  constructor(
    protected identifier: string
  ) {}

  getIdentifier() {
    return this.identifier;
  }

  isSameEmployeeAs(employee: this) {
    return this.identifier === employee.identifier;
  }
}
```

```
class FinanceEmployee extends Employee {
  specialFieldToFinanceEmployee = '';
}

class MarketingEmployee extends Employee {
  specialFieldToMarketingEmployee = '';
}

const finance = new FinanceEmployee("fin-123");
const marketing = new MarketingEmployee("mkt-123");

marketing.isSameEmployeeAs(finance);
```

When compiling this code, the TypeScript compiler will now show the error `2345`:

```
Output
Argument of type 'FinanceEmployee' is not assignable to parameter of type 'MarketingEmpl
  Property 'specialFieldToMarketingEmployee' is missing in type 'FinanceEmployee' but re
```

With the `this` keyword, you can change typing dynamically in different class contexts. Next, you will use typing for passing in a class itself, rather than an instance of a class.

Using Construct Signatures

There are times when a programmer needs to create a function that takes a class directly, instead of an instance. For that, you need to use a special type with a construct signature. In this section, you will go through how to create such types.

One particular scenario in which you may need to pass in a class itself is a *class factory*, or a function that generates new instances of classes that are passed in as arguments. Imagine you want to create a function that takes a class based on `Employee`, creates a new instance with an incremented identifier, and prints the identifier to the console. One may try to create this like the following:

```
class Employee {
  constructor(
    public identifier: string
  )
}
```

Copy

```
let identifier = 0;
function createEmployee(ctor: Employee) {
  const employee = new ctor(`test-${identifier++}`);
  console.log(employee.identifier);
}
```

In this snippet, you create the `Employee` class, initialize the `identifier`, and create a function that instantiates a class based on a constructor parameter `ctor` that has the shape of `Employee`. But if you tried to compile this code, the TypeScript compiler would give the error `2351`:

```
Output
This expression is not constructable.
  Type 'Employee' has no construct signatures. (2351)
```

This happens because when you use the name of your class as the type for `ctor`, the type is only valid for instances of the class. To get the type of the class constructor itself, you have to use `typeof ClassName`. Check the following highlighted code with the change:

```
class Employee {
  constructor(
    public identifier: string
  ) {}
}

let identifier = 0;
function createEmployee(ctor: typeof Employee) {
  const employee = new ctor(`test-${identifier++}`);
  console.log(employee.identifier);
}
```

Copy

Now your code will compile successfully. But there is still a pending issue: Since class factories build instances of new classes built from a base class, using `abstract` classes could improve the workflow. However, this will not work initially.

To try this out, turn the `Employee` class into an `abstract` class:

```
abstract class Employee {
  constructor(
    public identifier: string
  ) {}
}
```

Copy

```
let identifier = 0;
function createEmployee(ctor: typeof Employee) {
  const employee = new ctor(`test-${identifier++}`);
  console.log(employee.identifier);
}
```

The TypeScript compiler will now give the error `2511`:

```
Output
Cannot create an instance of an abstract class. (2511)
```

This error shows that you cannot create an instance from the `Employee` class, since it is `abstract`. But you may want to use such a function to create different kinds of employees that extend from your `Employee` abstract class, like such:

```
abstract class Employee {
  constructor(
    public identifier: string
  ) {}
}

class FinanceEmployee extends Employee {}

class MarketingEmployee extends Employee {}

let identifier = 0;
function createEmployee(ctor: typeof Employee) {
  const employee = new ctor(`test-${identifier++}`);
  console.log(employee.identifier);
}

createEmployee(FinanceEmployee);
createEmployee(MarketingEmployee);
```

To make your code work for this scenario, you have to use a type with a constructor signature. You can do this by using the `new` keyword, followed by a syntax similar to that of an arrow function, where the parameter list contains the parameters expected by the constructor and the return type is the class instance this constructor returns.

Highlighted in the following code is the change introducing the type with a constructor signature to your `createEmployee` function:

```
abstract class Employee {
  constructor(
    public identifier: string
  ) {}
}

class FinanceEmployee extends Employee {}

class MarketingEmployee extends Employee {}

let identifier = 0;
function createEmployee(ctor: new (identifier: string) => Employee) {
  const employee = new ctor(`test-${identifier++}`);
  console.log(employee.identifier);
}

createEmployee(FinanceEmployee);
createEmployee(MarketingEmployee);
```

Copy

The TypeScript compiler now will correctly compile your code.

Conclusion

Classes in TypeScript are even more powerful than they are in JavaScript because you have access to the type system, extra syntax like arrow function methods, and completely new features like member visibility and abstract classes. This offers a way for you to deliver code that is type-safe, more reliable, and that better represents the business model of your application.

For more tutorials on TypeScript, check out our How To Code in TypeScript series page.

Next in series: How To Use Decorators in TypeScript →

Tutorial Series: How To Code in TypeScript

TypeScript is an extension of the JavaScript language that uses JavaScript's runtime with a compile-time type checker. This combination allows developers to use the full JavaScript ecosystem and language features, while also adding optional static type-checking, enum data types, classes, and interfaces.

This series will show you the syntax you need to get started with TypeScript, allowing you to leverage its typing system to make scalable, enterprise-grade code.

Subscribe

Development TypeScript JavaScript

Browse Series: 9 articles

Expand to view all

About the authors

Jonathan Cardoso Author

Software Engineer - Lindy.ai

Still looking for an answer? Ask a question

Search for more help

Was this helpful? Yes No

Comments

Leave a comment

Leave a comment...

This textbox defaults to using **Markdown** to format your answer.

You can type **!ref** in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

All tutorials →

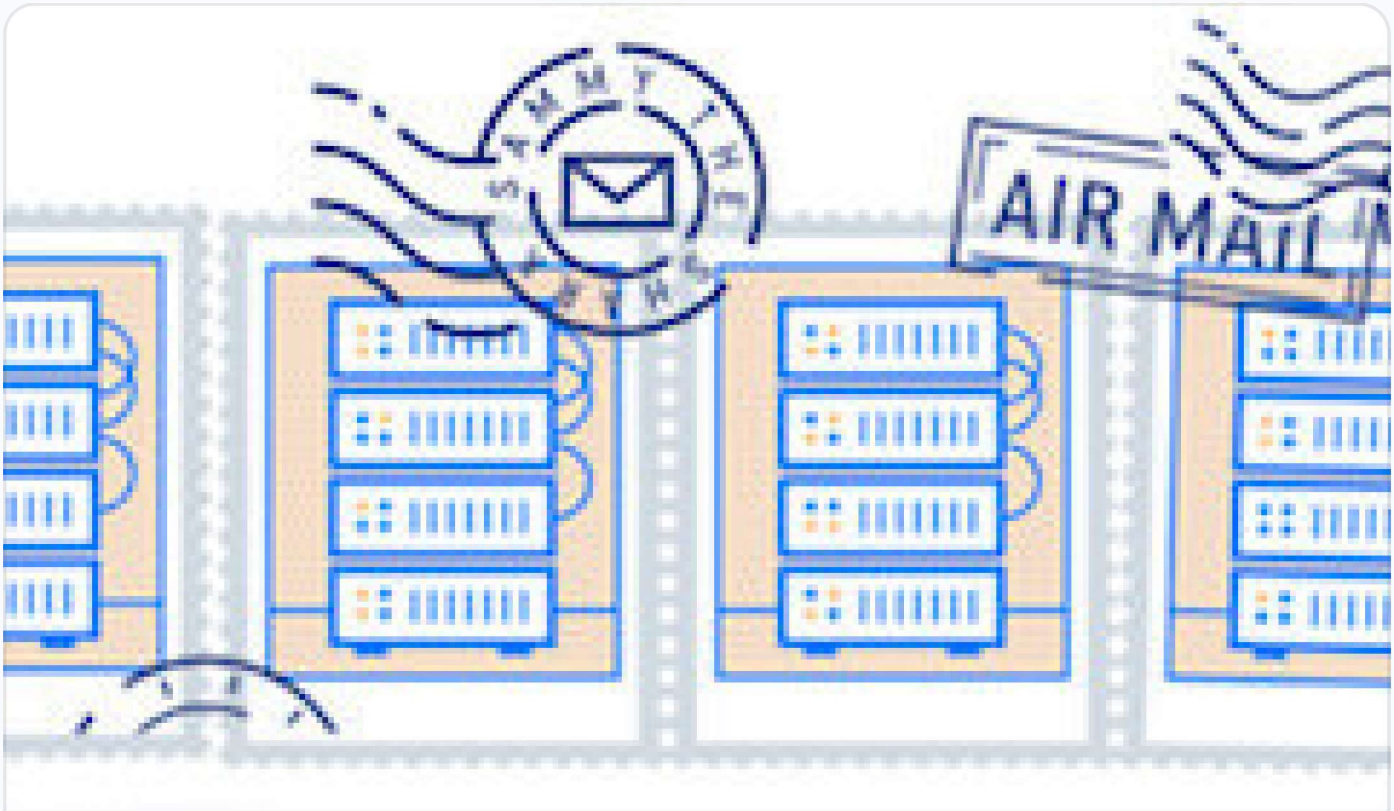
Talk to an expert →

Congratulations on unlocking the whale ambience easter egg! Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you read.

Thank you to the Glacier Bay National Park & Preserve and Merrick079 for the sounds behind this easter egg.

Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the Whale and Dolphin Conservation.

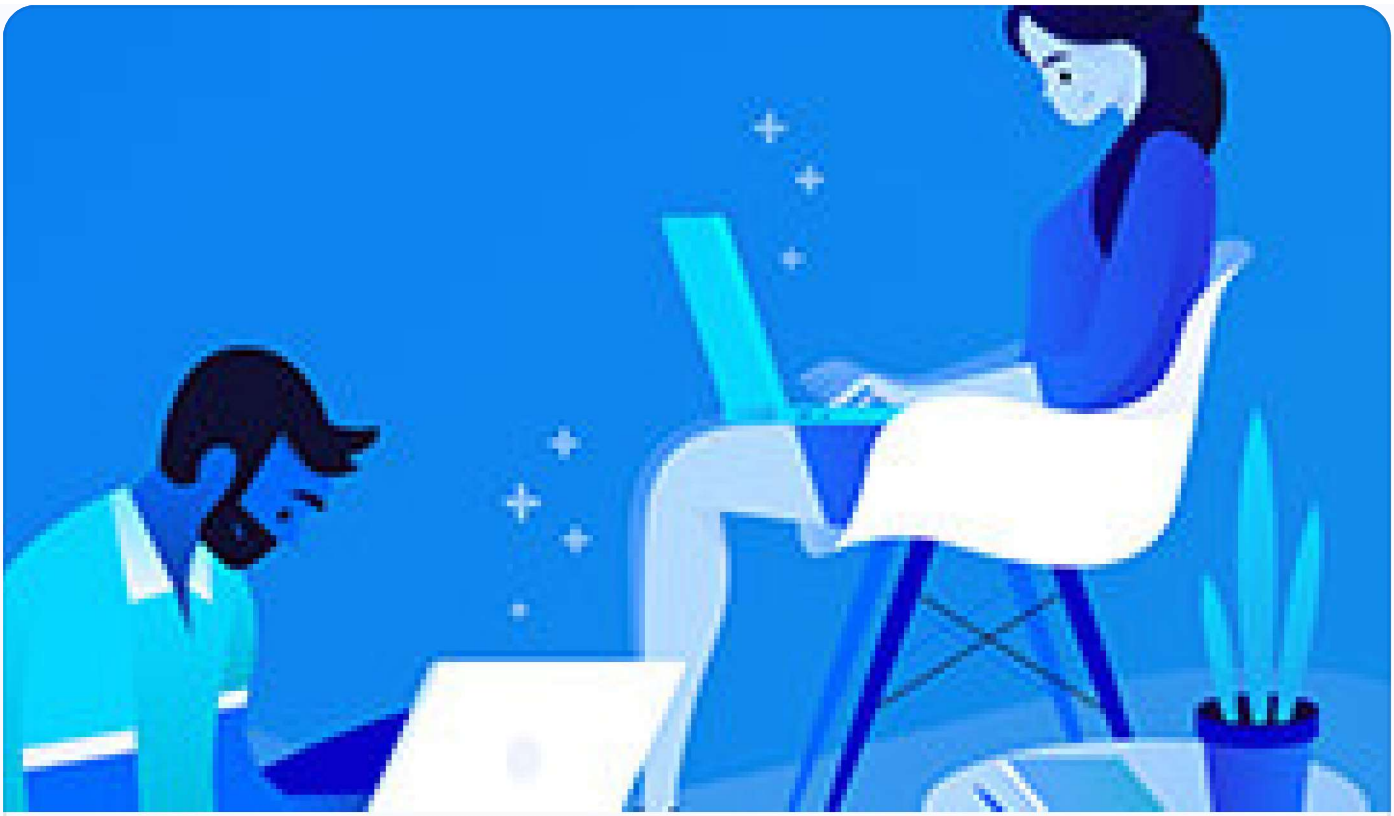
Reset easter egg to be discovered again / Permanently dismiss and hide easter egg



Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

Sign up →



Community

Solutions

Contact