**Clases**

[**Introducción**](https://www.typescriptlang.org/docs/handbook/classes.html#introduction)

JavaScript tradicional utiliza funciones y herencia basada en prototipos para construir componentes reutilizables, pero esto puede resultar un poco incómodo para los programadores más cómodos con un enfoque orientado a objetos, donde las clases heredan la funcionalidad y los objetos se crean a partir de estas clases. A partir de ECMAScript 2015, también conocido como ECMAScript 6, los programadores de JavaScript podrán construir sus aplicaciones utilizando este enfoque basado en clases orientado a objetos. En TypeScript, permitimos que los desarrolladores usen estas técnicas ahora y las compilen en JavaScript que funcione en todos los principales navegadores y plataformas, sin tener que esperar a la próxima versión de JavaScript.

[**Clases**](https://www.typescriptlang.org/docs/handbook/classes.html#classes)

Echemos un vistazo a un ejemplo simple basado en clases:

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

greet() {

**return** "Hello, " + **this**.greeting;

}

}

**let** greeter = **new** Greeter("world");

La sintaxis debería resultarle familiar si ha usado C # o Java anteriormente. Declaramos una nueva clase Greeter. Esta clase tiene tres miembros: una propiedad llamada greeting, un constructor y un método greet.

Notarás que en la clase cuando nos referimos a uno de los miembros de la clase que anteponemos this.. Esto denota que es un acceso de miembro.

En la última línea construimos una instancia de la Greeterclase usando new. Esto llama al constructor que definimos anteriormente, creando un nuevo objeto con la Greeterforma y ejecutando el constructor para inicializarlo.

[**Herencia**](https://www.typescriptlang.org/docs/handbook/classes.html#inheritance)

En TypeScript, podemos usar patrones comunes orientados a objetos. Uno de los patrones más fundamentales en la programación basada en clases es poder extender las clases existentes para crear otras nuevas usando la herencia.

Echemos un vistazo a un ejemplo:

**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();

Este ejemplo muestra la característica de herencia más básica: las clases heredan propiedades y métodos de las clases base. Aquí, Doghay una clase *derivada* que deriva de la clase Animal *base* usando la extendspalabra clave. Las clases derivadas a menudo se denominan *subclases* , y las clases base a menudo se denominan *superclases* .

Debido a que Dogextiende la funcionalidad desde Animal, pudimos crear una instancia de Dogque podría ambos bark()y move().

[**Modificadores públicos, privados y protegidos.**](https://www.typescriptlang.org/docs/handbook/classes.html#public-private-and-protected-modifiers)

[**Público por defecto**](https://www.typescriptlang.org/docs/handbook/classes.html#public-by-default)

En nuestros ejemplos, hemos podido acceder libremente a los miembros que declaramos en todos nuestros programas. Si está familiarizado con las clases en otros idiomas, puede haber notado en los ejemplos anteriores que no hemos tenido que usar la palabrapublic para lograr esto; por ejemplo, C # requiere que cada miembro esté explícitamente etiquetado publiccomo visible. En TypeScript, cada miembro es publicpor defecto.

Aún puede marcar un miembro publicexplícitamente. Podríamos haber escrito la Animalclase de la sección anterior de la siguiente manera:

**class** Animal {

**public** name: string;

**public** **constructor**(theName: string) { **this**.name = theName; }

**public** move(distanceInMeters: number) {

console.log(`${this.name} moved ${distanceInMeters}m.`);

}

}

[**Comprensión private**](https://www.typescriptlang.org/docs/handbook/classes.html#understanding-private)

Cuando se marca un miembro private, no se puede acceder desde fuera de su clase que lo contiene. Por ejemplo:

**class** Animal {

**private** name: string;

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

}

**new** Animal("Cat").name; // Error: 'name' is private;

TypeScript es un sistema de tipo estructural. Cuando comparamos dos tipos diferentes, independientemente de su procedencia, si los tipos de todos los miembros son compatibles, entonces decimos que los tipos mismos son compatibles.

Sin embargo, al comparar tipos que tienen privatey protectedmiembros, tratamos estos tipos de manera diferente. Para que dos tipos se consideren compatibles, si uno de ellos tiene un privatemiembro, el otro debe tener un privatemiembro que se originó en la misma declaración. Lo mismo se aplica a los protectedmiembros.

Veamos un ejemplo para ver mejor cómo se desarrolla esto en la práctica:

**class** Animal {

**private** name: string;

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

}

**class** Rhino extends Animal {

**constructor**() { **super**("Rhino"); }

}

**class** Employee {

**private** name: string;

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

}

**let** animal = **new** Animal("Goat");

**let** rhino = **new** Rhino();

**let** employee = **new** Employee("Bob");

animal = rhino;

animal = employee; // Error: 'Animal' and 'Employee' are not compatible

En este ejemplo, tenemos una Animaly una Rhino, con Rhinoser una subclase de Animal. También tenemos una nueva clase Employeeque se ve idéntica Animalen términos de forma. Creamos algunas instancias de estas clases y luego tratamos de asignarlas entre sí para ver qué sucederá. Porque Animaly Rhinocomparten el privatelado de su forma desde la misma declaración de private name: stringin Animal, son compatibles. Sin embargo, este no es el caso Employee. Cuando intentamos asignar de a Employeea Animal, obtenemos un error de que estos tipos no son compatibles. Aunque Employeetambién tiene un privatemiembro llamado name, no es el que declaramos enAnimal .

[**Comprensiónprotected**](https://www.typescriptlang.org/docs/handbook/classes.html#understanding-protected)

El protectedmodificador actúa de manera muy similar al privatemodificador con la excepción de que los miembros declarados protectedtambién pueden accederse dentro de las clases derivadas. Por ejemplo,

**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); // error

Tenga en cuenta que si bien no podemos usarlo namedesde fuera Person, aún podemos usarlo desde un método de instancia de Employeeporque Employeederiva dePerson .

Un constructor también puede estar marcado protected. Esto significa que la clase no se puede instanciar fuera de su clase que contiene, sino que se puede extender. Por ejemplo,

**class** Person {

**protected** name: string;

**protected** **constructor**(theName: string) { **this**.name = theName; }

}

// Employee can extend Person

**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");

**let** john = **new** Person("John"); // Error: The 'Person' constructor is protected