



Esta copia del manual de TypeScript se
creó el viernes 24 de junio de 2022 con la
confirmación e538f6 con TypeScript 4.7.

Tabla de contenido

El manual de TypeScript	Su primer paso para aprender TypeScript
Los basicos	Paso uno para aprender TypeScript: los tipos básicos.
Tipos cotidianos	Los primitivos del lenguaje.
Estrechamiento	Comprender cómo TypeScript usa el conocimiento de JavaScript para reducir la cantidad de sintaxis de tipos en sus proyectos.
Más sobre funciones	Obtenga información sobre cómo funcionan las funciones en TypeScript.
Tipos de objetos	Cómo describe TypeScript las formas de JavaScript objetos.
Crear tipos a partir de tipos	Una descripción general de las formas en que puede crear más tipos de tipos existentes.
Genéricos	Tipos que toman parámetros
Operador de tipo Keyof	Uso del operador keyof en contextos de tipo.
Tipo de operador de tipo	Uso del operador typeof en contextos de tipo.
Tipos de acceso indexado	Uso de la sintaxis Type['a'] para acceder a un subconjunto de un tipo.
Tipos condicionales	Cree tipos que actúen como declaraciones if en el tipo sistema.
Tipos asignados	Generación de tipos mediante la reutilización de un tipo existente.
Tipos de literales de plantilla	Generación de tipos de mapeo que cambian propiedades a través de cadenas de literales de plantilla.
Clases	Cómo funcionan las clases en TypeScript
Módulos	Cómo maneja JavaScript la comunicación entre archivos límites.

El manual de mecanografiado

Acerca de este Manual

Más de 20 años después de su introducción a la comunidad de programación, JavaScript es ahora uno de los lenguajes multiplataforma más extendidos jamás creados. Comenzando como un pequeño lenguaje de secuencias de comandos para agregar interactividad trivial a las páginas web, JavaScript se ha convertido en un lenguaje de elección para aplicaciones de todos los tamaños, tanto de frontend como de backend. Si bien el tamaño, el alcance y la complejidad de los programas escritos en JavaScript han crecido exponencialmente, la capacidad del lenguaje JavaScript para expresar las relaciones entre diferentes unidades de código no lo ha hecho. Combinado con la semántica de tiempo de ejecución bastante peculiar de JavaScript, este desajuste entre el lenguaje y la complejidad del programa ha hecho que el desarrollo de JavaScript sea una tarea difícil de administrar a escala.

Los tipos de errores más comunes que escriben los programadores se pueden describir como errores de tipo: se utilizó un cierto tipo de valor donde se esperaba un tipo diferente de valor. Esto podría deberse a errores tipográficos simples, una falta de comprensión de la superficie API de una biblioteca, suposiciones incorrectas sobre el comportamiento del tiempo de ejecución u otros errores. El objetivo de TypeScript es ser un verificador de tipos estático para los programas de JavaScript; en otras palabras, una herramienta que se ejecuta antes de que se ejecute el código (estático) y garantiza que los tipos del programa sean correctos (con verificación de tipos).

Si llega a TypeScript sin experiencia en JavaScript, con la intención de que TypeScript sea su primer idioma, le recomendamos que primero comience a leer la documentación en el [tutorial de Microsoft Learn JavaScript](#) o [lea JavaScript en Mozilla Web Docs](#). Si tiene experiencia en otros idiomas, debería poder aprender la sintaxis de JavaScript con bastante rapidez leyendo el manual.

¿Cómo está estructurado este manual?

El manual se divide en dos secciones:

- **el manual**

El Manual de TypeScript pretende ser un documento completo que explica TypeScript a los programadores cotidianos. Puede leer el manual yendo de arriba a abajo en la barra de navegación de la izquierda.

Debe esperar que cada capítulo o página le brinde una sólida comprensión de los conceptos dados. El Manual de TypeScript no es una especificación completa del lenguaje, pero pretende ser una guía completa de todas las funciones y comportamientos del lenguaje.

Un lector que complete el recorrido debe ser capaz de:

- Leer y comprender la sintaxis y los patrones de TypeScript de uso común
- Explicar los efectos de las opciones importantes del compilador.
- Predecir correctamente el comportamiento del sistema de tipos en la mayoría de los casos

En aras de la claridad y la brevedad, el contenido principal del Manual no explorará todos los casos extremos o minucias de las características que se cubren. Puede encontrar más detalles sobre conceptos particulares en los artículos de referencia.

● Archivos de referencia

La sección de referencia debajo del manual en la navegación está diseñada para proporcionar una mejor comprensión de cómo funciona una parte particular de TypeScript. Puede leerlo de arriba a abajo, pero cada sección tiene como objetivo proporcionar una explicación más profunda de un solo concepto, lo que significa que no hay ningún objetivo de continuidad.

Sin objetivos

El Manual también pretende ser un documento conciso que se pueda leer cómodamente en unas pocas horas. Ciertos temas no se cubrirán para acortar las cosas.

Específicamente, el manual no introduce completamente los conceptos básicos básicos de JavaScript, como funciones, clases y cierres. Cuando corresponda, incluiremos enlaces a lecturas previas que puede usar para leer sobre esos conceptos.

El Manual tampoco pretende ser un reemplazo de una especificación de idioma. En algunos casos, los casos límite o las descripciones formales del comportamiento se saltean en favor de explicaciones de alto nivel y más fáciles de entender. En cambio, hay páginas de referencia separadas que describen de manera más precisa y formal muchos aspectos del comportamiento de TypeScript. Las páginas de referencia no están destinadas a lectores que no estén familiarizados con TypeScript, por lo que pueden usar terminología avanzada o temas de referencia sobre los que aún no ha leído.

Finalmente, el manual no cubrirá cómo interactúa TypeScript con otras herramientas, excepto cuando sea necesario.

Temas como cómo configurar TypeScript con webpack, rollup, package, react, babel, closure, lerna, rush, babel, preact, vue, angular, svelte, jquery, yarn o npm están fuera del alcance; puede encontrar estos recursos en otros lugares En la red.

Empezar

Antes de comenzar con [los conceptos básicos](#), recomendamos leer una de las siguientes páginas introductorias. Estas introducciones pretenden resaltar las similitudes y diferencias clave entre

TypeScript y su lenguaje de programación favorito, y aclare conceptos erróneos comunes específicos de esos lenguajes.

- [TypeScript para nuevos programadores](#)
- [TypeScript para programadores de JavaScript](#)
- [TypeScript para programadores OOP](#)
- [TypeScript para programadores funcionales](#)

De lo contrario, salta a [Lo básico](#) o toma una copia en [Epub](#) o [PDF](#) forma.

Los basicos

Todos y cada uno de los valores en JavaScript tienen un conjunto de comportamientos que puede observar al ejecutar diferentes operaciones. Eso suena abstracto, pero como un ejemplo rápido, considere algunas operaciones que podríamos ejecutar en una variable llamada `mensaje`.

```
// Accediendo a la propiedad 'toLowerCase' // en  
'mensaje' y luego llámándolo message.toLowerCase();  
  
// Llamando a 'mensaje'  
mensaje();
```

Si desglosamos esto, la primera línea de código ejecutable accede a una propiedad llamada `toLowerCase` y luego la llama. El segundo intenta llamar al `mensaje` directamente.

Pero suponiendo que no conocemos el valor del `mensaje`, y eso es bastante común, no podemos decir de manera confiable qué resultados obtendremos al intentar ejecutar este código. El comportamiento de cada operación depende enteramente de qué valor teníamos en primer lugar.

- ¿El `mensaje` es invocable?
- ¿Tiene una propiedad llamada `toLowerCase`?
- Si es así, ¿se puede llamar a `toLowerCase`?
- Si ambos valores son invocables, ¿qué devuelven?

Las respuestas a estas preguntas suelen ser cosas que mantenemos en nuestras cabezas cuando escribimos JavaScript, y tenemos que esperar que tengamos todos los detalles correctos.

Digamos que el `mensaje` se definió de la siguiente manera.

```
const mensaje = "¡Hola mundo!";
```

Como probablemente puedas adivinar, si intentamos ejecutar `message.toLowerCase()`, obtendremos la misma cadena solo en minúsculas.

¿Qué pasa con esa segunda línea de código? Si está familiarizado con JavaScript, sabrá que esto falla con un excepción:

TypeError: el mensaje no es una función

Sería genial si pudiéramos evitar errores como este.

Cuando ejecutamos nuestro código, la forma en que nuestro tiempo de ejecución de JavaScript elige qué hacer es averiguar el ~~declarar~~- qué tipo de comportamientos y capacidades tiene. Eso es parte de lo que

TypeError se refiere a: dice que la cadena "¡Hola mundo!" no se puede llamar como función.

Para algunos valores, como las primitivas cadena y número , podemos identificar su tipo en tiempo de ejecución utilizando el operador `typeof` . Pero para otras cosas como funciones, no hay tiempo de ejecución correspondiente mecanismo para identificar sus tipos. Por ejemplo, considere esta función:

```
función fn(x) {  
    volver x.flip();  
}
```

Podemos observar leyendo el código que esta función solo funcionará si se le da un objeto con un propiedad `flip invocable` , pero JavaScript no muestra esta información de una manera que podamos verificar mientras el código se está ejecutando. La única forma en JavaScript puro de decir qué hace `fn` con un determinado el valor es llamarlo y ver qué pasa. Este tipo de comportamiento hace que sea difícil predecir qué código hará antes de que se ejecute, lo que significa que es más difícil saber qué hará su código mientras está escribiéndolo

Visto de esta manera, un escribe es el concepto de describir qué valores se pueden pasar a `fn` y cuáles se estrellará. JavaScript solo proporciona verdaderamente dinámica escribiendo - ejecutando el código para ver qué sucede.

La alternativa es utilizar un estático escriba el sistema para hacer predicciones sobre qué código se espera antes de corre.

Comprobación estática de tipos

Piense en ese `TypeError` que obtuvimos antes al intentar llamar a una cadena como una función.

La mayoría

gente no me gusta recibir ningún tipo de error al ejecutar su código, ¡esos se consideran errores!

Y cuando escribimos código nuevo, hacemos todo lo posible para evitar la introducción de nuevos errores.

Si agregamos solo un poco de código, guardamos nuestro archivo, volvemos a ejecutar el código e inmediatamente vemos el error, podríamos aislar el problema rápidamente; pero no siempre es así. ¡Es posible que no hayamos probado la función lo suficientemente a fondo, por lo que es posible que nunca nos encontremos con un error potencial que se produzca! O si tuviéramos la suerte de presenciar el error, podríamos haber terminado haciendo grandes refactorizaciones y agregando una gran cantidad de código diferente que nos vemos obligados a revisar.

Idealmente, podríamos tener una herramienta que nos ayude a encontrar estos errores tipo-checker como lo hace TypeScript. [Sistemas de tipos estáticos](#) nos informan de cuáles serán nuestros valores cuando ejecutemos nuestros programas. Un verificador de tipos como TypeScript usa esa información y nos dice cuándo las cosas pueden salirse de los rieles.

```
const mensaje = "¡hola!";
```

```
mensaje();
```

Esta expresión no es invocable.

El tipo 'String' no tiene firmas de llamada.

Ejecutar esa última muestra con TypeScript nos dará un mensaje de error antes de ejecutar el código en primer lugar.

Fallos no excepcionales

Hasta ahora hemos discutido ciertas cosas como errores de tiempo de ejecución, casos en los que el tiempo de ejecución de JavaScript nos dice que piensa que algo no tiene sentido. Esos casos surgen porque la [especificación ECMAScript](#) tiene [instrucciones explícitas](#) sobre [cómo debe comportarse](#) el lenguaje cuando se encuentra con algo inesperado.

Por ejemplo, la especificación dice que intentar llamar a algo que no se puede llamar debería arrojar un error. Tal vez eso suene como "comportamiento obvio", pero podría imaginar que acceder a una propiedad que no existe en un objeto también debería arrojar un error. En cambio, JavaScript nos da un comportamiento diferente y devuelve el valor undefined :

```
usuario constante = {
  nombre: "Daniel",
  edad: 26,};
```

```
usuario.ubicación; // devuelve indefinido
```

En última instancia, un sistema de tipo estático tiene que hacer la llamada sobre qué código debe marcarse como un error en su sistema, incluso si es JavaScript "válido" que no generará un error de inmediato. En TypeScript, el siguiente código produce un error sobre la ubicación no definida:

```
usuario constante = {
    nombre: "Daniel",
    edad: 26, };

usuario.ubicación;
```

'ubicación' no existe en el tipo '{ nombre: cadena; edad: número; }'.^{adicional} la propiedad

Si bien a veces eso implica una compensación en lo que puede expresar, la intención es detectar errores legítimos en nuestros programas. Y capturas de TypeScript ^{mucho} de errores legítimos.

Por ejemplo: errores tipográficos,

```
const anuncio = "¡Hola mundo!";

// ¿Qué tan rápido puedes detectar los errores
tipográficos? anuncio.toLocaleLowercase();
anuncio.toLocalLowerCase();

// Probablemente quisimos escribir esto...
anuncio.toLocaleLowerCase();
```

funciones no llamadas,

```
function flipCoin() { //
    Pretende ser Math.random() return
    Math.random < 0.5;

    El operador '<' no se puede aplicar a los tipos '() => número' y 'número'.
```

}

o errores lógicos básicos.

```
valor constante = Math.random() < 0.5 ? "a" : "b"; si (valor !==
"a") { // ...
```

```
} más si (valor === "b") {
```

devolverá 'falso' ya que los tipos "a" y "b" no se superponen. Esta condición siempre **falso**.

```
// Ups, inalcanzable
}
```

Tipos de herramientas

TypeScript puede detectar errores cuando cometemos errores en nuestro código. Eso es genial, pero TypeScript **además** puede evitar que cometamos esos errores en primer lugar.

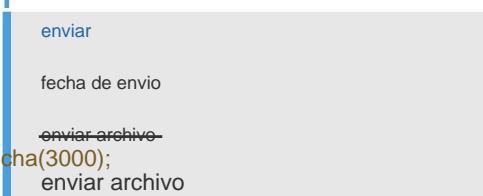
El verificador de tipos tiene información para verificar cosas como si estamos accediendo a las propiedades correctas en variables y otras propiedades. Una vez que tiene esa información, también puede iniciar propiedades o tipos que **quizás** desee usar.

Eso significa que TypeScript también se puede aprovechar para editar código, y el verificador de tipo central puede proporcionar mensajes de error y completar el código a medida que escribe en el editor. Eso es parte de lo que la gente suele referirse cuando habla de herramientas en TypeScript.

```
importar expreso de "expreso"; const
aplicación = express();
```

```
app.get("/", función (requerido, res) {
```

```
nada.
```



Enviar

fecha de envío

Enviar archivo

```
});
```

aplicación.escucha(3000);

Enviar archivo

TypeScript se toma muy en serio las herramientas, y eso va más allá de las terminaciones y los errores mientras escribe. Un editor que admite TypeScript puede ofrecer "arreglos rápidos" para corregir errores automáticamente, refactorizaciones para reorganizar fácilmente el código y funciones de navegación útiles para saltar a las definiciones de una variable o encontrar todas las referencias a una variable determinada. Todo esto se basa en el verificador de tipos y es totalmente multiplataforma, por lo que es probable que [su editor favorito tenga compatibilidad con TypeScript disponible](#).

tsc , el compilador de TypeScript

Hemos estado hablando sobre la verificación de tipos, pero aún no hemos utilizado nuestro tipo inspector . Vamos a llegar para familiarizarse con nuestro nuevo amigo tsc , el compilador de TypeScript. Primero tendremos que tomarlo a través de npm.

```
npm install -g mecanografiado
```

Esto instala el compilador de TypeScript tsc globalmente. Puede usar npx o herramientas similares si lo prefiere ejecute tsc desde un paquete local de node_modules en su lugar.

Ahora pasemos a una carpeta vacía e intentemos escribir nuestro primer programa TypeScript: hello.ts :

```
// Saluda al mundo.  
consola.log("¡Hola mundo!");
```

Tenga en cuenta que aquí no hay lujo; este programa "hola mundo" se ve idéntico a lo que escribirías para un programa "hola mundo" en JavaScript. Y ahora vamos a verificar el tipo ejecutando el comando tsc que fue instalado para nosotros por el paquete mecanografiado .

```
tsc hola.ts
```

¡Después!

Espera, ¿ "tada" qué exactamente? ¡Ejecutamos tsc y no pasó nada! Bueno, no hubo errores tipográficos, así que no obtuvimos ningún resultado en nuestra consola ya que no había nada que informar.

Pero verifique nuevamente: en su lugar, obtuvimos algo de salida. Si buscamos en nuestro directorio actual, veremos un archivo hello.js junto a hello.ts . Esa es la salida de nuestro archivo hello.ts después de tsc en un archivo JavaScript compilado o transformado simple. Y si revisamos el contenido, veremos qué TypeScript escupe después de procesar un archivo .ts :

```
// Saluda al mundo.  
consola.log("¡Hola mundo!");
```

En este caso, TypeScript tenía muy poco que transformar, por lo que se ve idéntico a lo que escribimos. El compilador intenta emitir un código limpio y legible que se parece a algo que escribiría una persona. Si bien eso no siempre es tan fácil, TypeScript sangra constantemente, tiene en cuenta cuándo nuestro código se extiende a través de diferentes líneas de código e intenta mantener los comentarios.

¿Qué pasa si nosotros hizo introducir un error de verificación de tipo? Reescribamos hello.ts :

```
// Esta es una función de bienvenida de uso general de grado industrial: function
saludar(persona, fecha) { console.log(`Hola ${persona}, ¡hoy es ${fecha}!`);

}

saludar("Brendan");
```

Si ejecutamos tsc hello.ts nuevamente, ¡observe que obtenemos un error en la línea de comando!

Esperaba 2 argumentos, pero obtuve 1.

TypeScript nos dice que olvidamos pasar un argumento a la función de saludo , y con razón. Hasta ahora, solo hemos escrito JavaScript estándar y, sin embargo, la verificación de tipos aún pudo encontrar problemas con nuestro código. Gracias mecanografiado!

Emitiendo con errores

Una cosa que quizás no haya notado en el último ejemplo fue que nuestro archivo hello.js cambió nuevamente. Si abrimos ese archivo, veremos que el contenido sigue siendo básicamente el mismo que nuestro archivo de entrada. Eso puede ser un poco sorprendente dado el hecho de que tsc informó un error sobre nuestro código, pero esto se basa en uno de los valores fundamentales de TypeScript: la mayor parte del tiempo, TypeScript sabrá mejor que

Para reiterar lo anterior, el código de verificación de tipo limita los tipos de programas que puede ejecutar, por lo que hay una compensación sobre qué tipo de cosas encuentra aceptable un verificador de tipo. La mayoría de las veces está bien, pero hay escenarios en los que esos controles se interponen. Por ejemplo, imagínese migrando código JavaScript a TypeScript e introduciendo errores de verificación de tipos. Eventualmente, podrá limpiar las cosas para el verificador de tipos, ¡pero ese código JavaScript original ya estaba funcionando! ¿Por qué debería convertirlo a TypeScript en un impedimento para ejecutarlo?

Entonces TypeScript no se interpone en tu camino. Por supuesto, con el tiempo, es posible que desee estar un poco más a la defensiva contra los errores y hacer que TypeScript actúe un poco más estrictamente. En ese caso, puede utilizar el

[noEmitOnError](#) opción del compilador. Intente cambiar su archivo hello.ts y ejecute tsc con eso

bandera:

```
tsc --noEmitOnError hola.ts
```

Notarás que hello.js nunca se actualiza.

Tipos explícitos

Hasta ahora, no le hemos dicho a TypeScript qué persona o fecha son. Editemos el código para contar TypeScript esa persona es una cadena , y esa fecha debe ser un objeto Fecha . También usaremos el Método toDateString() en la fecha .

```
función saludar(persona: cadena, fecha: Fecha) {  
    console.log(`Hola ${persona}, hoy es ${date.toDateString()}!`);  
}
```

lo que hicimos fue agregar anotaciones de tipo de en persona y fecha para describir qué tipos de valores saludar se puede llamar con. Puede leer esa firma como y una saludar toma una persona de tipo cuerda , fecha de tipo Fecha ".

Con esto, TypeScript puede informarnos sobre otros casos en los que saludar podría haberse llamado incorrectamente. Por ejemplo...

```
función saludar(persona: cadena, fecha: Fecha) {  
    console.log(`Hola ${persona}, hoy es ${date.toDateString()}!`);  
}  
  
saludar("Maddison", Fecha());
```

El argumento de tipo 'cadena' no se puede asignar al parámetro de tipo 'Fecha'.

¿Eh? TypeScript informó un error en nuestro segundo argumento, pero ¿por qué?

Quizás sorprendentemente, llamar a Date() en JavaScript devuelve una cadena . Por otro lado, construir una Fecha con new Date() en realidad nos da lo que esperábamos.

De todos modos, podemos corregir rápidamente el error:

```
función saludar(persona: cadena, fecha: Fecha) {
    console.log(`Hola ${persona}, hoy es ${fecha.toDateString()}!`);
}

saludar("Maddison", nueva Fecha());
```

Tenga en cuenta que no siempre tenemos que escribir anotaciones de tipo explícitas. En muchos casos, TypeScript puede incluso simplemente inferir (o "descifrar") los tipos para nosotros incluso si los omitimos.

```
let msg = "¡Hola!";
        dejar mensaje: cadena
```

Aunque no le dijimos a TypeScript que msg tenía la cadena de tipo , fue capaz de averiguarlo.

Esa es una característica, y es mejor no agregar anotaciones cuando el sistema de tipos termine infiriendo el mismo tipo de todos modos.

Nota: La burbuja de mensaje dentro del ejemplo de código anterior es lo que su editor mostraría si tuviera flotaba sobre la palabra.

Tipos borrados

Echemos un vistazo a lo que sucede cuando compilamos la función anterior saludar con tsc para generar JavaScript:

```
"uso estricto";
function saludar(persona, fecha)
    { console.log("Hola ".concat(persona, ", hoy es ").concat(fecha.toDateString()));
}

saludar("Maddison", nueva Fecha());
```

Note dos cosas aquí:

1. Nuestros parámetros de persona y fecha ya no tienen anotaciones de tipo.
2. Nuestra "cadena de plantilla", esa cadena que usaba acentos graves (el carácter `), se convirtió en cadenas simples con concatenaciones.

Más sobre ese segundo punto más adelante, pero centrémonos ahora en el primer punto. Las anotaciones de tipo no forman parte de JavaScript (o ECMAScript para ser pedante), por lo que realmente no hay navegadores u otros tiempos de ejecución que solo puede ejecutar TypeScript sin modificar. Es por eso que TypeScript necesita un compilador en primer lugar: necesita alguna forma de eliminar o transformar cualquier código específico de TypeScript para que pueda ejecutarlo. La mayoría del código específico de TypeScript se borra y, del mismo modo, aquí nuestras anotaciones de tipo fueron completamente borrado.

Recuerde: las anotaciones de tipo nunca cambian el comportamiento del tiempo de ejecución de su programa.

bajar de nivel

Otra diferencia con lo anterior fue que nuestra cadena de plantilla se reescribió a partir de

```
`Hola ${persona}, hoy es ${date.toDateString()}`;
```

a

```
"Hola " + persona + ", hoy es" + fecha.toDateString() + "!";
```

¿Por qué pasó esto?

Las cadenas de plantilla son una función de una versión de ECMAScript llamada ECMAScript 2015 (también conocida como ECMAScript 6, ES2015, ES6, etc.: no pregunte). TypeScript tiene la capacidad de reescribir código de más reciente versiones de ECMAScript a versiones anteriores como ECMAScript 3 o ECMAScript 5 (también conocidas como ES3 y ES5). Este proceso de pasar de una versión más nueva o "superior" de ECMAScript a una más antigua o "inferior" a veces se le llama rebajando .

Por defecto, TypeScript apunta a ES3, una versión extremadamente antigua de ECMAScript. Podríamos haber elegido algo un poco más reciente usando el objetivo opción. Corriendo con --target es2015 cambia TypeScript para apuntar a ECMAScript 2015, lo que significa que el código debería poder ejecutarse en cualquier lugar. Se admite ECMAScript 2015. Así que ejecutar tsc --target es2015 hello.ts nos da la siguiente salida:

```
función saludar(persona, fecha) {
    console.log(`Hola ${persona}, hoy es ${date.toDateString()}!`);

} saludar("Maddison", nueva Fecha());
```

Si bien el objetivo predeterminado es ES3, la gran mayoría de los navegadores actuales son compatibles con ES2015. Por lo tanto, la mayoría de los desarrolladores pueden especificar de forma segura ES2015 o superior como objetivo, a menos que la compatibilidad con ciertos antiguos navegadores es importante.

Rigor

Diferentes usuarios llegan a TypeScript buscando diferentes cosas en un verificador de tipos. Algunas personas buscan una experiencia de suscripción más flexible que pueda ayudar a validar solo algunas partes de su programa y aún así tener herramientas decentes. Esta es la experiencia predeterminada con TypeScript, donde los tipos son opcionales, la inferencia toma los tipos más indulgentes y no hay verificación de valores potencialmente nulos o indefinidos . Al igual que la forma en que tsc emite ante los errores, estos valores predeterminados se establecen para mantenerse fuera de su camino. Si está migrando JavaScript existente, ese podría ser un primer paso deseable.

Por el contrario, muchos usuarios prefieren que TypeScript valide todo lo que pueda de inmediato, y es por eso que el lenguaje también proporciona configuraciones estrictas. Esta configuración estricta convierte la verificación de tipo estático de un interruptor (ya sea que su código esté verificado o no) en algo más parecido a un dial. Cuanto más suba este dial, más comprobará TypeScript por usted. Esto puede requerir un poco de trabajo adicional, pero en términos generales se amortiza a largo plazo y permite controles más exhaustivos y herramientas más precisas. Cuando sea posible, una nueva base de código siempre debe activar estas comprobaciones estrictas.

TypeScript tiene varios indicadores de rigor de verificación de tipo que se pueden activar o desactivar, y todos nuestros ejemplos se escribirán con todos ellos habilitados a menos que se indique lo contrario. [el estricto](#) indicador en la CLI, o "estricto": verdadero en [un tsconfig.json](#) los activa todos simultáneamente, pero podemos desactivarlos individualmente. Los dos más importantes que debe conocer son [noImplicitAny](#) y [estrictosNullChecks](#).

noImplicitAny

Recuerde que, en algunos lugares, TypeScript no intenta inferir tipos por nosotros y, en cambio, recurre al tipo más indulgente: any . Esto no es lo peor que puede pasar; después de todo, recurrir a cualquiera es solo la experiencia simple de JavaScript de todos modos.

Sin embargo, usar `any` a menudo anula el propósito de usar TypeScript en primer lugar. Cuanto más tipo que tiene su programa, más validación y herramientas obtendrá, lo que significa que se encontrará con menos errores a medida que codifica. Activando el [`nolnimplicitAny`](#) flag emitirá un error en cualquier variable cuya type se deduce implícitamente como `any`.

estrictosNullChecks

De forma predeterminada, los valores como nulo e indefinido se pueden asignar a cualquier otro tipo. esto puede hacer escribir algo de código es más fácil, pero olvidar manejar nulo e indefinido es la causa de innumerables errores en el mundo: ¡algunos lo consideran un [error de mil millones de dólares!](#) Los [`controles nulos estrictos`](#) bandera hace manejo nulo e indefinido más explícito, y repuestos de preocuparnos por si olvidó para manejar nulo e indefinido.

Tipos cotidianos

En este capítulo, cubriremos algunos de los tipos de valores más comunes que encontrará en el código JavaScript, y explique las formas correspondientes de describir esos tipos en TypeScript. Esto no es un exhaustivo lista, y los capítulos futuros describirán más formas de nombrar y usar otros tipos.

Los tipos también pueden aparecer en mucho más [lugares](#) que solo anotaciones de tipo. A medida que aprendemos sobre los tipos ellos mismos, también aprenderemos sobre los lugares donde podemos referirnos a estos tipos para formar nuevas construcciones

Comenzaremos revisando los tipos más básicos y comunes que puede encontrar al escribir Código JavaScript o TypeScript. Estos luego formarán los bloques de construcción centrales de tipos más complejos.

Las primitivas: cadena , número , y booleano

JavaScript tiene tres [primitivas de uso muy común](#): `string` , [tipo correspondiente en](#) `número` , y `booleano` . cada uno tiene un TypeScript. Como era de esperar, estos son los mismos nombres que vería si usó el operador `typeof` de JavaScript en un valor de esos tipos:

- `cadena` representa valores de cadena como "Hola, mundo"
- `number` es para números como 42 . JavaScript no tiene un valor de tiempo de ejecución especial para números enteros, entonces no hay equivalente a `int` o `float` - todo es simplemente número
- `booleano` es para los dos valores `verdadero` y `falso`

Los nombres de tipo `String` , `Número` , y `booleanos` (que comienzan con letras mayúsculas) son legales, pero se refieren a algunos tipos integrados especiales que muy raramente aparecerán en su código. Siempre usar `cadena` , `número` , o `booleano` para tipos.

arreglos

Para especificar el tipo de una matriz como `[1, 2, 3]` , puede usar la sintaxis `number[]` ; esta sintaxis funciona para cualquier tipo (por ejemplo , `string[]` es una matriz de cadenas, etc.). También puede ver esto escrito como `matriz<número>` , lo que significa lo mismo. Aprenderemos más sobre la sintaxis `T<U>` cuando nosotros cubrimos `genéricos` .

Tenga en cuenta que `[número]` es una cosa diferente; consulte la sección sobre [Tuplas](#).

ningún

TypeScript también tiene un tipo especial, `any`, que puede usar cuando no quiera un tipo particular valor para causar errores de verificación de tipo.

Cuando un valor es de tipo `any`, puede acceder a cualquier propiedad del mismo (que a su vez será de tipo `any`), llámelo como una función, asígnelo a (o desde) un valor de cualquier tipo, o prácticamente cualquier otra cosa eso es sintácticamente legal:

```
sea obj: cualquiera = { x: 0 };
// Ninguna de las siguientes líneas de código generará errores de compilación.
// El uso de `cualquiera` deshabilita todas las comprobaciones de tipo adicionales, y se
// supone que // conoce el entorno mejor que TypeScript.
obj.foo();
objeto();
obj.bar = 100;
obj = "hola";
const n: numero = obj;
```

Cualquier tipo es útil cuando no desea escribir un tipo largo solo para convencer a TypeScript que una línea de código en particular está bien.

nolmplicitAny

Cuando no especifica un tipo y TypeScript no puede deducirlo del contexto, el compilador generalmente por defecto a `cualquiera`.

Sin embargo, por lo general desea evitar esto, porque `ninguno` tiene verificación de tipo. Usar la bandera del compilador [`nolmplicitAny`](#) para marcar cualquier implícito como un error.

Escriba anotaciones en variables

Cuando declara una variable usando `const` para _____, _____, o dejar _____, opcionalmente puede agregar una anotación de tipo especificar explícitamente el tipo de la variable:

```
let myName: cadena = "Alicia";
```

TypeScript no usa declaraciones de estilo "tipos a la izquierda" como int x = 0; Escriba las anotaciones siempre vas después la cosa que se escribe.

En la mayoría de los casos, sin embargo, esto no es necesario. Siempre que sea posible, TypeScript intenta inferir automáticamente los tipos en su código. Por ejemplo, el tipo de una variable se infiere en función del tipo de su inicializador:

```
// No se necesita anotación de tipo -- 'myName' inferido como tipo 'string' let myName =
"Alice";
```

En su mayor parte, no necesita aprender explícitamente las reglas de inferencia. Si está comenzando, intente usar menos anotaciones de tipo de las que cree; se sorprenderá de la cantidad de anotaciones que necesita para que TypeScript comprenda completamente lo que está sucediendo.

Funciones

Las funciones son el medio principal para pasar datos en JavaScript. TypeScript le permite especificar los tipos de los valores de entrada y salida de las funciones.

Anotaciones de tipo de parámetro

Cuando declara una función, puede agregar anotaciones de tipo después de cada parámetro para declarar qué tipos de parámetros acepta la función. Las anotaciones de tipo de parámetro van después del nombre del parámetro:

```
// Función de anotación de tipo de
parámetro saludar (nombre: cadena)
{ console.log ("Hola, " + nombre.toUpperCase() + "!!");
}
```

Cuando un parámetro tiene una anotación de tipo, se comprobarán los argumentos de esa función:

```
// ¡Sería un error de tiempo de ejecución si se ejecuta!
saludar(42);
```

de tipo 'número' no se puede asignar al parámetro de tipo 'cadena'. El argumento

Incluso si no tiene anotaciones de tipo en sus parámetros, TypeScript aún verificará que pasó la cantidad correcta de argumentos.

Anotaciones de tipo de retorno

También puede agregar anotaciones de tipo de devolución. Las anotaciones de tipo de devolución aparecen después de la lista de parámetros:

```
función getFavoriteNumber(): número { return 26;  
}
```

Al igual que las anotaciones de tipo variable, por lo general no necesita una anotación de tipo de devolución porque TypeScript deducirá el tipo de devolución de la función en función de sus declaraciones de devolución . La anotación de tipo en el ejemplo anterior no cambia nada. Algunas bases de código especificarán explícitamente un tipo de devolución con fines de documentación, para evitar cambios accidentales o simplemente por preferencia personal.

Funciones anónimas

Las funciones anónimas son un poco diferentes de las declaraciones de funciones. Cuando aparece una función en un lugar donde TypeScript puede determinar cómo se va a llamar, los parámetros de esa función reciben automáticamente tipos.

Aquí hay un ejemplo:

```
// No hay anotaciones de tipo aquí, pero TypeScript puede detectar el error
const nombres = ["Alicia", "Bob", "Eva"];

// Tipificación contextual para la función
nombres.paraCada(función (es) {
    consola.log(s.toUpperCase());

    La propiedad 'toUpperCase' no existe en el tipo 'cadena'. Querías decir
    'a Mayúsculas'?

});

// La escritura contextual también se aplica a las funciones de flecha
nombres.paraCada(s => {
    consola.log(s.toUpperCase());

    La propiedad 'toUpperCase' no existe en el tipo 'cadena'. Querías decir
    'a Mayúsculas'?

});
```

Aunque los parámetros no tenían una anotación de tipo, TypeScript usó los tipos del función forEach , junto con el tipo inferido de la matriz, para determinar el tipo que tendrá s .

Este proceso se llama escritura contextual porque el contexto que la función ocurrió dentro informa qué tipo debe tener.

Al igual que con las reglas de inferencia, no necesita aprender explícitamente cómo sucede esto, pero comprender que sucede puede ayudarlo a notar cuándo no se necesitan anotaciones de tipo. Luego, Veremos más ejemplos de cómo el contexto en el que ocurre un valor puede afectar su tipo.

Tipos de objetos

Aparte de los primitivos, el tipo de tipo más común que encontrará es un . Esto se refiere tipo de objeto a cualquier valor de JavaScript con propiedades, ¡que son casi todas! Para definir un tipo de objeto, simplemente enumere sus propiedades y sus tipos.

Por ejemplo, aquí hay una función que toma un objeto similar a un punto:

```
// La anotación de tipo del parámetro es un tipo de objeto
function imprimirCoord(pt: { x: número; y: número }) {
    console.log("El valor x de la coordenada es " + pt.x);
    console.log("El valor de la coordenada y es " + pt.y);
}
imprimirCoord({ x: 3, y: 7 });
```

Aquí, anotamos el parámetro con un tipo con dos propiedades, x e y , que son ambas de número de tipo de cualquier manera. , o ; para separar las propiedades, y el último separador es opcional

La parte tipo de cada propiedad también es opcional. Si no especifica un tipo, se supondrá que es cualquiera

Propiedades opcionales

Los tipos de objetos también pueden especificar que algunas o todas sus propiedades están opcional . Para hacer esto, agregue un ? después del nombre de la propiedad:

```
function printName(obj: { primero: cadena; ¿último?: cadena }) {
    // ...
}
// Ambos bien
printName({ primero: "Bob" });
printName({ primero: "Alice", último: "Alisson" });
```

En JavaScript, si accede a una propiedad que no existe, obtendrá el valor indefinido en lugar de un error de tiempo de ejecución. Debido a esto, leer de una propiedad opcional, tendrá que buscar cuando no lo definió antes de usarlo.

```
function printName(obj: { primero: cadena; último?: cadena }) {
    // Error: ¡podría fallar si no se proporcionó 'obj.last'!
    consola.log(obj.last.toUpperCase());
```

El objeto es posiblemente 'indeterminado'.

```
if (obj.último !== indefinido) {
    // OK
    consola.log(obj.last.toUpperCase());
}

// Una alternativa segura usando la sintaxis moderna de JavaScript:
consola.log(obj.último?.toUpperCase());
}
```

Tipos de unión

El sistema de tipos de TypeScript le permite crear nuevos tipos a partir de los existentes utilizando una gran variedad de operadores. Ahora que sabemos cómo escribir algunos tipos, es hora de comenzar a combinatorio ellos en formas interesantes.

Definición de un tipo de unión

La primera forma de combinar tipos que puede ver es un tipo Union. Un tipo de unión es un tipo formado por dos o más de otros tipos, que representan valores que pueden ser de estos tipos. Nos referimos a cada de este tipo como el sindicato miembros .

Escribamos una función que pueda operar en cadenas o números:

```
función printId(id: número | cadena) {
    console.log("Tu identificación es: " + identificación);
}
// OK
imprimirId(101);
// OK
imprimirId("202");
// Error
printId({ milD: 22342 });
```

Argumento de tipo '{ milD: número; }' no es assignable al parámetro de escriba 'cadena | número'.

Trabajo con tipos de unión

Es fácil para proveer un valor que coincide con un tipo de unión: simplemente proporcione un tipo que coincida con cualquiera de los los miembros del sindicato. Si usted tener un valor de un tipo de unión, ¿cómo se trabaja con él?

TypeScript solo permitirá una operación si es válida para el miembro de la unión. Por ejemplo, si tienes la cadena de union | number , no puede usar métodos que solo están disponibles en string :

```
función printId(id: número | cadena) {  
    consola.log(id.toUpperCase());
```

La propiedad 'toUpperCase' no existe en el tipo 'cadena | número'.

La propiedad 'toUpperCase' no existe en el tipo 'number'.

```
}
```

La solución son las angostas la unión con código, lo mismo que harías en JavaScript sin tipo anotaciones. Estrechamiento ocurre cuando TypeScript puede deducir un tipo más específico para un valor basado sobre la estructura del código.

Por ejemplo, TypeScript sabe que solo un valor de cadena tendrá un tipo de valor " cadena" :

```
función printId(id: número | cadena) {  
    if (tipo de id === "cadena") {  
        // En esta rama, la identificación es del tipo 'cadena'  
        consola.log(id.toUpperCase());  
    } más {  
        // Aquí, la identificación es del tipo 'número'  
        consola.log(id);  
    }  
}
```

Otro ejemplo es usar una función como Array.isArray :

```
función bienvenidaGente(x: cadena[] | cadena) {
    si (Array.isArray(x)) {
        // Aquí: 'x' es 'cadena[]'
        console.log("Hola, " } else { + x.join(" y "));
        // Aquí: 'x' es 'cadena'
        console.log("Bienvenido viajero solitario" + x);
    }
}
```

Tenga en cuenta que en la rama `else`, no necesitamos hacer nada especial: si `x` no fuera una cadena `[]`, entonces debe haber sido una cadena .

A veces tendrás un sindicato donde todos los miembros tienen algo en común. Por ejemplo, tanto las matrices como las cadenas tienen un método de división . Si cada miembro de un sindicato tiene una propiedad en común, puede usar esa propiedad sin restringir:

```
// El tipo de retorno se infiere como número[] | cuerda
función obtenerPrimeroTres(x: número[] | cadena) {
    return x.slice(0, 3);
}
```

Puede ser confuso que un Unión de tipos parece tener la intersección de las propiedades de esos tipos. Este no es un accidente - el nombre Unión proviene de la teoría de tipos. Los Unión número | la cadena es compuesto tomando los datos de los valores de cada tipo. Nótese que dados dos conjuntos con sus correspondientes Unión de cada conjunto, solo el intersección de esos hechos se aplica a la Unión de los propios conjuntos. Para ejemplo, si tuviéramos una sala de personas altas con sombreros, y otra sala de hablantes de español que usaran sombreros, después de combinar esas habitaciones, lo único que sabemos sobre cada persona es que debe ser llevar sombrero.

Tipo de alias

Hemos estado usando tipos de objetos y tipos de unión escribiéndolos directamente en anotaciones de tipo. Esto es conveniente, pero es común querer usar el mismo tipo más de una vez y referirse a él por un solo nombre.

A alias tipo de es exactamente eso - un nombre para cualquier escribe La sintaxis para un alias de tipo es:

```

tipo Punto = { x:
  número; y:
  número; };

// Exactamente igual que el ejemplo anterior function
printCoord(pt: Point) { console.log("El valor x de la
  coordenada es " console.log("El valor y de la coordenada + pt.x); +
  es "
}

imprimirCoord({ x: 100, y: 100 });

```

De hecho, puede usar un alias de tipo para dar un nombre a cualquier tipo, no solo a un tipo de objeto. Por ejemplo, un alias de tipo puede nombrar un tipo de unión:

```
tipo ID = número | cuerda;
```

Tenga en cuenta que los ~~alias~~ son alias: no puede usar alias de tipo para crear "versiones" diferentes/distintas del mismo tipo. Cuando usa el alias, es exactamente como si hubiera escrito el tipo con alias. En otras palabras, este código puede ser ilegal, pero está bien según TypeScript porque ambos tipos son alias para el mismo tipo:

```

escriba UserInputSanitizedString = cadena;

function sanitelizeInput(str: cadena): UserInputSanitizedString {
  volver desinfectar (str);
}

// Crear una entrada sanitizada let
userInput = sanitelizeInput(getInput());

// Todavía se puede reasignar con una cadena aunque userInput
= "nueva entrada";

```

Interfaces

Un declaración de interfaz es otra forma de nombrar un tipo de objeto:

```
Punto de interfaz {
    x: número;
    y: número;
}

function imprimirCoord(pt: Punto) {
    console.log("El valor x de la coordenada es " + pt.x);
    console.log("El valor y de la coordenada es " + pt.y);
}

imprimirCoord({ x: 100, y: 100 });
```

Al igual que cuando usamos un alias de tipo arriba, el ejemplo funciona como si hubiéramos usado un alias anónimo

tipo de objeto. TypeScript solo se preocupa por el valor que pasamos a printCoord

- solo le importa que tenga las propiedades esperadas. Preocupándonos únicamente de la estructura y capacidades de los tipos es por eso que llamamos a TypeScript un sistema de tipos

Diferencias entre alias de tipo e interfaces

Los alias de tipo y las interfaces son muy similares y, en muchos casos, puede elegir entre ellos libremente.

Casi todas las funciones de una interfaz están disponibles en type , la distinción clave es que un type

no se puede volver a abrir para agregar nuevas propiedades frente a una interfaz que siempre es extensible.

Interfaz

Escribe

Ampliación de una interfaz

Extender un tipo a través de intersecciones

```
interfaz Animal { nombre:  
    cadena  
}
```

```
interfaz Oso extiende Animal { cariño:  
    booleano  
}
```

```
const oso = obtenerOso()  
oso.nombre  
osito.miel
```

```
tipo Animal =  
    { nombre: cadena  
}
```

```
tipo Oso = Animal & { miel:  
    booleano  
}
```

```
const oso = obtenerOso();  
oso.nombre; miel de oso;
```

Agregar nuevos campos a una interfaz existente

Un tipo no se puede cambiar después de haber sido creado

```
Ventana de interfaz  
{título: cadena  
}
```

```
ventana de interfaz {  
    ts: API de TypeScript  
}
```

```
const src = 'const a = "Hola mundo"';  
ventana.ts.transpileModule(origen, {});
```

```
escriba Ventana =  
    {título: cadena  
}
```

```
escriba Ventana =  
    { ts: TypeScriptAPI  
}
```

// Error: Identificador duplicado 'Windo

Aprenderá más sobre estos conceptos en capítulos posteriores, así que no se preocupe si no los entiende todos de inmediato.

- Antes de la versión 4.2 de TypeScript, los nombres de los [alias de tipo](#) aparecían en los [mensajes de error](#), a veces en lugar del tipo anónimo equivalente (que puede ser deseable o no). Las interfaces siempre se nombrarán en los mensajes de error.
- Es posible que los alias de tipo no participen [en la fusión de declaraciones](#), pero las interfaces sí.
- Las interfaces solo se pueden usar para [declarar las formas de los objetos](#), no para [cambiar el nombre de las primitivas](#).

- Los nombres de las interfaces siempre aparecen en su forma original en los mensajes de error, pero solamente cuando ellos se utilizan por nombre.

En su mayor parte, puede elegir según sus preferencias personales y TypeScript le dirá si necesita algo para ser el otro tipo de declaración. Si desea una heurística, use la interfaz hasta que necesite usar funciones del tipo .

Escriba aserciones

A veces tendrá información sobre el tipo de un valor que TypeScript no puede conocer.

Por ejemplo, si está utilizando `document.getElementById`, TypeScript solo sabe que esto devolver alguno tipo de elemento HTML, pero es posible que sepa que su página siempre tendrá un `HTMLCanvasElement` con un ID determinado.

En esta situación, puede utilizar un afirmación de tipo para especificar un tipo más específico:

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement
```

Al igual que una anotación de tipo, el compilador elimina las aserciones de tipo y no afectarán el tiempo de ejecución. comportamiento de su código.

También puede usar la sintaxis de paréntesis angular (excepto si el código está en un archivo `.tsx`), que es equivalente:

```
const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas")
```

Recordatorio: debido a que las aserciones de tipo se eliminan en tiempo de compilación, no hay una verificación de tiempo de ejecución asociada con una aserción de tipo. No se generará una excepción o un valor nulo si la afirmación de tipo es incorrecta.

TypeScript solo permite aserciones de tipo que se convierten en un tipo. mas específico o menos específico versión de un Esta regla previene coacciones "imposibles" como:

```
const x = "hola" como número;
```

La conversión del tipo 'cadena' al tipo 'número' puede ser un error porque ningún tipo se superpone suficientemente con el otro. si esto fuera intencional, primero convierta la expresión a 'desconocido'.

A veces, esta regla puede ser demasiado conservadora y no permitirá coacciones más complejas que podrían Sé valido. Si esto sucede, puede usar dos aserciones, primero para cualquier (o introducción , que vamos a desconocida más tarde), luego para el tipo deseado:

```
const a = (expr como cualquier) como T;
```

Tipos literales

Además de los tipos generales cadena y número en posiciones , podemos referirnos a específico cadenas y numeros de tipo.

Una forma de pensar en esto es considerar cómo JavaScript viene con diferentes formas de declarar un variable. Tanto var como let permiten cambiar lo que se mantiene dentro de la variable, y const lo hace. no. Esto se refleja en cómo TypeScript crea tipos para literales.

```
let cadenaCambiante = "Hola Mundo";
cadenaCambiante = "Hola mundo";
// Porque `cadenaCambiante` puede representar cualquier cadena posible, eso
// es como lo describe TypeScript en el sistema de tipos
cambiandoCadena;
```

vamos cambiandoCadena: cadena

```
const cadenaConstante = "Hola mundo";
// Debido a que `constantString` solo puede representar 1 cadena posible,
// tiene una representación de tipo literal
cadenaConstante;
```

const constanteString: "Hola Mundo"

Por sí mismos, los tipos literales no son muy valiosos:

```
sea x: "hola" = "hola";
// OK
x = "hola"; // ...
x = "hola";
```

El tipo "hola" no se puede asignar al tipo "hola".

¡No sirve de mucho tener una variable que solo puede tener un valor!

Pero combinando literales en uniones, puede expresar un concepto mucho más útil, por ejemplo, por funciones que solo aceptan un cierto conjunto de valores conocidos:

```
función imprimirTexto(s: cadena, alineación: "izquierda" | "derecha" | "centro") {
    // ...

} printText("Hola, mundo", "izquierda");
printText("G'day, compañero", "centro");
```

El tipo "centro" no se puede asignar al parámetro de tipo "izquierda" | "correcto" | "centro".

Los tipos literales numéricos funcionan de la misma manera:

```
función comparar (a: cadena, b: cadena): -1 | 0 | 1 {
    devolver a === b ? 0 : a > b ? 1 : -1;
}
```

Por supuesto, puede combinarlos con tipos no literales:

```
Opciones de interfaz {
    ancho: número;
}
función configurar(x: Opciones | "auto") {
    // ...
}
configurar ({ ancho: 100 });
configurar("auto");
configurar("automático");
```

El argumento de tipo "automático" no se puede asignar a un parámetro de tipo 'Opciones | "auto"'.

Hay un tipo más de tipo literal: los literales booleanos. Solo hay dos tipos de literales booleanos, y como puede suponer, son los tipos verdadero y falso . El tipo booleano en sí mismo es en realidad solo un alias para el sindicato true | falso .

Inferencia literal

Cuando inicializa una variable con un objeto, TypeScript asume que las propiedades de ese objeto podría cambiar los valores más adelante. Por ejemplo, si escribiste un código como este:

```
const obj = { contador: 0 };
si (alguna condición) {
    obj.contador = 1;
}
```

TypeScript no asume que la asignación de 1 a un campo que anteriormente tenía 0 es un error. Otra forma de decir esto es que obj.counter debe tener el tipo de número de tipos que se , no 0 , porque utilizan para determinar ambos comportamientos y escritura

Lo mismo se aplica a las cadenas:

```
const req = { url: "https://example.com", método: "GET" };
handleRequest(req.url, req.método);
```

El argumento de tipo 'cadena' no se puede asignar al parámetro de tipo "GET" | "CORREO"?"

En el ejemplo anterior, se deduce que `req.method` es `string`. Porque el código puede ser evaluado entre la creación de `req` y la llamada de `handleRequest` que podría asignar un nuevo cadena como "GUESS" a `req.method`, TypeScript considera que este código tiene un error.

Hay dos formas de evitar esto.

1. Puede cambiar la inferencia agregando una aserción de tipo en cualquier ubicación:

```
// Cambio 1:  
const req = { url: "https://example.com", método: "GET" as "GET" };  
// Cambio 2  
handleRequest(req.url, req.method as "GET");
```

El cambio 1 significa "Pretendo que el método `req` tenga siempre la tipo literal "OBTENER", prevención de la posible asignación de "GUESS" a ese campo después. El cambio 2 significa "Sé por otras razones, `req.method` tiene el valor "GET".

2. Puede usar `as const` para convertir todo el objeto en literales de tipo:

```
const req = { url: "https://example.com", método: "GET" } as const;  
handleRequest(req.url, req.método);
```

El sufijo `as const` actúa como `const` pero para el sistema de tipos, asegurando que todas las propiedades sean asignado el tipo literal en lugar de una versión más general como cadena o número .

nulo e indefinido

JavaScript tiene dos valores primitivos que se utilizan para señalar un valor ausente o no inicializado: nulo y indefinido _

TypeScript tiene dos correspondientes controles nulos mismos nombres. El comportamiento de estos tipos depende de si tiene los [estrictos NullChecks](#) opción activada.

StricNullChecks off

Con [estrictos controles nulos](#) apagado todavía se puede acceder a los valores que pueden ser nulos o indefinidos normalmente, y los valores nulo e indefinido se pueden asignar a una propiedad de cualquier tipo. Esto es

similar a cómo se comportan los lenguajes sin controles nulos (por ejemplo, C #, Java). La falta de verificación de estos los valores tienden a ser una fuente importante de errores; siempre recomendamos que la gente se vuelva [estrictosNullChecks](#) sobre si es práctico hacerlo en su base de código.

controles nulos estrictos en

Con [estrictos controles nulos](#) esos en , cuando un valor es nulo o indefinido , deberá probar valores antes de usar métodos o propiedades en ese valor. Al igual que verificar si no está definido antes de usar una propiedad opcional, podemos usar para [verificar valores](#) que podrían ser nulos :

```
function hacerAlgo(x: cadena | nulo) {
    si (x === nulo) {
        // hacer nada
    } más {
        consola.log("Hola, " + x.toUpperCase());
    }
}
```

Operador de aserción no nulo (¡Postfijo !)

TypeScript también tiene una sintaxis especial para eliminar nulos e indefinidos de un tipo sin haciendo ninguna comprobación explícita. escribiendo ! después de cualquier expresión es efectivamente una aserción de tipo que el el valor no es nulo o indefinido :

```
function vivirPeligrosamente(x?: número | nulo) {
    // No hay error
    consola.log(x!.toFixed());
}
```

Al igual que otras aseraciones de tipo, esto no cambia el comportamiento en tiempo de ejecución de su código, por lo que es ¡ Importante utilizar solamente ! cuando sabe que el valor es nulo o indefinido.

Enumeraciones

Las enumeraciones son una función añadida a JavaScript por TypeScript que permite describir un valor que podría ser una de un conjunto de posibles constantes con nombre. A diferencia de la mayoría de las funciones de no a TypeScript, esta es una adición de nivel de tipo a JavaScript, pero algo agregado al lenguaje y al tiempo de ejecución. Debido a esto,

es una función que debe saber que existe, pero tal vez no la use a menos que esté seguro. Puede leer más sobre las enumeraciones en la [página de referencia de Enum](#).

Primitivas menos comunes

Vale la pena mencionar el resto de las primitivas en JavaScript que se representan en el sistema de tipos. Aunque no profundizaremos aquí.

Empezando

Desde ES2020 en adelante, hay una primitiva en JavaScript que se usa para números enteros muy grandes, BigInt :

```
// Crear un bigint a través de la función BigInt const  
oneHundred: bigint = BigInt(100);  
  
// Crear un BigInt a través de la sintaxis literal const  
anotherHundred: bigint = 100n;
```

Puede obtener más información sobre BigInt en [las notas de la versión de TypeScript 3.2](#).

símbolo

Hay una primitiva en JavaScript que se usa para crear una referencia globalmente única a través de la función

Símbolo() :

```
const primerNombre = Símbolo("nombre");  
const segundoNombre = Símbolo("nombre");
```

```
if ( nombre === segundo nombre) {
```

devolverá 'false' ya que los tipos 'typeof firstName' y 'typeof secondName'. Esta condición siempre superponen. 'firstName' y 'typeof secondName' no se superponen.

```
    // Nunca puede pasar  
}
```

Puede obtener más información sobre ellos en la [página de referencia de Símbolos](#).

Estrechamiento

Imagina que tenemos una función llamada `padLeft`.

```
function padLeft(relleno: número | cadena, entrada: cadena): cadena {
    throw new Error("¡Aún no implementado!");
}
```

Si el relleno es un número , lo tratará como el número de espacios que queremos anteponer a input .

Si padding es una cadena , ~~padding se pone a padding o input padding~~ debemos implementar la lógica para cuando

```
function padLeft(relleno: número | cadena, entrada: cadena) { " ".repeat(relleno)
    devolver + entrada;
```

'string | number' no se puede asignar a un parámetro de tipo 'number'. ~~del~~ Argument de tipo

El tipo 'cadena' no se puede asignar al tipo 'número'.

```
}
```

Uh-oh, estamos recibiendo un error en el relleno . TypeScript nos advierte que agregar un número | cadena a un número podría no darnos lo que queremos, y es correcto. En otras palabras, no hemos verificado explícitamente si padding es un número primero, ni estamos manejando el caso donde es una cadena , así que hagamos exactamente eso.

```
function padLeft(relleno: número | cadena, entrada: cadena) {
    if (tipo de relleno === "número") {
        devolver ".repeat(relleno) + entrada;

    } relleno de retorno + entrada;
}
```

Si esto en su mayoría parece un código JavaScript poco interesante, ese es el punto. Aparte de las anotaciones que colocamos, este código TypeScript se parece a JavaScript. La idea es que TypeScript

El sistema de tipos tiene como objetivo hacer que sea lo más fácil posible escribir código JavaScript típico sin doblar hacia atrás para obtener seguridad de tipo.

Si bien puede no parecer mucho, en realidad hay muchas cosas ocultas aquí. Al igual que cómo TypeScript analiza los valores de tiempo de ejecución utilizando tipos estáticos, superpone el análisis de tipos en JavaScript construcciones de flujo de control de tiempo de ejecución como `if/else`, ternarios condicionales, bucles, comprobaciones de veracidad, etc., que pueden afectar a esos tipos.

Dentro de nuestra verificación `if`, TypeScript ve el tipo de relleno === "número" y lo entiende como un `any`. Una forma especial de código llamada `guardado` de `any`. TypeScript sigue las posibles rutas de ejecución que nuestro programa puede tomar para analizar el tipo más específico posible de un valor en una posición dada. Parece en estos cheques especiales (llamados) y `asignación` el proceso de refinar tipos para tipos más específicos que los declarados se llama. En muchos `editores` podemos observar estos tipos a medida que cambian, e incluso lo haremos en nuestros ejemplos.

```
function padLeft(relleno: número | cadena, entrada: cadena) {
    if (tipo de relleno === "número") {
        devolver " ".repetir(relleno) + entrada;
    }
    relleno de retorno + entrada;
}
```

(parámetro) relleno: número

(parámetro) relleno: cadena

Hay un par de construcciones diferentes que TypeScript entiende para estrechar.

tipos de guardias

Como hemos visto, JavaScript admite un operador `typeof` que puede dar información muy básica sobre el tipo de valores que tenemos en tiempo de ejecución. TypeScript espera que esto devuelva un determinado conjunto de cadenas:

- "cuerda"
- "número"
- "Empezando"
- "booleano"

- "símbolo"
- "indefinido"
- "objeto"
- "función"

Como vimos con `padLeft`, este operador aparece con bastante frecuencia en varias bibliotecas de JavaScript, y TypeScript puede entenderlo para limitar los tipos en diferentes ramas.

En TypeScript, comprobar el valor devuelto por `typeof` es una protección de tipos. Debido a que TypeScript codifica cómo opera `typeof` en diferentes valores, conoce algunas de sus peculiaridades en JavaScript.

Por ejemplo, observe que en la lista anterior, `typeof` no devuelve la cadena nula . Mira el siguiente ejemplo:

```
función imprimirTodas(cadenas: cadena | cadena[] | nulo) { if ( tipo de
cadenas === "objeto") { for (const s de cadenas) {
```

El objeto es posiblemente 'nulo'.

```
    consola.log(s);
}
} else if (typeof strs === "string") { console.log(strs); }
más { // no hacer nada

}
```

En la función `printAll`, tratamos de verificar si `strs` es un `objeto` para ver si es un tipo de matriz (ahora podría ser un buen momento para reforzar que las matrices son tipos de objetos en JavaScript). ¡Pero resulta que en JavaScript, `typeof null` es en realidad "objeto" ! Este es uno de esos desafortunados accidentes de la historia.

Es posible que los usuarios con suficiente experiencia no se sorprendan, pero no todos se han topado con esto en JavaScript; afortunadamente, TypeScript nos permite saber que `strs` solo se redujo a `string[] | null` en lugar de solo `string[]`.

Esta podría ser una buena transición a lo que llamaremos verificación de "veracidad".

Estrechamiento de veracidad

Veracidad puede no ser una palabra que encuentre en el diccionario, pero es algo que escuchar en JavaScript.

En JavaScript, podemos usar cualquier expresión en condicionales, `&&`s, `||`s, declaraciones `if`, booleanas negaciones (`!`), y más. Como ejemplo, si las sentencias no esperan que su condición siempre tener el tipo booleano .

```
función getUsersOnlineMessage(numUsersOnline: número) {
    if (numUsersOnline) {
        return `¡Hay ${numUsersOnline} en línea ahora!`;
    }
    return "No hay nadie aquí. :(";
}
```

En JavaScript, las construcciones como `if` first "forzan" sus condiciones a boolean s para darles sentido, y luego elegir sus ramas dependiendo de si el resultado es verdadero o falso . Valores como

- 0
- Yaya
- "" (la cadena vacía)
- 0n (la versión bigint de cero)
- nulo
- indefinido

toda coacción a falso , y otros valores se convierten en verdaderos .Siempre puedes obligar a los valores a boolean s ejecutándolos a través de la función booleana , o usando la negación booleana doble más corta. (Este último tiene la ventaja de que TypeScript infiere un tipo booleano literal estrecho verdadero , mientras se infiere el primero como tipo booleano).

```
// ambos dan como resultado 'verdadero'
Boolean("hola"); // tipo: booleano, valor: verdadero
!!"mundo"; // tipo: verdadero,           valor: verdadero
```

Es bastante popular aprovechar este comportamiento, especialmente para protegerse contra valores como nulo o indefinido _ Como ejemplo, intentemos usarlo para nuestra función `imprimirTodo` .

```
function imprimirTodo(cadenas: cadena | cadena[] | nulo) { if (cadenas
  && tipo de cadenas === "objeto") { for (const s of strs)
    { console.log(s);

    }
  } else if (typeof strs === "string") { console.log(strs);

  }
}
```

Notará que nos hemos deshecho del error anterior al verificar si strs es veraz. Esto al menos nos previene de temidos errores cuando ejecutamos nuestro código como:

TypeError: nulo no es iterable

Sin embargo, tenga en cuenta que la verificación de la veracidad de las primitivas a menudo puede ser propensa a errores. Como ejemplo, considere un intento diferente de escribir printAll

```
función imprimirTodo(cadenas: cadena | cadena[] | nulo) {
  // !!!!!!!!
  // ¡NO HAGAS ESTO!
  // SIGUE
  LEYENDO // !!!!!!!!
  if
  (strs) { if (typeof strs ===
    "objeto") { for (const s of strs)
      { console.log(s);

      }
    } else if (typeof strs === "string") { console.log(strs);

    }
  }
}
```

Envolvimos todo el cuerpo de la función en una verificación de veracidad, pero esto tiene una desventaja sutil: es posible que ya no estemos manejando correctamente el caso de la cadena vacía.

TypeScript no nos hace daño aquí en absoluto, pero vale la pena señalar este comportamiento si está menos familiarizado con JavaScript. TypeScript a menudo puede ayudarlo a detectar errores desde el principio, pero si elige hacerlo ~~negado~~ un valor, hay mucho que puede hacer sin ser demasiado prescriptivo. Si tu quieres tu puedes asegúrese de manejar situaciones como estas con un linter.

Una última palabra sobre la reducción por veracidad es que las negaciones booleanas con ! filtrar de negado sucursales.

```
funcion multiplicartodo(  
    valores: n mero[] | indefinido,  
    factor: n mero  
): n mero[] | indefinido {  
    si (!valores) {  
        valores de retorno ;  
    } m s {  
        devolver valores.map((x) => x * factor);  
    }  
}
```

Reducción de la igualdad

TypeScript también usa declaraciones de cambio y comprobaciones de igualdad como , != , == , y != a === tipos estrechos. Por ejemplo:

```
ejemplo de función (x: cadena | número, y: cadena | booleano) { if (x === y) {
```

```
// Ahora podemos llamar a cualquier método de 'cadena' en 'x' o 'y'.  
x.toUpperCase();
```

(método) String.toUpperCase(): cadena

```
y.toLowerCase();
```

(método) String.toLowerCase(): cadena

```
} más
```

```
{ consola.log(x);
```

(parámetro) x: cadena | número

```
consola.log(y);
```

(parámetro) y: cadena | booleano

```
}
```

```
}
```

Cuando verificamos que x e y son iguales en el ejemplo anterior, TypeScript sabía que sus tipos también tenían que ser iguales. Dado que la cadena es el único tipo común que pueden adoptar tanto x como y , TypeScript sabe que x e y deben ser una cadena en la primera rama.

La verificación de valores literales específicos (a diferencia de las variables) también funciona. En nuestra sección sobre el estrechamiento de la veracidad, escribimos una función printAll que era propensa a errores porque accidentalmente no manejaba correctamente las cadenas vacías. En su lugar, podríamos haber hecho una verificación específica para bloquear los nulos , y TypeScript aún elimina correctamente los nulos del tipo de str

```

function imprimirTodo(cadenas: cadena | cadena[] | nulo) { if (cadenas !
== nulo) { if ( tipo de cadenas === "objeto") { for (const s of strs ) {

    (parámetro) strs: cadena[]

        consola.log(s);
    }
} else if (typeof strs === "string") { console.log(strs);

    (parámetro) strs: cadena

}
}
}
}

```

Las comprobaciones de igualdad más flexibles de JavaScript con == y != también se reducen correctamente. Si no está familiarizado, verificar si algo == nulo en realidad no solo verifica si es específicamente el valor nulo , sino que también verifica si es potencialmente indefinido . Lo mismo se aplica a == undefined : comprueba si un valor es nulo o indefinido .

```

interfaz Contenedor { valor:
    número | nulo | indefinido;
}

función multiplicarValor(contenedor: Contenedor, factor: número) { // Eliminar tanto
    'nulo' como 'indefinido' del tipo. if (contenedor.valor != nulo)
    { console.log(contenedor.valor);

        (propiedad) Container.value: número

        // Ahora podemos multiplicar con seguridad 'container.value'.
        contenedor.valor *= factor;
    }
}

```

El estrechamiento del operador in

JavaScript tiene un operador para determinar si un objeto tiene una propiedad con un nombre: el operador `in`. TypeScript tiene esto en cuenta como una forma de reducir los tipos potenciales.

Por ejemplo, con el código: "valor" en `x` . donde "valor" es un literal de cadena y `x` es un tipo de unión. La rama "verdadera" reduce los tipos de `x` que tienen una propiedad opcional o requerida `valor` , y la rama "falsa" se reduce a tipos que tienen una propiedad opcional o faltante `valor` .

```
tipo Fish = { nadar: () => void }; escriba Bird
= { volar: () => void };

función mover (animal: pez | pájaro) {
    if ("nadar" en animal) { return
        animal.swim();
    }

    volver animal.fly();
}
```

Para reiterar, existirán propiedades opcionales en ambos lados para el estrechamiento, por ejemplo, un ser humano podría nadar y volar (con el equipo adecuado) y, por lo tanto, debería aparecer en ambos lados del control :

```

tipo Fish = { nadar: () => void }; escriba Bird
= { volar: () => void }; tipo Humano =
{ nadar?: () => vacío; volar?: () => void };

función mover (animal: pez | pájaro | humano) {
  if ("nadar" en animal) { animal;

    _____
    (parámetro) animal: Pescado | Humano

} más
{ animal;
  _____
  (parámetro) animal: Pájaro | Humano

}
}

```

instancia de estrechamiento

JavaScript tiene un operador para verificar si un valor es o no una "instancia" de otro valor.

Más específicamente, en JavaScript `x instanceof Foo` comprueba si contiene `Foo.prototype` en el prototipo de cadena de `x`.

Si bien no profundizaremos aquí, y verá más de esto cuando ingresemos a las clases, aún pueden ser útiles para la mayoría de los valores que se pueden construir con `new`. Como habrás adivinado, `instanceof` también es un tipo de `TypeScript` que se estrecha en las ramas protegidas por `instanceof`.

```

función logValue(x: Fecha | cadena) {
  if (x instanceof Fecha)
    { console.log(x.toUTCString());

      _____
      (parámetro) x: Fecha

} más
{ consola.log(x.toUpperCase());
  _____
  (parámetro) x: cadena

}
}

```

Tareas

Como mencionamos anteriormente, cuando asignamos a cualquier variable, TypeScript mira el lado derecho de la asignación y estrecha el lado izquierdo apropiadamente.

```
sea x = Math.random() < 0.5 ? 10 : "¡hola mundo!";
```

sea x: cadena | número

```
x = 1;
```

```
consola.log(x);
```

sea x: numero

```
x = "¡adiós!";
```

```
consola.log(x);
```

sea x: cadena

Observe que cada una de estas asignaciones es válida. Aunque el tipo observado de x cambió a número después de nuestra primera asignación, todavía pudimos asignar una cadena a x . Esto se debe a que el tipo declarado de x - el tipo con el que comenzó x - es una cadena | número , y la asignabilidad es siempre se compara con el tipo declarado.

Si le hubiésemos asignado un booleano al , habríamos visto un error ya que eso no era parte del declarado tipo x .

```
sea x = Math.random() < 0.5 ? 10 : "¡hola mundo!";
```

sea x: cadena | número

```
x = 1;
```

```
consola.log(x);
```

sea x: número

```
x = verdadero;
```

El tipo 'booleano' no se puede asignar al tipo 'cadena | número'.

```
consola.log(x);
```

sea x: cadena | número

Análisis de flujo de control

Hasta este punto, hemos visto algunos ejemplos básicos de cómo TypeScript se reduce dentro ramas específicas. Pero están sucediendo un poco más que simplemente caminar desde cada variable y buscando guardias de tipo en if s, while s, condicionales, etc. Por ejemplo

```
function padLeft(relleno: número | cadena, entrada: cadena) {
    if (tipo de relleno === "número") {
        devolver " ".repetir(relleno) + entrada;
    }
    relleno de retorno + entrada;
}
```

padLeft regresa desde dentro de su primer bloque if . TypeScript pudo analizar este código y ver que el resto del cuerpo (return padding + input;) está en el caso donde padding es un número . Como resultado, pudo eliminar el número del tipo de relleno (estrechamiento de cadena | number to string) para el resto de la función.

Este análisis de código basado en la accesibilidad se denomina análisis de flujo de control , y TypeScript usa esto análisis de flujo para tipos restringidos a medida que encuentra asignaciones y protecciones de tipo. Cuando una variable es

analizado, el flujo de control puede dividirse y volver a fusionarse una y otra vez, y se puede observar que esa variable tiene un tipo diferente en cada punto.

```
ejemplo de función () { let
  x: string | número | booleano;

  x = Matemáticas.aleatoria() < 0.5;

  consola.log(x);

    sea x: booleano

  if (Math.random() < 0.5) { x = "hola";
    consola.log(x);

      sea x: cadena

  } más { x
    = 100;
    consola.log(x);

      sea x: numero

  }

  devolver x;

    sea x: cadena | número

}
```

Uso de predicados de tipo

Hemos trabajado con construcciones de JavaScript existentes para manejar la reducción hasta el momento, sin embargo, a veces desea un control más directo sobre cómo cambian los tipos en su código.

Para definir un protector de tipo definido por el usuario, simplemente necesitamos definir una función cuyo tipo de retorno sea un escrito predicado :

```
función esPez(mascota: Pez | Pájaro): mascota es Pez { return
  (mascota como Pez).nadar !== indefinido;
}
```

pet is Fish es nuestro predicado de tipo en este ejemplo. Un predicado toma la forma nombre_parámetro donde nombre_parámetro debe ser el nombre de un parámetro de la función actual es Tipo , firma.

Cada vez que se llama a isFish con alguna variable, TypeScript angosto esa variable a ese específico escribirá si el tipo original es compatible.

```
// Ambas llamadas a 'nadar' y 'volar' ahora están bien. dejar
mascota = obtenerPequeñaMascota();

if (isFish(pet))
  { pet.swim(); } más
  { pet.fly();

}
```

Tenga en cuenta que TypeScript no solo sabe que pet es un pez en la rama if ; también sabe que en la rama else , por lo que debe tener un pájaro .

Puede usar el tipo de protección isFish para filtrar una matriz de Fish | Pájaro y obtener una serie de Pescado :

```
const zoológico: (Pez | Pájaro)[] = [obtenerPequeñaMascota(), obtenerPequeñaMascota(),
obtenerPequeñaMascota()] const bajoAgua1: Pez[] = zoo.filter(esPez); // o, equivalentemente const
bajoAgua2: Pez[] = zoo.filter(esPez) as Pez[];
```

```
// Es posible que sea necesario repetir el predicado para ejemplos más complejos
const bajo el agua3: Pez[] = zoo.filter((mascota): mascota es Pez => {
  if (pet.name === "tiburón") devuelve false; return
  esPez(mascota);});
```

Además, las clases pueden [usar this is Type](#) para reducir su tipo.

Sindicatos discriminados

La mayoría de los ejemplos que hemos visto hasta ahora se han centrado en reducir variables individuales con tipos simples como `string` y `number`. Si bien esto es útil para las formas más sencillas, en JavaScript trataremos con

Para motivarnos, imaginemos que estamos tratando de codificar formas como círculos y cuadrados. Los círculos llevan la cuenta de sus radios y los cuadrados llevan la cuenta de la longitud de sus lados. Usaremos un campo llamado `tipo` para indicar con qué forma estamos tratando. Aquí hay un primer intento de definir `Shape`.

```
interfaz Forma { tipo:  
  "círculo" | "cuadrado"; radio?:  
  número; sideLength?: número;  
}
```

Observe que estamos usando una unión de tipos de cadenas literales: "círculo" y "cuadrado" para decirnos si debemos tratar la forma como un círculo o un cuadrado respectivamente. Mediante el uso de "círculo" | "square" en lugar de `string`, podemos evitar errores de ortografía.

```
function handleShape(forma: Forma) { // ¡ups! if  
  (forma.tipo === "rect") {  
  
    devolverá 'falso' ya que los tipos "círculo" | "cuadrado" no se superponen.  
    Esta condición siempre se cumple porque  
    "rect" no se superponen.  
    // ...  
  }  
}
```

Podemos escribir una función `getArea` que aplique la lógica correcta en función de si se trata de un círculo o un cuadrado. Primero intentaremos tratar con círculos.

```
función getArea(forma: Forma) {
    devuelve Math.PI * forma.radio ** 2;
}

El objeto es posiblemente 'indefinido'.
```

Bajo [estrictos NullChecks](#) eso nos da un error, lo cual es apropiado ya que el radio podría no estar definido. Pero, ¿y si realizamos las comprobaciones oportunas sobre la propiedad kind ?

```
función getArea(forma: Forma) { if
    (forma.tipo === "círculo") {
        devuelve Math.PI * forma.radius ** 2;
}

El objeto es posiblemente 'indefinido'.
```

Hmm, TypeScript todavía no sabe qué hacer aquí. Hemos llegado a un punto en el que sabemos más sobre nuestros valores que el verificador de tipos. Podríamos intentar usar una afirmación no nula (un `!` después de `shape.radius`) para decir que el radio definitivamente está presente.

```
función getArea(forma: Forma) { if
    (forma.tipo === "círculo") {
        devuelve Math.PI * forma.radius! ** 2;
    }
}
```

Pero esto no se siente ideal. Tuvimos que gritar un poco al verificador de tipos con esas afirmaciones no nulas (`!`) para convencerlo de que se definió `shape.radius` , pero esas afirmaciones son propensas a errores si comenzamos a mover el código. Además, fuera de [los estrictos NullChecks](#) podemos acceder accidentalmente a cualquiera de esos campos de todos modos (ya que se supone que las propiedades opcionales siempre están presentes al leerlas). Definitivamente podemos hacerlo mejor.

El problema con esta codificación de Shape es que el verificador de tipos no tiene forma de saber si el radio o la longitud lateral están presentes o no en función de la propiedad kind . Necesitamos comunicar lo que sabemos al verificador de tipos. Con eso en mente, demos otro golpe para definir Shape .

```

interfaz Círculo { tipo:
  "círculo"; radio:
  número;
}

interfaz Cuadrado { tipo:
  "cuadrado"; longitud
  del lado: número;
}

tipo Forma = Círculo | Cuadrado;

```

Aquí, sepámos correctamente Shape out en dos tipos con diferentes valores para la propiedad kind , pero el radio y la longitud lateral se declaran como propiedades requeridas en sus respectivos tipos.

Veamos qué sucede aquí cuando tratamos de acceder al radio de una Forma .

```

función getArea(forma: Forma) {
  devuelve Math.PI * forma.radio ** 2;

  La propiedad 'radio' no existe en el tipo 'Forma'.
  La propiedad 'radio' no existe en el tipo 'Cuadrado'.

}

```

Al igual que con nuestra primera definición de Shape , esto sigue siendo un error. Cuando el radio era opcional, obtuvimos un error (con `strictNullChecks` habilitado) porque TypeScript no pudo determinar si la propiedad estaba presente. Ahora que Shape es una unión, TypeScript nos dice que shape podría ser un Square , ¡y los Square s no tienen un radio definido en ellos! Ambas interpretaciones son correctas, pero solo la codificación de unión de Shape provocará un error independientemente de cuán estrictas sean las comprobaciones nulas. está configurado.

Pero, ¿y si intentáramos comprobar de nuevo la propiedad kind ?

```

función getArea(forma: Forma) {
    if (forma.tipo === "círculo") {
        devuelve Math.PI * forma.radio ** 2;
    }
}

```

(parámetro) forma: Círculo

¡Eso eliminó el error! Cuando cada tipo en una unión contiene una propiedad común con tipos literales, TypeScript considera que es un y puede ~~sindicar discriminado~~ reducir los miembros de la Unión.

En este caso, ~~especie~~ era aquella propiedad común (que es lo que se considera una ~~propiedad~~ discriminante de forma). Verificar si la propiedad de tipo era "círculo" eliminó todos los tipos en Forma que no tenía una propiedad amable con el tipo "círculo". Esa forma estrechada hasta el tipo Círculo .

La misma verificación también funciona con declaraciones de cambio . Ahora podemos tratar de escribir nuestro completo ~~getArea sin molestias !~~ aserciones no nulas.

```

función getArea(forma: Forma) {
    cambiar (forma.tipo) {
        caso "círculo":
            devuelve Math.PI * forma.radio ** 2;
        (parámetro) forma: Círculo
        caso "cuadrado":
            volver forma.sideLength ** 2;
        (parámetro) forma: Cuadrado
    }
}

```

Lo importante aquí fue la codificación de Shape . Comunicar la información correcta a TypeScript: que Circle y Square eran en realidad dos tipos separados con campos de tipos específicos. fue crucial. Hacer eso nos permite escribir código TypeScript con seguridad de tipo que no se ve diferente al

JavaScript que habríamos escrito de otra manera. A partir de ahí, el sistema de tipos pudo hacer lo "correcto" y descubrir los tipos en cada rama de nuestra sentencia switch .

Aparte, intente jugar con el ejemplo anterior y elimine algunas de las palabras clave de retorno. Verá que la verificación de tipos puede ayudar a evitar errores cuando accidentalmente pasa por diferentes cláusulas en un declaración de cambio .

Las uniones discriminadas son útiles para algo más que hablar de círculos y cuadrados. Son buenos para representar cualquier tipo de esquema de mensajería en JavaScript, como enviar mensajes a través de la red (comunicación cliente/servidor) o codificar mutaciones en un marco de gestión de estado.

El tipo nunca

Al estrechar, puede reducir las opciones de una unión hasta un punto en el que haya eliminado todas las posibilidades y no quede nada. En esos casos, TypeScript usará un tipo nunca para representar un estado que no debería existir.

Comprobación de exhaustividad

El tipo nunca se puede asignar a todos los tipos; sin embargo, ningún tipo es assignable a never (excepto nunca a sí mismo). Esto significa que puede usar el estrechamiento y confiar en que nunca aparecerá para realizar una verificación exhaustiva en una declaración de cambio.

Por ejemplo, agregar un valor predeterminado a nuestra función getArea que intenta asignar la forma a nunca se generará cuando no se hayan manejado todos los casos posibles.

```
tipo Forma = Círculo | Cuadrado;

función getArea(forma: Forma) { interruptor
    (forma.tipo) { caso "círculo":
        return Math.PI * shape.radius ** 2;
    "square":
        volver forma.sideLength ** 2;
    defecto:
        const _exhaustiveCheck: nunca = forma; volver
        _comprobación exhaustiva;
    }
}
```

Agregar un nuevo miembro a la unión Shape provocará un error de TypeScript:

```
interfaz Triángulo { tipo:  
    "triángulo"; longitud del  
    lado: número;  
}  
  
tipo Forma = Círculo | cuadrado | Triángulo;  
  
función getArea(forma: Forma) { interruptor  
    (forma.tipo) { caso "círculo":  
  
        return Math.PI * shape.radius ** caso      2;  
        "square":  
            volver forma.sideLength ** por      2;  
            defecto:  
                const _exhaustiveCheck: nunca = forma;  
  
        volver _comprobación exhaustiva;  
    }  
}
```

El tipo 'Triángulo' no se puede asignar al tipo 'nunca'.

Más sobre funciones

Las funciones son el componente básico de cualquier aplicación, ya sean funciones locales, importadas de otro módulo o métodos en una clase. También son valores y, al igual que otros valores, TypeScript tiene muchas formas de describir cómo se pueden llamar las funciones. Aprendamos cómo escribir tipos que describen funciones.

Expresiones de tipo de función

La forma más sencilla de describir una función es con una tipo de expresión de función . Estos tipos son sintácticamente similar a las funciones de flecha:

```
función saludador(fn: (a: string) => void) { fn("Hola, Mundo");

}

función imprimirEnConsola(s: cadena)
  { consola.log(s);
}

saludador(imprimirEnConsola);
```

La sintaxis (a: string) => void significa "una función con un parámetro, llamada cadena, que no tiene un , de tipo valor de retorno". Al igual que con las declaraciones de funciones, si no se especifica un tipo de parámetro, implícitamente es any .

Tenga en cuenta que el nombre del parámetro es **obligatorio**. El tipo de función (cadena) => vacío significa "una función con un parámetro llamado cadena de tipo cualquiera ".

Por supuesto, podemos usar un alias de tipo para nombrar un tipo de función:

```
escriba GreetFunction = (a: string) => void; saludador
de funciones (fn: GreetFunction) {
  // ...
}
```

Firmas de llamada

En JavaScript, las funciones pueden tener propiedades además de ser invocables. Sin embargo, el tipo de función la sintaxis de expresión no permite declarar propiedades. Si queremos describir algo llamable con propiedades, podemos escribir a en [función de objeto](#):

```
tipo Función Describible = {
  descripción: cadena;
  (someArg: número): booleano;
};

function hacerAlgo(fn: FunciónDescribible) {
  consola.log(fn.descripción + + fn(6));  "devuelto"
}
```

Tenga en cuenta que la sintaxis es ligeramente diferente en comparación con una expresión de tipo de función: use : entre la lista de parámetros y el tipo de retorno en lugar de => .

Construir firmas

Las funciones de JavaScript también se pueden invocar con el operador new . TypeScript se refiere a estos como constructores porque por lo general crean un nuevo objeto. Puedes escribir un por firma de construcción agregando la nueva palabra clave delante de una firma de llamada:

```
escriba AlgunConstructor = {
  nuevo (s: cadena): SomeObject;
};

function fn(ctor: AlgunConstructor) {
  volver nuevo ctor("hola");
}
```

Algunos objetos, como el objeto Date de JavaScript , se pueden llamar con o sin new . Puedes combinar llame y construya firmas en el mismo tipo arbitrariamente:

```
interfaz CallOrConstruct {
  nuevo (s: cadena): Fecha;
  (n?: número): número;
}
```

Funciones genéricas

Es común escribir una función donde los tipos de entrada se relacionan con el tipo de salida, o donde los tipos de dos entradas están relacionados de alguna manera. Consideremos por un momento una función que devuelve el primer elemento de una matriz:

```
function primerElemento(arr: cualquiera[]) {
    retorno arr[0];
}
```

Esta función hace su trabajo, pero desafortunadamente tiene el tipo de retorno `any`. Sería mejor si la función devolvía el tipo del elemento de la matriz.

En mecanografiado, genéricos se utilizan cuando queremos describir una correspondencia entre dos valores. Hacemos esto declarando un parámetro de tipo en la firma de la función:

```
función firstElement<Tipo>(arr: Tipo[]): Tipo | indefinido {
    retorno arr[0];
}
```

Al agregar un parámetro de tipo Type a esta función y usarlo en dos lugares, hemos creado un enlace entre la entrada de la función (la matriz) y la salida (el valor de retorno). Ahora cuando lo llamamos, sale un tipo mas específico:

```
// s es de tipo 'cadena'
const s = primerElemento(["a", "b", "c"]);
// n es del tipo 'numero'
const n = primerElemento([1, 2, 3]);
// u es de tipo indefinido
const u = primerElemento([]);
```

Inferencia

Tenga en cuenta que no tuvimos que especificar `Tipo` en esta muestra. El tipo fue `inferido` -elegido elegido automáticamente - por TypeScript.

También podemos usar múltiples parámetros de tipo. Por ejemplo, una versión independiente del mapa se vería como esto:

```
función map<Entrada, Salida>(arr: Entrada[], func: (arg: Entrada) => Salida): O
    return arr.mapa(función);
}

// El parámetro 'n' es del tipo 'cadena'
// 'analizado' es del tipo 'número[]'
const analizado = map(["1", "2", "3"], (n) => parseInt(n));
```

Tenga en cuenta que, en este ejemplo, TypeScript podría inferir tanto el tipo del parámetro Tipo de entrada (de la matriz de cadenas dada), así como el parámetro Tipo de salida basado en el valor de retorno del expresión de función (número).

Restricciones

Hemos escrito algunas funciones genéricas que pueden funcionar para ningún tipo de valor A veces queremos relacionar dos valores, pero solo pueden operar en un determinado subconjunto de valores. En este caso, podemos utilizar un restricción para limitar los tipos de tipos que puede aceptar un parámetro de tipo.

Escribamos una función que devuelva el mayor de dos valores. Para hacer esto, necesitamos una propiedad de longitud . eso es un numero Nosotros constreñir el parámetro de tipo a ese tipo escribiendo una cláusula extends :

```

función más larga<Tipo se extiende { longitud: número }>(a: Tipo, b: Tipo) {
    if (a.longitud >= b.longitud) { return
        a; } más { devuelve b;

    }
}

// la matriz más larga es del tipo 'número []' const
la matriz más larga = la más larga ([1, 2], [1, 2, 3]); // cadena
más larga es del tipo 'alicia' | 'bob' const cadena más larga =
más larga("alicia", "bob"); // ¡Error! Los números no tienen una
propiedad de 'longitud' const notOK = más largo (10, 100);

```

El argumento de tipo 'número' no se puede asignar al parámetro de tipo '{longitud: número;}'.

Hay algunas cosas interesantes a tener en cuenta en este ejemplo. Permitimos que TypeScript inferir el regreso escriba más largo . La inferencia de tipo de retorno también funciona en funciones genéricas.

Debido a que restringimos Type a { length: number } , se nos permitió acceder a la propiedad .length de los parámetros a y b . Sin la restricción de tipo, no podríamos acceder a esas propiedades porque los valores podrían haber sido de otro tipo sin una propiedad de longitud.

Los tipos de longArray y longString se dedujeron en función de los argumentos.

¡Recuerde, los genéricos tienen que ver con relacionar dos o más valores con el mismo tipo!

Finalmente, tal como nos gustaría, la llamada a más larga (10, 100) se rechaza porque el tipo de número no tiene la propiedad .length .

Trabajar con valores restringidos

Aquí hay un error común cuando se trabaja con restricciones genéricas:

```
función longitud mínima<Tipo extiende { longitud: número }>(
    obj: tipo,
    mínimo: número
): Tipo { if
    (obj.longitud >= mínimo) { return obj; }
    else { return { longitud: mínimo }; }
```

Escriba '{longitud: número; }' no se puede asignar al tipo 'Tipo'.

asignable a la restricción de tipo 'Tipo', ~~por lo tanto 'arr': Tipo~~ '{longitud: número; }' es diferente
número; }'. ~~pero 'Tipo'~~ podría instanciarse con un subtipo diferente de restricción '{ longitud:

```
}
```

Puede parecer que esta función está bien: el tipo está restringido a {longitud: número} y la función devuelve el tipo o un valor que coincide con esa restricción. El problema es que la función promete devolver el tipo de objeto que se pasó, no solo el objeto que coincide con la restricción. ~~Este código fuera legal, podría escribir código que definitivamente no funcionaría:~~

```
// 'arr' obtiene valor { longitud: 6 } const arr =
longitud mínima ([1, 2, 3], 6); // y falla aquí porque las
matrices tienen // un método 'segmento', ¡pero no el
objeto devuelto! consola.log(arr.segmento(0));
```

Especificación de argumentos de tipo

TypeScript generalmente puede inferir los argumentos de tipo deseados en una llamada genérica, pero no siempre. Por ejemplo, supongamos que escribió una función para combinar dos matrices:

```
función combine<Tipo>(arr1: Tipo[], arr2: Tipo[]): Tipo[] { return arr1.concat(arr2);
}
```

Normalmente sería un error llamar a esta función con matrices que no coinciden:

```
const arr = combinar([1, 2, 3], ["hola"]);
```

El tipo 'cadena' no se puede asignar al tipo 'número'.

Sin embargo, si tenía la intención de hacer esto, podría especificar manualmente Type :

```
const arr = combinar <cadena | número>([1, 2, 3], ["hola"]);
```

Pautas para escribir buenas funciones genéricas

Escribir funciones genéricas es divertido y puede ser fácil dejarse llevar por los parámetros de tipo.

Tener demasiados parámetros de tipo o usar restricciones donde no son necesarios puede hacer que la inferencia sea menos exitosa y frustrar a las personas que llaman a su función.

Empuje los parámetros de tipo hacia abajo

Aquí hay dos formas de escribir una función que parecen similares:

```
function primerElemento1<Tipo>(matriz: Tipo[]) {
    retorno arr[0];
}

function firstElement2<Tipo extiende cualquier[]>(arr: Tipo) {
    retorno arr[0];
}

// a: numero (bien) const
a = primerElemento1([1, 2, 3]); // b: cualquier
(mala) const b = firstElement2([1, 2, 3]);
```

Estos pueden parecer idénticos a primera vista, pero firstElement1 es una forma mucho mejor de escribir esta función. Su tipo de retorno inferido es Type , pero el tipo de retorno inferido de firstElement2 es solo el tipo de este tipo de expresión de expresión para resolver el elemento durante una llamada.

Regla: cuando sea posible, use el parámetro de tipo en sí mismo en lugar de restringirlo

Utilice menos parámetros de tipo

Aquí hay otro par de funciones similares:

```
función filtro1<Tipo>(arr: Tipo[], func: (arg: Tipo) => booleano): Tipo[]
    return arr.filter(func);
}

function filter2<Tipo, Func extiende (arg: Tipo) => booleano>(
    matriz: Tipo[],
    función: función
): Tipo[] {
    return arr.filter(func);
}
```

Hemos creado un parámetro de tipo Func que . Eso siempre es una ~~referencia a valores de~~ roja, porque significa que las personas que llaman que desean especificar argumentos de tipo deben especificar manualmente un tipo adicional argumento sin razón. Func no hace nada más que hacer que la función sea más difícil de leer y razonar sobre!

Regla: utilice siempre la menor cantidad posible de parámetros de tipo

Los parámetros de tipo deben aparecer dos veces

A veces olvidamos que una función puede no necesitar ser genérica:

```
función saludar<Str extiende cadena>(s: Str) {
    consola.log("Hola, " + s);
}

saludar("mundo");
```

Podríamos haber escrito fácilmente una versión más simple:

```
función saludar(s: cadena) {
    consola.log("Hola, " + s);
}
```

Recuerde, los parámetros de tipo se usan para relacionarlos con tipos de valores múltiples · Si un parámetro de tipo es sólo una vez en la firma de la función, no están relacionados con nada.

Regla: si un parámetro de tipo solo aparece en una ubicación, considere si realmente lo necesita

Parámetros opcionales

Las funciones en JavaScript a menudo toman un número variable de argumentos. Por ejemplo, el `toFixed` método de número toma un conteo de dígitos opcional:

```
función f(n: número) {
    consola.log(n.toFixed()); // 0 argumentos
    consola.log(n.toFixed(3)); // 1 argumento
}
```

Podemos modelar esto en TypeScript marcando el parámetro como opcional con `? :`

```
función f(x?: número) {
    // ...
}
F(); // OK
f(10); // OK
```

Aunque el parámetro se especifica como número de tipo , el parámetro `x` en realidad tendrá el tipo `número | undefined` porque los parámetros no especificados en JavaScript obtienen el valor `undefined` .

También puede proporcionar un parámetro `defecto` :

```
función f(x = 10) {
    // ...
}
```

Ahora en el cuerpo de f , x tendrá un número de tipo porque cualquier argumento indefinido será reemplazado por 10 . Tenga cuenta que cuando un parámetro es opcional, las personas que llaman siempre pueden pasar , como indefinido , esto simplemente simula un argumento "faltante":

```
declarar función f(x?: número): void;
// Corte
// Todo bien
F();
f(10);
f(indefinido);
```

Parámetros opcionales en devoluciones de llamada

Una vez que haya aprendido acerca de los parámetros opcionales y las expresiones de tipo de función, es muy fácil cometer los siguientes errores al escribir funciones que invocan devoluciones de llamada:

```
function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {
    for (sea i = 0; i < arr.longitud; i++) {
        devolución de llamada (arr [i], i);
    }
}
```

¿Qué intención suele tener la gente al escribir un índice? como parámetro opcional es que quieren ambos de estos llamados a ser legales:

```
myForEach([1, 2, 3], (a) => console.log(a));
myForEach([1, 2, 3], (a, i) => console.log(a, i));
```

Que es esto Realmente significa que la **devolución de llamada** podría invocarse de dos maneras diferentes . En otras palabras, la definición de la función dice que la implementación podría verse así:

```
function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {
    for (let i = 0; i < arr.length; i++) {
        // No tengo ganas de proporcionar el índice hoy
        devolución de llamada (arr [i]);
    }
}
```

A su vez, TypeScript hará cumplir este significado y emitirá errores que en realidad no son posibles:

```
myForEach([1, 2, 3], (a, i) => {
    console.log(i.toFixed());
});
```

El objeto es posiblemente 'indefinido'.

En JavaScript, si llama a una función con más argumentos que parámetros, el extra simplemente se ignoran los argumentos. TypeScript se comporta de la misma manera. Funciones con menos parámetros (del mismo tipo) siempre puede tomar el lugar de funciones con más parámetros.

Al escribir un tipo de función para una devolución nunca escriba un parámetro opcional a menos que tenga la intención de llamar la de llamada, funciona sin pasar ese argumento

Sobrecargas de funciones

Algunas funciones de JavaScript se pueden llamar en una variedad de recuentos y tipos de argumentos. Por ejemplo, puede escribir una función para producir una Fecha que tome una marca de tiempo (un argumento) o una Especificación de mes/día/año (tres argumentos).

En TypeScript, podemos especificar una función que se puede llamar de diferentes maneras escribiendo sobrecarga firmas . Para hacer esto, escriba una cierta cantidad de firmas de función (generalmente dos o más), seguidas de el cuerpo de la función:

```

función hacerFecha(marca de tiempo: número): Fecha;
function hacerFecha(m: número, d: número, y: número): Fecha; function
makeDate(mOrTimestamp: número, d?: número, y?: número): Fecha {
    if (d !== indefinido && y !== indefinido) { return new
        Date(y, mOrTimestamp, d); } else { devuelve nueva
    fecha (mOrTimestamp);

}

} const d1 = hacerFecha(12345678);
const d2 = hacerFecha(5, 5, 5); const d3
= hacerFecha(1, 3);

```

espera 2 argumentos, pero existen sobrecargas que esperan ~~1 o 3 argumentos~~ Sin sobrecarga.

En este ejemplo, escribimos dos sobrecargas: una que acepta un argumento y otra que acepta tres argumentos. Estas dos primeras firmas se denominan sobrecargar firmas .

Luego, escribimos una implementación de función con una firma compatible. Las funciones tienen una implementación firma, pero esta firma no se puede llamar directamente. Aunque escribimos una función con dos parámetros opcionales después del requerido, ¡no se puede llamar con dos parámetros!

Firmas de sobrecarga y la firma de implementación

Esta es una fuente común de confusión. A menudo, las personas escriben código como este y no entienden por qué hay un error:

```

función fn(x: cadena): vacío; función
fn() { // ...
}

// Se espera poder llamar con cero argumentos fn();

```

Esperaba 1 argumentos, pero obtuvo 0.

Una vez más, la firma utilizada para escribir el cuerpo de la función no se puede "ver" desde el exterior.

La firma del que implementación no es visible desde el exterior. Al escribir una función sobrecargada, siempre debes tener dos o más firmas por encima de la implementación de la función.

La firma de implementación también debe estar ~~co~~ compatible de sobrecarga. Por ejemplo, estas funciones tienen errores porque la firma de implementación no coincide con las sobrecargas de forma correcta:

```
función fn(x: booleano): vacío;
// El tipo de argumento no es correcto
function fn(x: string): void;
```

sobrecarga no es compatible con su ~~firmado de implementación~~ firma. Esta firma de
función fn(x: booleano) {}

```
función fn(x: cadena): cadena;
// El tipo devuelto no es correcto
function fn(x: number): boolean;
```

sobrecarga no es compatible con su ~~firmado de implementación~~ firma. Esta firma de
function fn(x: cadena | número) { return "ups";
}

Escribir buenas sobrecargas

Al igual que los genéricos, hay algunas pautas que debe seguir al usar sobrecargas de funciones. Seguir estos principios hará que su función sea más fácil de llamar, más fácil de entender y más fácil de implementar.

Consideremos una función que devuelve la longitud de una cadena o una matriz:

```
función len(s: cadena): número; función
len(arr: cualquier[]): numero; function len(x:
cualquiera) { return x.longitud;
}
```

Esta función está bien; podemos invocarlo con cadenas o matrices. Sin embargo, no podemos invocarlo con un valor que podría ser una cadena o una matriz, porque TypeScript solo puede resolver una llamada de función a una sola

```
solamente(""); //  
OK solo ([0]); // OK  
solamente(Math.random() > 0.5 ? "hola" : [0]);
```

Ninguna sobrecarga coincide con esta llamada.

Sobrecarga 1 de 2, '(s: cadena): número', dio el siguiente error.

'número[] | "hola"' no se puede asignar a un parámetro Argumento de tipo 'cadena'.

El tipo 'número[]' no se puede asignar al tipo 'cadena'.

Sobrecarga 2 de 2, '(arr: any[]): número', dio el siguiente error.

'número[] | "hola"' no se puede asignar a un parámetro Argumento de tipo 'any[]'.

El tipo 'cadena' no se puede asignar al tipo 'any[]'.

Debido a que ambas sobrecargas tienen el mismo número de argumentos y el mismo tipo de devolución, podemos escribir una versión no sobrecargada de la función:

```
function len(x: cualquier[] | cadena) {  
    volver x.longitud;  
}
```

¡Esto es mucho mejor! Las personas que llaman pueden invocar esto con cualquier tipo de valor y, como beneficio adicional, no tenemos que averiguar una firma de implementación correcta.

Siempre prefiera parámetros con tipos de unión en lugar de sobrecargas cuando sea posible

Declarando esto en una función

TypeScript deducirá cuál debería ser esto en una función a través del análisis de flujo de código, por ejemplo, en lo siguiente:

```
const usuario =
  { id: 123,
    admin: falso, se
    convierte en administrador: función
    () { this.admin = true; }, };
```

TypeScript entiende que la función user.becomeAdmin tiene un correspondiente, que es el objeto externo user puede ser ~~señalado para más o menos basado en el tipo de dato representado en la especificación de JavaScript establece que no puede tener un parámetro llamado this que declare el tipo de this en el cuerpo de la función.~~, y así TypeScript usa ese espacio de sintaxis para permitirle

```
base de datos de interfaz {
  filterUsers(filtro: (este: Usuario) => booleano): Usuario[];
}

const db = getDB(); const
admins = db.filterUsers(función (este: Usuario) {
  devolver este.admin; });
```

Este patrón es común con las API de estilo de devolución de llamada, donde otro objeto normalmente controla cuándo se llama a su función. Tenga en cuenta que necesita usar la función y no las funciones de flecha para obtener este comportamiento:

```
base de datos de interfaz {
  filterUsers(filtro: (este: Usuario) => booleano): Usuario[];
}

const db = getDB(); const
administradores = db.filterUsers(() => this.admin);
```

La función de flecha contenedora captura el valor global de 'este'.

El elemento tiene implícitamente un tipo 'cualquiera' porque el tipo 'typeof globalThis' no tiene firma de índice. no tiene firma de índice.

Otros tipos que debe conocer

Hay algunos tipos adicionales que querrá reconocer que aparecen a menudo cuando trabaja con tipos de funciones. Como todos los tipos, puedes usarlos en todas partes, pero estos son especialmente relevantes en el contexto de funciones.

vacío

`void` representa el valor de retorno de las funciones que no devuelven un valor. Es el tipo inferido cualquiera vez que una función no tiene declaraciones de retorno, o no devuelve ningún valor explícito de esas declaraciones de retorno:

```
// El tipo de retorno inferido es nulo
función noop() {
    devolver;
}
```

En JavaScript, una función que no devuelve ningún valor devolverá implícitamente el valor `undefined`.

Sin embargo, `void` e `indefinido` no son lo mismo en TypeScript. Hay más detalles en el final de este capítulo.

`void` no es lo mismo que `indefinido`.

objeto

El objeto de tipo especial hace referencia a cualquier valor que no sea primitivo (`número`, `bigint`, `booleano`, `símbolo`, `nulo`, (cadena o indefinido)). Esto es diferente de la `objeto` tipo ({ }), y también diferente del tipo global `Object`. Es muy probable que nunca uses `Object`.

el objeto no es `Object`. ¡Utilice **siempre** el objeto !

Tenga en cuenta que en JavaScript, los valores de función son objetos: tienen propiedades, tienen `Object.prototype` en su cadena de prototipos, son una instancia de `Object`, puede llamar `Object.keys` en ellos, y así sucesivamente. Por esta razón, los tipos de función se consideran objetos en Mecanografiado.

desconocido

El tipo desconocido representa el valor. Esto es similar a cualquier tipo, pero es más seguro porque no es legal hacer nada con un valor desconocido :

```
función f1(a: cualquiera)
{ ab(); // OK

} función f2 ( a : desconocido )
{ ab ( ) ;

}

El objeto es de tipo 'desconocido'.
```

Esto es útil cuando se describen tipos de funciones porque puede describir funciones que aceptan cualquier valor sin tener ningún valor en el cuerpo de la función.

Por el contrario, puede describir una función que devuelve un valor de tipo desconocido:

```
función safeParse(s: cadena): desconocido {
    devuelve JSON.parse(s);
}

// ¡Hay que tener cuidado con 'obj'! const obj
= safeParse(someRandomString);
```

nunca

Algunas funciones nunca devolver un valor:

```
falla de la función (mensaje: cadena): nunca
{ lanzar un nuevo error (mensaje);
}
```

El tipo nunca representa valores que se observan. En un tipo de retorno, esto significa que la función lanza una excepción o finaliza la ejecución del programa.

never también aparece cuando TypeScript determina que no queda nada en una unión.

```
función fn(x: cadena | número) {
    if (tipo de x === "cadena") {
        // hacer algo
    } else if (tipo de x === "número") {
        // hacer algo más
    } más {
        X; // tiene tipo 'nunca'!
    }
}
```

Función

La función de tipo global describe propiedades como bind y otras presentes en llamar, aplicar, todos los valores de función en JavaScript. También tiene la propiedad especial de que los valores de tipo Función pueden ser llamado siempre; estas llamadas devuelven cualquiera:

```
función hacerAlgo(f: Función) {
    devolver f(1, 2, 3);
}
```

Esto es un llamada de función sin tipo y generalmente es mejor evitarlo debido a la inseguridad de cualquier devolución que escribe.

Si necesita aceptar una función arbitraria pero no pretende llamarla, el tipo () => void es generalmente más seguro.

Resto de parámetros y argumentos

Parámetros de descanso

Además de usar parámetros opcionales o sobrecargas para hacer funciones que pueden aceptar una variedad de conteos de argumentos fijos, podemos también definir funciones que toman un número ilimitado de argumentos usando

Lectura de fondo:

[Parámetros de descanso](#)

[Sintaxis extendida](#)

parámetros de descanso.

Un parámetro de descanso aparece después de todos los demás parámetros y utiliza la sintaxis ... :

```
función multiplicar(n: número, ...m: número[]) { return m.map((x)
    => n
        * x);

} // 'a' obtiene valor [10, 20, 30, 40] const a =
multiplicar (10, 1, 2, 3, 4);
```

En TypeScript, la anotación de tipo en estos parámetros es `any[]` en lugar de `any`, y cualquier anotación de tipo dada debe tener la forma `Array<T>` o `T[]`, o un tipo de tupla (que aprenderemos más adelante).

Resto de argumentos

A la inversa, podemos proveer un número variable de argumentos de una matriz utilizando la sintaxis de propagación. Por ejemplo, el método `push` de matrices toma cualquier número de argumentos:

```
const arr1 = [1, 2, 3]; const
arr2 = [4, 5, 6]; arr1.push(...arr2);
```

Tenga en cuenta que, en general, TypeScript no asume que las matrices son inmutables. Esto puede conducir a un comportamiento sorprendente:

```
// El tipo inferido es número[] -- "una matriz con cero o más números", // no específicamente
dos números const args = [8, 5]; const ángulo = Math.atan2(...args);
```

Un argumento de propagación debe tener un tipo de tupla o pasarse a un parámetro de descanso.

La mejor solución para esta situación depende un poco de su código, pero en general, un contexto `const` es la solución más sencilla:

```
// Inferido como tupla de 2 longitudes
const args = [8, 5] as const;
// OK

const ángulo = Math.atan2(...args);
```

El uso de argumentos de descanso puede requerir activar [downlevelIteration](#) cuando se dirige a tiempos de ejecución más antiguos.

Destrucción de parámetros

Puede usar la desestructuración de parámetros para desempaquetar convenientemente los objetos proporcionados como argumento en una o más variables locales en el cuerpo de la función. En JavaScript, se ve así:

Lectura de fondo:
[Asignación de desestructuración](#)

```
función suma({ a, b, c })
  { consola.log(a + b + c);

} suma({ a: 10, b: 3, c: 9 });
```

La anotación de tipo para el objeto va después de la sintaxis de desestructuración:

```
función suma({ a, b, c }: { a: número; b: número; c: número }) {
  consola.log(a + b + c);
}
```

Esto puede parecer un poco detallado, pero también puede usar un tipo con nombre aquí:

```
// Igual que el ejemplo anterior
type ABC = { a: number; b: número; c: número }; función
suma({ a, b, c }: ABC) { consola.log(a + b + c);

}
```

Asignabilidad de funciones

Tipo de devolución void

El tipo de retorno void para funciones puede producir un comportamiento inusual pero esperado.

La tipificación contextual con un tipo de retorno de vacío no obliga a las funciones a no devolver algo.

Otra forma de decir esto es un tipo de función contextual con un tipo de retorno nulo (tipo vf = () => vacío), cuando se implementa, puede devolver ningún otro valor, pero será ignorado.

Así, las siguientes implementaciones del tipo () => void son válidas:

```
escriba voidFunc = () => void;

const f1: voidFunc = () => { return
    true;
};

const f2: voidFunc = () => verdadero;

const f3: voidFunc = function () { return true; };
```

Y cuando el valor de retorno de una de estas funciones se asigna a otra variable, conservará el tipo de vacío :

```
constante v1 = f1();
constante v2 = f2();
constante v3 = f3();
```

Este comportamiento existe para que el código siguiente sea válido aunque Array.prototype.push devuelva un número y el método Array.prototype.forEach espere una función con un tipo de valor devuelto void .

```
const src = [1, 2, 3]; horario
constante = [0];

src.forEach((el) => dst.push(el));
```

Hay otro caso especial a tener en cuenta, cuando una definición de función literal tiene un tipo de retorno nulo , esa función **no** debe devolver nada.

```
function f2(): void { // @ts-
    expect-error return true;

}

const f3 = function (): void { // @ts-expect-
    error return true;

};
```

Para obtener más información sobre la anulación , consulte estas otras entradas de documentación:

- [manual v1](#)
- [manual v2](#)
- [Preguntas frecuentes: "¿Por qué las funciones que devuelven no nulas se pueden asignar a funciones que devuelven nulas?"](#)

Tipos de objetos

En JavaScript, la forma fundamental en que agrupamos y transmitimos datos es a través de objetos. En TypeScript, los representamos a través de tipos de objetos.

Como hemos visto, pueden ser anónimos:

```
function saludar(persona: { nombre: cadena; edad: número }) { return
    "Hola"
        + persona.nombre;
}
```

o pueden ser nombrados usando una interfaz

```
interface Persona
{
    nombre: cadena;
    edad: número;
}

function saludar(persona: Persona) { +
    devolver "Hola"
        persona.nombre;
}
```

o un alias de tipo.

```
tipo Persona =
{
    nombre: cadena;
    edad: número;
};

function saludar(persona: Persona) { +
    devolver "Hola"
        persona.nombre;
}
```

En los tres ejemplos anteriores, hemos escrito funciones que toman objetos que contienen el nombre de la propiedad (que debe ser una cadena) y la edad (que debe ser un número).

Modificadores de propiedad

Cada propiedad en un tipo de objeto puede especificar un par de cosas: el tipo, si la propiedad es opcional y si se puede escribir en la propiedad.

Propiedades opcionales

La mayor parte del tiempo, nos encontraremos tratando con objetos que tienen un **conjunto de propiedades**. En esos casos, podemos marcar esas propiedades agregando un signo de interrogación (?) al final de su nombre.

```
interfaz PaintOptions { forma:  
    Forma; xPos?: número;  
    yPos?: número;  
  
}  
  
función pintarForma(opciones: PaintOptions) {  
    // ...  
}  
  
const forma = obtenerForma();  
pintarForma({ forma });  
pintarForma({ forma, xPos: 100 });  
pintarForma({ forma, yPos: 100 });  
pintarForma({ forma, xPos: 100, yPos: 100 });
```

En este ejemplo, tanto xPos como yPos se consideran opcionales. Podemos optar por proporcionar cualquiera de ellos, por lo que todas las llamadas anteriores a paintShape son válidas. Toda la optionalidad realmente dice que si la propiedad se establece, es mejor que tenga un tipo específico.

También podemos leer de esas propiedades, pero cuando lo hacemos bajo [estrictos NullChecks](#), TypeScript nos dirá que son potencialmente indefinidos .

```
función pintarForma(opciones: PaintOptions) {  
    let xPos = opts.xPos; _____  
        (propiedad) PaintOptions.xPos?: número | indefinido  
  
    let yPos = opts.yPos; _____  
        (propiedad) PaintOptions.yPos?: número | indefinido  
  
    // ...  
}
```

En JavaScript, incluso si la propiedad nunca se ha establecido, aún podemos acceder a ella; solo nos dará el valor `undefined`. Solo podemos manejar `undefined` especialmente.

```
function paintShape(opts: PaintOptions) { let xPos =  
    opts.xPos === indefinido ? 0 : opciones.xPos;  
        sea xPos: número  
  
    let yPos = opts.yPos === indefinido ? 0 : opciones.yPos;  
        sea yPos: número  
  
    // ...  
}
```

Tenga en cuenta que este patrón de establecer valores predeterminados para valores no especificados es tan común que JavaScript tiene una sintaxis para admitirlo.

```
function pintarForma({ forma, xPos = 0, yPos = 0 }: Opciones de Pintura) {
    console.log(" coordenada x en", xPos);
    // ...
}
```

(parámetro) xPos: número

```
console.log(" coordenada y en", yPos);
```

(parámetro) yPos: número

Aquí usamos [un patrón de desestructuración](#) para el parámetro de paintShape y [valores predeterminados](#) proporcionados para xPos y yPos . Ahora xPos y yPos están definitivamente presentes dentro del cuerpo de pero son opcionales para cualquier persona que llame a paintShape . forma de pintura ,

Tenga en cuenta que actualmente no hay forma de colocar anotaciones de tipo dentro de los patrones de desestructuración. Esto se debe a que la siguiente sintaxis ya significa algo diferente en JavaScript.

```
function dibujar({ forma: Forma, xPos: numero = 100 /*...*/ }) { renderizar(forma);
```

No se puede encontrar el nombre 'forma'. ¿Querías decir 'forma'?

```
render(xPos);
```

No se puede encontrar el nombre 'xPos'.

```
}
```

En un patrón de desestructuración de objetos, forma: Forma significa "agarrar la propiedad forma y redefinirla localmente como una variable llamada Forma . Del mismo modo , xPos: número crea una variable llamada número cuyo valor se basa en los xPos del parámetro .

Usando [modificadores de mapeo](#), puede eliminar atributosopcionales .

Propiedades de solo lectura

Las propiedades también se pueden marcar como de solo lectura para TypeScript. Si bien no cambiará ningún comportamiento en tiempo de ejecución, una propiedad marcada como de solo lectura no se puede escribir durante la verificación de tipos.

```

interfaz AlgúnTipo {
    acceso de solo lectura : cadena;
}

function doSomething(obj: SomeType) { // Podemos
    leer desde 'obj.prop'. console.log(`prop tiene el
    valor '${obj.prop}'.`);

    // Pero no podemos reasignarlo. obj.prop
    = "hola";

```

No se puede asignar a 'prop' porque es una propiedad de solo lectura.

```
}
```

El uso del modificador de solo lectura no implica necesariamente que un valor sea totalmente inmutable o, en otras palabras, que su contenido interno no se pueda cambiar. Simplemente significa que la propiedad en sí no se puede volver a escribir.

```

interfaz Inicio {
    residente de solo lectura : { nombre: cadena; edad: numero };
}

función visitaParaCumpleaños(casa: Casa) {
    // Podemos leer y actualizar propiedades desde 'home.resident'. console.log(`¡Feliz
    cumpleaños ${home.resident.name}!`); domicilio.residente.edad++;

}

función desalojo (casa: Casa) {
    // Pero no podemos escribir en la propiedad 'residente' en sí misma en un 'Inicio'.
    domicilio.residente = {

        nombre: "Victor the Evictor", edad:
        42, };

}

```

No se puede asignar a 'residente' porque es una propiedad de solo lectura.

Es importante gestionar las expectativas de lo que implica solo lectura . Es útil señalar la intención durante el tiempo de desarrollo de TypeScript sobre cómo se debe usar un objeto. TypeScript no tiene en cuenta

si las propiedades de dos tipos son de solo lectura al verificar si esos tipos son compatibles, por lo que las propiedades de solo lectura también pueden cambiar a través de alias.

```
interfaz Persona
{ nombre: cadena;
edad: número;
}

interfaz ReadonlyPerson { nombre
de solo lectura : cadena; edad
de solo lectura : número;
}

let WritablePerson: Person = { nombre:
"Persona McPersonface", edad: 42, };

// funciona
let readonlyPerson: ReadonlyPerson = writeablePerson;

console.log(solecturaPersona.edad); // imprime '42'
personaescribible.edad++; console.log(solecturaPersona.edad); // 
imprime '43'
```

Usando [modificadores de mapeo](#), puede eliminar los atributos de solo lectura .

índice de firmas

A veces, no conoce todos los nombres de las propiedades de un tipo de antemano, pero sí conoce la forma de los valores.

En esos casos, puede usar una firma de índice para describir los tipos de valores posibles, por ejemplo:

```
interfaz StringArray {
    [índice: número]: cadena;
}

const miArray: StringArray = getStringArray(); const
segundoElemento = miArray[1];
```

const segundo elemento: cadena

Arriba, tenemos una interfaz StringArray que tiene una firma de índice. Esta firma de índice establece que cuando un StringArray se indexa con un número , devolverá una cadena .

Un tipo de propiedad de firma de índice debe ser 'cadena' o 'número'.

- Es posible admitir ambos tipos de indexadores...

Si bien las firmas de índice de cadena son una forma poderosa de describir el patrón de "diccionario", también exigen que todas las propiedades coincidan con su tipo de retorno. Esto se debe a que un índice de cadena declara que obj.property también está disponible como obj["property"]. En el siguiente ejemplo, el tipo de nombre no coincide con el tipo de índice de cadena y el verificador de tipo da un error:

```
interfaz NumberDictionary { [índice:
    cadena]: número;

    longitud: número; // ok
    nombre: cadena;

    'nombre' del tipo 'cadena' no se puede asignar al índice 'cadena' del tipo 'número'. propiedad número'.
}
```

Sin embargo, las propiedades de diferentes tipos son aceptables si la firma del índice es una unión de los tipos de propiedad:

```
interfaz NumberOrStringDictionary { [índice:  
    cadena]: número | cuerda; longitud: número; //  
    ok, la longitud es un nombre numérico: string; // ok, el  
    nombre es una cadena  
}
```

Finalmente, puede hacer que las firmas de índice sean de solo lectura para evitar la asignación a sus índices:

```
interfaz ReadonlyStringArray {  
    solo lectura [índice: número]: cadena;  
}
```

```
let myArray: ReadonlyStringArray = getReadOnlyStringArray(); miArray[2] =  
"Mallory";
```

La firma de índice en el tipo 'ReadonlyStringArray' solo permite la lectura.

No puede establecer myArray[2] porque la firma del índice es de solo lectura .

Tipos de extensión

Es bastante común tener tipos que pueden ser versiones más específicas de otros tipos. Por ejemplo, podríamos tener un tipo de dirección básica que describa los campos necesarios para enviar cartas y paquetes en EE. UU.

```
interfaz BasicAddress { nombre?:  
    cadena; calle: cadena; ciudad:  
    cadena; país: cadena; código  
    postal: cadena;  
}
```

En algunas situaciones, eso es suficiente, pero las direcciones a menudo tienen un número de unidad asociado si el edificio en una dirección tiene varias unidades. Entonces podemos describir una DirecciónConUnidad .

```
interfaz DirecciónConUnidad {
    nombre?: cadena;
    unidad: cadena;
    calle: cadena;
    ciudad: cadena;
    país: cadena; código
    postal: cadena;
}
```

Esto hace el trabajo, pero la desventaja aquí es que tuvimos que repetir todos los demás campos de BasicAddress cuando nuestros cambios eran puramente aditivos. En su lugar, podemos extender el tipo BasicAddress original y simplemente agregar los nuevos campos que son exclusivos de AddressWithUnit .

```
interfaz BasicAddress { nombre?:
    cadena; calle: cadena; ciudad:
    cadena; país: cadena; código
    postal: cadena;
```

```
}
```

```
interfaz AddressWithUnit extiende BasicAddress { unidad: cadena;
```

```
}
```

La palabra clave extends en una interfaz nos permite copiar efectivamente miembros de otros tipos con nombre y agregar los miembros nuevos que queramos. Esto puede ser útil para reducir la cantidad de declaraciones repetitivas de tipo que tenemos que escribir y para señalar la intención de que varias declaraciones diferentes de la misma propiedad puedan estar relacionadas. Por ejemplo, AddressWithUnit no necesitaba repetir la propiedad street , y debido a que street se origina en BasicAddress , el lector sabrá que esos dos tipos , están relacionados de alguna manera.

Las interfaces también pueden extenderse desde múltiples tipos.

```

interfaz Colorido { color:
  cadena;
}

círculo de interfaz
{ radio: número;
}

interfaz ColorfulCircle extiende Colorful, Circle {}

const cc: ColorfulCircle = { color:
  "rojo", radio: 42, };

```

Tipos de intersección

Las interfaces nos permitieron construir nuevos tipos a partir de otros tipos al extenderlos. TypeScript proporciona otra construcción llamada tipos de intersección que se utiliza principalmente para combinar objetos existentes

Un tipo de intersección se define mediante el operador & .

```

interfaz Colorido { color:
  cadena;

} interfaz Círculo { radio:
  número;
}

type ColorfulCircle = Colorful & Circle;

```

Aquí, hemos cruzado Colorful y Circle para producir un nuevo tipo que tiene todos los miembros de Vistoso y Circulo .

```

function dibujar (círculo: colorido y círculo) { console.log ('El
color era ${circle.color}); console.log('El radio era $
{circle.radius});
}

// está
bien dibujar ({ color: "azul", radio: 42 });

// vaya
dibujar ({ color: "rojo", raidus: 42 });

raidus: número; } no se puede asignar a un parámetro de tipo 'Color' Arg y no coincide con el tipo '{ color: string}' o 'Colorido y Círculo'.
solo puede especificar propiedades conocidas, pero 'raidus' no existe en el tipo 'Color' Colorido y circular'. ¿Querías escribir 'radio'? no existe en el tipo 'Colorido y circular'. ¿Querías escribir 'radio'?

```

Interfaces frente a intersecciones

Acabamos de ver dos formas de combinar tipos que son similares, pero que en realidad son sutilmente diferentes. Con las interfaces, podríamos usar una cláusula extends para extendernos desde otros tipos, y pudimos hacer algo similar con las intersecciones y nombrar el resultado con un alias de tipo. La principal diferencia entre los dos es cómo se manejan los conflictos, y esa diferencia suele ser una de las razones principales por las que elegiría uno sobre el otro entre una interfaz y un alias de tipo de un tipo de intersección.

Tipos de objetos genéricos

Imaginemos un tipo de cuadro que puede contener cualquier valor: `cadenas` , `números` , `jirafas` , lo que sea.

```

caja de interfaz
{ contenido: cualquiera;
}
```

En este momento, la propiedad de contenido se escribe como `any` , lo que funciona, pero puede provocar accidentes en el futuro.

En su lugar, podríamos usar `desconocido` , pero eso significaría que, en los casos en los que ya conocemos el tipo, tendríamos de contenidos , que hacer comprobaciones preventivas o utilizar aserciones de tipo propensas a errores.

```
caja de interfaz
{ contenido: desconocido;
}

let x: Box =
{ contenidos: "hola mundo", };

// podríamos verificar 'x.contents' if
(typeof x.contents === "string") {
    consola.log(x.contents.toLowerCase());
}

// o podríamos usar una aserción de tipo
console.log((x.contents as string).toLowerCase());
```

Un tipo de enfoque seguro sería, en su lugar, crear diferentes tipos de cajas para cada tipo de contenido .

```
interfaz NumberBox
{ contenido: número;
}

interfaz StringBox { contenido:
    cadena;
}

interfaz BooleanBox
{ contenido: booleano;
}
```

Pero eso significa que tendremos que crear diferentes funciones, o sobrecargas de funciones, para operar en estos tipos.

```
function setContents(box: StringBox, newContents: string): void;
function setContents(box: NumberBox, newContents: number): void;
function setContents(box: BooleanBox, newContents: boolean): void;
function setContents(caja: { contenido: cualquiera }, nuevoContenido: cualquiera) {
    box.contents = newContents;
}
```

Eso es mucho repetitivo. Además, más adelante podríamos necesitar introducir nuevos tipos y sobrecargas. Este es frustrante, ya que nuestros tipos de cajas y sobrecargas son todos iguales.

En su lugar, podemos hacer un genérico **Tipo de casilla que declara un parámetro de tipo**.

```
Caja de interfaz <Tipo> {
    contenido: Tipo;
}
```

Puede leer esto como "Una caja de tipo es algo cuyo contenido tiene tipo Tipo".

Más adelante, cuando nos referimos a **Box<string>**, que **dice** tipo de argumento

```
let box: Box<cadena>;
```

Piense en **Box** como una plantilla para un tipo real, donde **Type** es un marcador de posición que será reemplazado por algún otro tipo. Cuando TypeScript ve **Box<string>**, reemplazará cada instancia de **Type** en

Box<Type> con **string**, y termina trabajando con algo como **{contents: string}**.

En otras palabras, **Box<string>** y nuestro **StringBox** anterior funcionan de manera idéntica.

```
interface Box<Tipo>
{ contenido: Tipo;

} interfaz StringBox
{ contenido: cadena;
}

let boxA: Box<string> = { contenidos: "hola" }; cajaA.contenido;
```

(propiedad) Box<cadena>.contents: cadena

```
let boxB: StringBox = { contenidos: "mundo" }; cajaB.contenido;
```

(propiedad) StringBox.contents: cadena

La caja es reutilizable porque el tipo se puede sustituir por cualquier cosa. Eso significa que cuando necesitamos un cuadro para un nuevo tipo, no necesitamos declarar un nuevo tipo de cuadro en absoluto (aunque ciertamente podríamos hacerlo si quisieramos).

```
interface Box<Tipo>
{ contenido: Tipo;
}

interfaz Apple { // ....
}

// Igual que '{ contenidos: Apple }'. escriba
AppleBox = Box<Apple>;
```

Esto también significa que podemos evitar sobrecargas por completo utilizando [funciones genéricas](#).

```
function setContents<Tipo>(caja: Caja<Tipo>, nuevoContenido: Tipo) { caja.contenido
= nuevoContenido;
}
```

Vale la pena señalar que los alias de tipo también pueden ser genéricos. Podríamos haber definido nuestra nueva interfaz Box<Type>, que era:

```
interface Box<Tipo>
{ contenido: Tipo;
}
```

utilizando un alias de tipo en su lugar:

```
tipo Box<Tipo> =
{ contenido: Tipo; };
```

Dado que los alias de tipo, a diferencia de las interfaces, pueden describir más que solo tipos de objetos, también podemos usarlos para escribir otros tipos de tipos auxiliares genéricos.

```
tipo OrNull<Tipo> = Tipo | nulo;
```

```
tipo OneOrMany<Tipo> = Tipo | Escribe[];
```

```
type OneOrManyOrNull<Tipo> = OrNull<OneOrMany<Tipo>>;
```

```
tipo OneOrManyOrNull<Tipo> = OneOrMany<Tipo> | nulo
```

```
escriba OneOrManyOrNullStrings = OneOrManyOrNull<cadena>;
```

```
escriba OneOrManyOrNullStrings = OneOrMany<cadena> | nulo
```

Regresaremos para escribir alias en un momento.

El tipo de matriz

Los tipos de objetos genéricos suelen ser algún tipo de tipo de contenedor que funciona independientemente del tipo de elementos que contienen. Es ideal que las estructuras de datos funcionen de esta manera para que puedan reutilizarse en diferentes tipos de datos.

Resulta que hemos estado trabajando con un tipo como ese a lo largo de este manual: el Array escribe. Cada vez que escribimos tipos como `number[]` o `string[]`, en realidad es solo una forma abreviada de `Array<número>` y `Array<cadena>`.

```
function hacerAlgo(valor: Array<cadena>) {
    // ...
}

let myArray: string[] = ["hola", "mundo"];

// ¡Cualquiera de estos funciona!
hacerAlgo(miArray);
hacerAlgo(nuevo Array("hola", "mundo"));
```

Al igual que el tipo Box anterior, Array en sí mismo es un tipo genérico.

```
matriz de interfaz <Tipo> {
    /**
     * Obtiene o establece la longitud de la matriz.
     */
    longitud: número;

    /**
     * Elimina el último elemento de una matriz y lo devuelve.
     */
    pop(): Escriba | indefinido;

    /**
     * Agrega nuevos elementos a una matriz y devuelve la nueva longitud de a
     */
    empujar (... elementos: Tipo[]): número;

    // ...
}
```

JavaScript moderno también proporciona otras estructuras de datos que son genéricas, como `Map<K, V>`, `Establecer<T>`, `V>` y `Promise<T>`. Todo esto realmente significa que debido a cómo `Map`, `Establecer`, y `promesa` se comportan, pueden trabajar con cualquier conjunto de tipos.

El tipo ReadonlyArray

ReadonlyArray es un tipo especial que describe matrices que no deben cambiarse.

```
function doStuff(values: ReadonlyArray<string>) { // Podemos leer
    desde 'valores'... const copy = valores.slice(); console.log('El
    primer valor es ${valores[0]}`);

    // ...pero no podemos mutar 'valores'.
    valores.push("¡hola!");

    La propiedad 'push' no existe en el tipo 'readonly string[]'.
}
```

Al igual que el modificador de solo lectura para las propiedades, es principalmente una herramienta que podemos usar para la intención. Cuando vemos una función que devuelve ReadonlyArray s, nos dice que no debemos cambiar el contenido en absoluto, y cuando vemos una función que consume ReadonlyArray s, nos dice que podemos pasar cualquier matriz a esa función sin preocuparnos. que cambiará su contenido.

A diferencia de Array , no hay un constructor ReadonlyArray que podamos usar.

```
nuevo ReadonlyArray("rojo", "verde", "azul");
a un tipo, pero se usa como valor aquí. ya que se usa como valor 'ReadonlyArray' solo se refiere
```

En su lugar, podemos asignar Array regulares a ReadonlyArray s .

```
const roArray: ReadonlyArray<cadena> = ["rojo", "verde", "azul"];
```

Así como TypeScript proporciona una sintaxis abreviada para Array<Type> con Type[] , también proporciona una sintaxis abreviada para ReadonlyArray<Type> con readonly Type[] .

```
function hacerCosas(valores: cadena de solo lectura []) {
    // Podemos leer de 'valores'...
    copia const = valores.slice(); console.log(`El
    primer valor es ${valores[0]}`);
    // ...pero no podemos mutar 'valores'.
    valores.push("¡hola!");
}

La propiedad 'push' no existe en el tipo 'readonly string[]'.
```

}

Una última cosa a tener en cuenta es que, a diferencia del modificador de propiedad de solo lectura , la capacidad de asignación no es bidireccional entre los Array normales y los ReadonlyArray .

```
let x: cadena de solo lectura [] = []; sea y: cadena[]
= [];

x = y;
y = x;

asignar al tipo mutable 'string[]'.let tipo mutable string[] = y; Ignorar a El tipo 'readonly string[]' es 'readonly' y no se puede
```

Tipos de tuplas

A tipo tupla es otro tipo de tipo Array que sabe exactamente cuántos elementos contiene y exactamente qué tipos contiene en posiciones específicas.

```
escriba StringNumberPair = [cadena, número];
```

Aquí, StringNumberPair es un tipo de tupla de cadena y número . Al igual que ReadonlyArray , no tiene representación en tiempo de ejecución, pero es importante para TypeScript. Para el sistema de tipos, StringNumberPair describe matrices cuyo índice 0 contiene una cadena y cuyo índice 1 contiene un número

```
function hacerAlgo(par: [cadena, número]) { const a = par[0];
  constante a: cadena
  const b = par[1];
  constante b: número
  // ...
}

hacerAlgo(["hola", 42]);
```

Si tratamos de indexar más allá del número de elementos, obtendremos un error.

```
function hacerAlgo(par: [cadena, número]) {
  // ...

  const c = par[2];
  tupla '[cadena, número]' de longitud '2' no tiene elemento en el índice '2'. índice El tipo de
}


```

También podemos [desestructurar tuplas](#) utilizando la desestructuración de matrices de JavaScript.

```
function hacerAlgo(cadenaHash: [cadena, número]) {
  const [inputString, hash] = stringHash;
  consola.log(cadena de entrada);
  cadena de entrada const: cadena
  consola.log(hash);
  hash const: número
}
```

Los tipos de tupla son útiles en API muy basadas en convenciones, donde el significado de cada elemento es "obvio".

Esto nos da flexibilidad en el nombre que queramos para nuestras variables cuando las desestructuramos. En el ejemplo anterior, pudimos nombrar los elementos 0 y 1 como quisieramos.

Sin embargo, dado que no todos los usuarios tienen la misma visión de lo que es obvio, puede valer la pena reconsiderar si el uso de objetos con nombres de propiedades descriptivos puede ser mejor para su API.

Además de esas comprobaciones de longitud, los tipos de tupla simples como estos son equivalentes a los tipos que son **versiones** de Array s que declaran propiedades para índices específicos y que declaran la longitud con un tipo literal numérico.

```
interface StringNumberPair { // longitud
  de propiedades especializadas: 2;
  0: cadena; 1: número;

  // Otro 'Array<cadena | número>' miembros... sector(inicio?:
  // número, final?: número): Array<cadena | número>;
}
```

Otra cosa que le puede interesar es que las tuplas pueden tener propiedades opcionales al escribir un signo de interrogación (? después del tipo de un elemento). Los elementos de tuplaopcionales solo pueden aparecer al final y también afectan el tipo de longitud .

```
escriba O2dO3d = [número, número, ¿número?];
```

```
function setCoordinate(coord: O2dOr3d) {
  constante [x, y, z] = coord;
```

constante z: número | indefinido

console.log(` Las coordenadas proporcionadas tenían dimensiones de \${coord.length} `);

(propiedad) longitud: 2 | 3

```
}
```

Las tuplas también pueden tener elementos de descanso, que deben ser de tipo matriz/tupla.

```
tipo StringNumberBooleans = [cadena, número, ...booleano[]]; escriba
StringBooleansNumber = [cadena, ...booleano[], número]; type
BooleansStringNumber = [...booleano[], cadena, número];
```

- `StringNumberBooleans` describe una tupla cuyos primeros dos elementos son cadena y número respectivamente, pero que puede tener cualquier número de booleanos siguientes.
- `StringBooleansNumber` describe una tupla cuyo primer elemento es una cadena y luego cualquier número de valores booleanos y termina con un número .
- `BooleansStringNumber` describe una tupla cuyos elementos iniciales son cualquier número de valores booleanos y terminan con una cadena y luego un número

Una tupla con un elemento de descanso no tiene una "longitud" establecida; solo tiene un conjunto de elementos conocidos en diferentes posiciones.

```
const a: StringNumberBooleans = ["hola", 1]; const b:
StringNumberBooleans = ["hermoso", 2, verdadero]; const c:
StringNumberBooleans = ["mundo", 3, verdadero, falso, verdadero, falso , verdadero
```

¿Por qué podrían ser útiles los elementos opcionales y de descanso? Bueno, permite que TypeScript corresponda tuplas con listas de parámetros. Los tipos de tuplas se pueden usar en [parámetros y argumentos de descanso](#), para que lo siguiente:

```
function readButtonInput(...argumentos: [cadena, número, ...booleano[]]) {
  const [nombre, versión, ... entrada] = args; // ...
}
```

es básicamente equivalente a:

```
function readButtonInput(nombre: cadena, versión: número, ...entrada: booleano[
  // ...
])
```

Esto es útil cuando desea tomar una cantidad variable de argumentos con un parámetro de descanso y necesita una cantidad mínima de elementos, pero no desea introducir variables intermedias.

Tipos de tupla de solo lectura

Una nota final sobre los tipos de tupla: los tipos de tupla tienen variantes de solo lectura y se pueden especificar colocando un modificador de solo lectura delante de ellos, al igual que con la sintaxis abreviada de matriz.

```
function hacerAlgo(par: solo lectura [cadena, número]) {  
    // ...  
}
```

Como era de esperar, no se permite escribir en ninguna propiedad de una tupla de solo lectura en TypeScript.

```
function hacerAlgo(par: solo lectura [cadena, número]) {  
    par[0] = "¡hola!";
```

No se puede asignar a '0' porque es una propiedad de solo lectura.

```
}
```

Las tuplas tienden a crearse y dejarse sin modificar en la mayoría de los códigos, por lo que anotar los tipos como tuplas de solo lectura cuando sea posible es una buena opción predeterminada. Esto también es importante dado que los literales de matriz con aserciones constantes se deducirán con tipos de tupla de solo lectura .

```
let point = [3, 4] as const;  
  
función distanciaDesdeOrigin([x, y]: [número, número]) { ** 2); 2 + y  
    devuelve Math.sqrt(x **  
}
```

```
distanciaDesdeOrigen(punto);
```

de tipo 'solo lectura [3, 4]' no se puede asignar al parámetro de tipo '[número, El tipo solo lectura [número, número].

'4]' es 'solo lectura' y no se puede asignar al tipo mutable '[número, El tipo solo lectura [3,4] es 'solo lectura' y no se puede asignar al tipo mutable '[número, número].

Aquí, `distanceFromOrigin` nunca modifica sus elementos, pero espera una tupla mutable. Ya que el tipo de punto se infirió como de solo lectura `[3, 4]`, no será compatible con `[número, número]` ya que ese tipo no puede garantizar que los elementos del punto no sean mutados.

Crear tipos a partir de tipos

El sistema de tipos de TypeScript es muy poderoso porque permite expresar tipos en términos de tipos.

en términos

La forma más simple de esta idea son los genéricos, en realidad tenemos una amplia variedad de operadores de tipos disponibles para usar. También es posible expresar tipos en términos de lo que ya tenemos.

valores

Al combinar varios tipos de operadores, podemos expresar operaciones y valores complejos de forma sucinta y manera mantenible. En esta sección, cubriremos formas de expresar un nuevo tipo en términos de un tipo existente o valor.

- [Genéricos](#) - Tipos que toman parámetros
- [Operador de tipo Keyof](#) - Usando el operador keyof para crear nuevos tipos
- [Tipo de operador de tipo](#) - Usando el operador typeof para crear nuevos tipos
- [Tipos de acceso indexado](#) - Uso de la sintaxis Type['a'] para acceder a un subconjunto de un tipo
- [Tipos condicionales](#) - Tipos que actúan como sentencias if en el sistema de tipos
- [Tipos asignados](#) - Crear tipos mapeando cada propiedad en un tipo existente
- [Tipos de literales de plantilla](#) - Tipos asignados que cambian propiedades a través de cadenas literales de plantilla

Genéricos

Una parte importante de la ingeniería de software consiste en crear componentes que no solo tienen componentes bien definidos y API consistentes, pero también son reutilizables. Componentes que son capaces de trabajar con los datos de hoy así como los datos del mañana le darán las capacidades más flexibles para construir grandes sistemas de software

En lenguajes como C# y Java, una de las herramientas principales en la caja de herramientas para crear reutilizables es decir, ser ~~capaz de~~ genéricos: un componente que pueda funcionar en una variedad de tipos en lugar de uno solo. Esto permite a los usuarios consumir estos componentes y usar sus propios tipos

Hola mundo de los genéricos

Para empezar, hagamos el "hola mundo" de los genéricos: la función de identidad. La función identidad es una función que devolverá todo lo que se pasa. Puede pensar en esto de una manera similar a la comando de eco .

Sin genéricos, tendríamos que darle a la función de identidad un tipo específico:

```
función identidad(arg: número): número {  
    devolver argumento;  
}
```

O bien, podríamos describir la función de identidad usando cualquier tipo:

```
función identidad(arg: cualquiera): cualquiera {  
    devolver argumento;  
}
```

Si bien el uso de `any` es ciertamente genérico, ya que hará que la función acepte todos y cada uno de los tipos para el tipo de argumento , en realidad estamos perdiendo la información sobre qué tipo era cuando la función devoluciones. Si pasamos un número, la única información que tenemos es que se puede devolver cualquier tipo.

En cambio, necesitamos una forma de capturar el tipo de argumento de tal manera que también podamos usarlo para indicar lo que se devuelve. Aquí, usaremos un tipo especializado `returnable` que

trabaja en tipos en lugar de valores.

```
función identidad<Tipo>(arg: Tipo): Tipo { return arg;
}
```

Ahora hemos agregado una variable de tipo Tipo a la función de identidad. Este tipo nos permite capturar el tipo que proporciona el usuario (por ejemplo, número), para que podamos usar esa información más adelante. Aquí, usamos Type nuevamente como el tipo de devolución. En la inspección, ahora podemos ver que se usa el mismo tipo para el argumento y el tipo de retorno. Esto nos permite traficar ese tipo de información en un lado de la función y fuera del otro.

Decimos que esta versión de la función de identidad es genérica, ya que funciona en una variedad de tipos.

A diferencia del uso de any, también es tan preciso (es decir, no pierde ninguna información) como la primera función de identidad que usaba números para el argumento y el tipo de retorno.

Una vez que hemos escrito la función de identidad genérica, podemos llamarla de dos maneras. La primera forma es pasar todos los argumentos, incluido el argumento de tipo, a la función:

```
let salida = identidad<cadena>("miCadena");
```

dejar salida: cadena

Aquí establecemos explícitamente que Type sea una cadena como uno de los argumentos de la llamada a la función, denotada usando <> alrededor de los argumentos en lugar de () .

La segunda forma también es quizás la más común. Aquí usamos, es decir, queremos que el compilador establezca el valor de Tipo para nosotros automáticamente en función del tipo de argumento que pasamos:

```
let salida = identidad("miCadena");
```

dejar salida: cadena

Tenga en cuenta que no tuvimos que pasar explícitamente el tipo entre corchetes angulares (<>); el compilador simplemente miró el valor "myString" y estableció Type en su tipo. Si bien la inferencia de argumentos de tipo puede ser una

herramienta útil para mantener el código más corto y más legible, es posible que deba pasar explícitamente los argumentos de tipo como hicimos en el ejemplo anterior cuando el compilador no puede inferir el tipo, como puede suceder en ejemplos más complejos.

Trabajar con variables de tipo genérico

Cuando comience a usar genéricos, notará que cuando crea funciones genéricas, como el compilador, correctamente ~~Es decir, que en la vida para estos tipos de datos se tratarán de ser de tipo~~, la función

Tomemos nuestra función de identidad de antes:

```
función identidad<Tipo>(arg: Tipo): Tipo { return arg;  
}
```

¿Qué pasa si también queremos registrar la longitud del argumento `arg` en la consola con cada llamada?

Podríamos tener la tentación de escribir esto:

```
función loggingIdentity<Tipo>(arg: Tipo): Tipo {  
    consola.log(arg.longitud);
```

La propiedad 'longitud' no existe en el tipo 'Tipo'.

```
    devolver argumento;  
}
```

Cuando lo hagamos, el compilador nos dará un error de que estamos usando el miembro `.length` de `arg`, pero en ninguna parte hemos dicho que `arg` tiene este miembro. Recuerde, dijimos anteriormente que estas variables de tipo reemplazan a todos los tipos, por lo que alguien que usa esta función podría haber pasado un número en su lugar, que no tiene un miembro `.length`.

Digamos que en realidad pretendemos que esta función funcione en arreglos de Tipo en lugar de Tipo directamente.

Como estamos trabajando con arreglos, el miembro `.length` debería estar disponible. Podemos describir esto tal como crearíamos matrices de otros tipos:

```
función loggingIdentity<Tipo>(arg: Tipo[]): Tipo[] { console.log(arg.longitud);
    devolver argumento;
}
```

Puede leer el tipo de `loggingIdentity` como "la función genérica `loggingIdentity` toma un y un argumento `arg` que es una matriz de números, obtendríamos una matriz, y devolvernos una matriz de tipo `Tipo`". Esto significa que la matriz sera variable de tipo genérico `Type` como parte de los tipos

estamos trabajando con, en lugar de todo el tipo, lo que nos da una mayor flexibilidad.

Alternativamente, podemos escribir el ejemplo de muestra de esta manera:

```
función loggingIdentity<Tipo>(arg: Matriz<Tipo>): Matriz<Tipo> {
    consola.log(arg.longitud); // Array tiene una longitud .así que no más retorno de error arg;
}
```

Es posible que ya esté familiarizado con este estilo de letra de otros idiomas. En la siguiente sección, cubriremos cómo puede crear sus propios tipos genéricos como `Array<Type>`.

Tipos genéricos

En secciones anteriores, creamos funciones de identidad genéricas que funcionaban en una variedad de tipos. En esta sección, exploraremos el tipo de funciones en sí mismas y cómo crear interfaces genéricas.

El tipo de funciones genéricas es como el de las funciones no genéricas, con los parámetros de tipo enumerados primero, de manera similar a las declaraciones de funciones:

```
función identidad<Tipo>(arg: Tipo): Tipo { return arg;
}

let myIdentity: <Tipo>(arg: Tipo) => Tipo = identidad;
```

También podríamos haber usado un nombre diferente para el parámetro de tipo genérico en el tipo, siempre que se alineen el número de variables de tipo y cómo se usan las variables de tipo.

```
función identidad<Tipo>(arg: Tipo): Tipo { return arg;  
}  
  
let myIdentity: <Entrada>(arg: Entrada) => Entrada = identidad;
```

También podemos escribir el tipo genérico como una firma de llamada de un tipo de objeto literal:

```
función identidad<Tipo>(arg: Tipo): Tipo { return arg;  
}  
  
let myIdentity: { <Tipo>(arg: Tipo): Tipo } = identidad;
```

Lo que nos lleva a escribir nuestra primera interfaz genérica. Tomemos el objeto literal del ejemplo anterior y movámoslo a una interfaz:

```
interfaz GenericIdentityFn {  
    <Tipo>(arg: Tipo): Tipo;  
}  
  
función identidad<Tipo>(arg: Tipo): Tipo { return arg;  
}  
  
let myIdentity: GenericIdentityFn = identidad;
```

En un ejemplo similar, podemos querer mover el parámetro genérico para que sea un parámetro de toda la interfaz. Esto nos permite ver qué tipo(s) somos genéricos (por ejemplo, `Dictionary<string>` en lugar de solo `Dictionary`). Esto hace que el parámetro de tipo sea visible para todos los demás miembros de la interfaz.

```

interfaz GenericIdentityFn<Tipo> {
    (arg: Tipo): Tipo;
}

función identidad<Tipo>(arg: Tipo): Tipo { return arg;

}

let myIdentity: GenericIdentityFn<número> = identidad;

```

Observe que nuestro ejemplo ha cambiado para ser algo ligeramente diferente. En lugar de describir una función genérica, ahora tenemos una firma de función no genérica que forma parte de un tipo genérico.

Cuando usamos `GenericIdentityFn`, ahora también necesitaremos especificar el argumento de tipo correspondiente (aquí: `número`), bloqueando efectivamente lo que usará la firma de llamada subyacente.

Comprender cuándo colocar el parámetro de tipo directamente en la firma de llamada y cuándo colocarlo en la interfaz es útil para describir qué aspectos de un tipo son genéricos.

Además de interfaces genéricas, también podemos crear clases genéricas. Tenga en cuenta que no es posible crear enumeraciones y espacios de nombres genéricos.

Clases Genéricas

Una clase genérica tiene una forma similar a una interfaz genérica. Las clases genéricas tienen una lista de parámetros de tipo genérico entre corchetes angulares (`<>`) después del nombre de la clase.

```

class GenericNumber<NumType>
{
    zeroValue: NumType; agregar: (x:
        NumType, y: NumType) => NumType;
}

let myGenericNumber = new GenericNumber<number>();
miNúmeroGenérico.zeroValue = 0; miNúmeroGenérico.add =
función (x, y) {
    devuelve x + y;
}

```

Este es un uso bastante literal de la clase `GenericNumber`, pero es posible que haya notado que nada lo restringe para usar solo el tipo de número. En su lugar, podríamos haber usado una cuerda o incluso más

objetos complejos

```
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = ""; stringNumeric.add = función (x, y) {
    devuelve x + y; }

console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Al igual que con la interfaz, poner el parámetro de tipo en la clase en sí nos permite asegurarnos de que todas las propiedades de la clase funcionen con el mismo tipo.

Como cubrimos en [nuestra sección de clases](#), una clase tiene dos lados en su tipo: el lado estático y el lado de instancia. Las clases genéricas solo son genéricas en su lado de instancia en lugar de su lado estático, por lo que cuando se trabaja con clases, los miembros estáticos no pueden usar el parámetro de tipo de clase.

Restricciones genéricas

Si recuerda un ejemplo anterior, es posible que a veces desee escribir una función genérica que funcione en un conjunto de tipos en los que tenga conocimiento sobre la propiedad de longitud de la entidad que creamos para este tipo. En nuestro ejemplo .length de arg , pero el compilador no pudo probar que todos los tipos tuvieran una propiedad .length , por lo que nos advierte que no podemos hacer esta suposición.

```
función loggingIdentity<Tipo>(arg: Tipo): Tipo {
    consola.log(arg.longitud);
```

La propiedad 'longitud' no existe en el tipo 'Tipo'.

```
    devolver argumento;
}
```

En lugar de trabajar con todos los tipos, nos gustaría restringir esta función para que funcione con cualquiera y tenga pero es obligatorio una sola propiedad llamada longitud. Para ello, podemos extender la interfaz que creamos para todos los tipos, restricción sobre qué tipo puede ser.

Para hacerlo, crearemos una interfaz que describa nuestra restricción. Aquí, crearemos una interfaz que tenga una sola propiedad .length y luego usaremos esta interfaz y la palabra clave extends para

denotemos nuestra restricción:

```
interfaz Longitudinalmente
{ longitud: número;
}

función loggingIdentity<Tipo se extiende a lo largo>(arg: Tipo): Tipo {
    consola.log(arg.longitud); // Ahora sabemos que tiene una propiedad .length, por lo que no se devuelve
    el argumento;
}
```

Debido a que la función genérica ahora está restringida, ya no funcionará con todos los tipos:

```
registro de identidad(3);
```

El argumento de tipo 'número' no se puede asignar al parámetro de tipo
'Longitudinalmente'.

En su lugar, necesitamos pasar valores cuyo tipo tenga todas las propiedades requeridas:

```
loggingIdentity({ longitud: 10, valor: 3 });
```

Uso de parámetros de tipo en restricciones genéricas

Puede declarar un parámetro de tipo que esté restringido por otro parámetro de tipo. Por ejemplo, aquí nos gustaría obtener una propiedad de un objeto dado su nombre. Nos gustaría asegurarnos de no tomar accidentalmente una propiedad que no existe en el obj , por lo que colocaremos una restricción entre los dos tipos:

```
function getProperty<Tipo, Clave extiende clave de Tipo>(obj: Tipo, clave: Clave) { return obj[clave];  
}  
  
sea x = { a: 1, b: 2, c: 3, d: 4 };  
  
obtenerPropiedad(x, "a");  
obtenerPropiedad(x, "m");  
  
tipo "m" no se puede asignar al parámetro de tipo "a" | "b" | "c" | "d". El argumento de
```

Uso de tipos de clase en genéricos

Al crear fábricas en TypeScript usando genéricos, es necesario referirse a los tipos de clase por sus funciones de constructor. Por ejemplo,

```
function create<Tipo>(c: { nuevo (): Tipo }): Tipo { return nuevo c();}  
}
```

Un ejemplo más avanzado usa la propiedad prototipo para inferir y restringir las relaciones entre la función del constructor y el lado de la instancia de los tipos de clase.

```
class BeeKeeper
{ hasMask: boolean = true;
}

clase ZooKeeper
{ etiqueta de nombre : cadena = "Mikle";
}

clase Animal
{ numPiernas: numero = 4;
}

class Bee extiende Animal
{ guardián: BeeKeeper = new BeeKeeper();
}

clase Lion extiende Animal { guardián:
    ZooKeeper = new ZooKeeper();
}

function createInstance<A extiende Animal>(c: new () => A): A {
    devolver nuevo c();
}

createInstance(Lion).keeper.nametag;
createInstance(Bee).keeper.hasMask;
```

Este patrón se usa para potenciar los [mixins](#). patrón de diseño.

Operador de tipo Keyof

El operador de tipo keyof

El operador `keyof` toma un tipo de objeto y produce una cadena o unión literal numérica de sus claves.

El siguiente tipo `P` es del mismo tipo que "`x`" | "`y`":

```
tipo Punto = { x: número; y: número }; tipo P =  
punto clave de;
```

```
    tipo P = punto clave de
```

Si el tipo tiene una firma de índice de cadena o número , `keyof` devolverá esos tipos en su lugar:

```
tipo Arrayish = { [n: número]: desconocido }; tipo A =  
clave de matriz;
```

```
    tipo A = número
```

```
escriba Mapish = { [k: cadena]: booleano }; tipo M =  
clave de Mapish ;
```

```
    tipo M = cadena | número
```

Tenga en cuenta que en este ejemplo, `M` es una cadena | `number` : esto se debe a que las claves de objeto de JavaScript siempre se coaccionan a una cadena, por lo que `obj[0]` siempre es lo mismo que `obj["0"]` .

Los tipos `keyof` se vuelven especialmente útiles cuando se combinan con tipos mapeados, de los cuales aprenderemos más sobre más tarde

Tipo de operador de tipo

El operador de tipo typeof

JavaScript ya tiene un operador `typeof` que puede usar en un

expresión contexto:

```
// Imprime "cadena"
console.log(tipo de "Hola mundo");
```

TypeScript agrega un operador `typeof` que puede usar en una propiedad o:

escribe contexto para referirse a la `escribe` de una variable

```
let s = "hola";
sea n: tipo de s;
```

sea n: cadena

Esto no es muy útil para tipos básicos, pero combinado con otros operadores de tipos, puede usar `typeof` para expresar convenientemente muchos patrones. Por ejemplo, comencemos mirando el tipo predefinido `TipoDeRetorno<T>`. Toma un `y` y produce `un tipo` de retorno:

```
tipo Predicado = (x: desconocido) => booleano;
tipo K = ReturnType<Predicado>;
```

tipo K = booleano

Si intentamos usar `ReturnType` en un nombre de función, vemos un error instructivo:

```
función f() {
    devolver { x: 10, y: 3 };
}
tipo P = TipoRetorno<f>;
```

'f' se refiere a un valor, pero aquí se usa como un tipo. Querías decir 'tipo de f'?

Recuérdalo valores y tipos no son lo mismo. para referirse a la escribe que el valor f tiene, nosotros
tipo de uso :

```
función f() {
    devolver { x: 10, y: 3 };
}
tipo P = ReturnType< tipo de f >;
```

```
tipo P = {
    x: número;
    y: número;
}
```

Limitaciones

TypeScript limita intencionalmente los tipos de expresiones en las que puede usar `typeof`.

Específicamente, solo es legal usar `typeof` en identificadores (es decir, nombres de variables) o sus propiedades. Este ayuda a evitar la trampa confusa de escribir código que cree que se está ejecutando, pero no es así:

```
// Pensado para usar = ReturnType<typeof msgbox>
let shouldContinue: typeof msgbox("¿ Está seguro de que desea continuar?");
```

'' esperado.

Tipos de acceso indexado

Podemos usar un tipo de acceso indexado para buscar una propiedad específica en otro tipo:

```
tipo Persona = { edad: número; nombre: cadena; vivo: booleano }; type Edad =  
Persona["edad"];
```

```
    tipo Edad = número
```

El tipo de indexación es en sí mismo un tipo, por lo que podemos usar uniones, keyof u otros tipos por completo:

```
tipo I1 = Persona["edad" | "nombre"];
```

```
    tipo I1 = cadena | número
```

```
tipo I2 = Persona[clave de Persona];
```

```
    tipo I2 = cadena | número | booleano
```

```
escriba AliveOrName = "vivo" | "nombre"; tipo I3  
= Persona[VivoONombre];
```

```
    tipo I3 = cadena | booleano
```

Incluso verá un error si intenta indexar una propiedad que no existe:

```
tipo I1 = Persona["alve"];
```

La propiedad 'alve' no existe en el tipo 'Persona'.

Otro ejemplo de indexación con un tipo arbitrario es usar un número para obtener el tipo de los elementos de una matriz.

Podemos combinar esto con `typeof` para capturar convenientemente el tipo de elemento de un literal de matriz:

```
const MyArray =
  [ { nombre: "Alice", edad: 15 },
  { nombre: "Bob", edad: 23 },
  { nombre: "Eve", edad: 38 }, ];
```

`tipo Persona = tipo de MiArray [número];`

```
tipo Persona =
  { nombre:
    cadena; edad: número;
  }
```

`tipo Edad = tipo de MiArray [número]["edad"];`

```
tipo Edad = número
```

// O

`escriba Edad2 = Persona["edad"];`

```
tipo Edad2 = número
```

Solo puede usar tipos al indexar, lo que significa que no puede usar una `const` para hacer una referencia variable:

```
const clave = "edad";
escriba Edad = Persona[clave];
```

El tipo 'clave' no se puede utilizar como tipo de índice.
 'clave' se refiere a un valor, pero aquí se usa como un tipo. ¿Querías decir 'typeof key'?

Sin embargo, puede usar un alias de tipo para un estilo similar de refactorización:

```
escriba clave = "edad";
escriba Edad = Persona[clave];
```

Tipos condicionales

En el corazón de la mayoría de los programas útiles, tenemos que tomar decisiones basadas en la entrada. Los programas de JavaScript no son diferentes, pero dado el hecho de que los valores se pueden introspeccionar fácilmente, esas decisiones también se basan en los tipos de entrada. Tipos condicionales desabilidades la relación entre los tipos de entradas y

```
interfaz Animal { vivo ():  
    vacío;  
  
} interfaz Perro extiende Animal { guau():  
    vacío;  
}
```

```
type Example1 = Perro extiende Animal ? numero : cadena;
```

tipo Ejemplo1 = número

```
type Example2 = RegExp extiende Animal ? numero : cadena;
```

tipo Ejemplo2 = cadena

Los tipos condicionales toman una forma que se parece un poco a las expresiones condicionales (`condition ? trueExpression : falseExpression`) en JavaScript:

```
SomeType extiende OtherType ? Tipo verdadero : tipo falso;
```

Cuando el tipo a la izquierda de las extensiones se puede asignar al de la derecha, obtendrá el tipo en la primera rama (la rama "verdadera"); de lo contrario, obtendrá el tipo en la última rama (la rama "falsa").

De los ejemplos anteriores, los tipos condicionales pueden no parecer útiles de inmediato: ¡podemos decírnos a nosotros mismos si Dog extiende Animal o no y elegir número o cadena! Pero el poder de los tipos condicionales proviene de usarlos con genéricos.

Por ejemplo, tomemos la siguiente función `createLabel` :

```
interfaz IdLabel { id:  
    número /* algunos campos */;  
  
} interfaz NameLabel  
{ nombre: cadena /* otros campos */;  
}  
  
función createLabel(id: número): IdLabel; función  
createLabel(nombre: cadena): NameLabel; función  
createLabel(nameOrId: cadena | número): IdLabel | NombreEtiqueta; función  
createLabel(nameOrId: cadena | número): IdLabel | NameLabel { throw "sin implementar";  
}
```

Estas sobrecargas para `createLabel` describen una única función de JavaScript que hace una elección en función de los tipos de sus entradas. Tenga en cuenta algunas cosas:

1. Si una biblioteca tiene que hacer el mismo tipo de elección una y otra vez a lo largo de su API, esto se vuelve engorroso.
2. Tenemos que crear tres sobrecargas: una para cada caso cuando seamos del tipo `una` para cadena y otra para `número`, y otra para el caso más general (tomando una `cadena | número`). Por cada nuevo tipo que `createLabel` puede manejar, la cantidad de sobrecargas crece exponencialmente.

En cambio, podemos codificar esa lógica en un tipo condicional:

```
tipo NameOrId<T extiende el número | cadena> = T extiende número  
? Etiqueta de identificación  
: EtiquetaNombre;
```

Luego podemos usar ese tipo condicional para simplificar nuestras sobrecargas a una sola función sin sobrecargas.

```
función createLabel<T extiende el número | string>(idOrName: T): NameOrId<T> throw "sin
    implementar";
}

let a = createLabel("mecanografiado");
    let a: NameLabel

sea b = crearEtiqueta(2.8);
    sea b: IdLabel

let c = createLabel(Math.random() ? "hola" : 42);
    let c: NameLabel | Etiqueta de identificación
```

Restricciones de tipo condicional

A menudo, las comprobaciones en un tipo condicional nos proporcionarán información nueva. Al igual que con el estrechamiento con guardias de tipo, puede darnos un tipo más específico, la rama verdadera de un tipo condicional restringirá aún más los genéricos por el tipo contra el que verificamos.

Por ejemplo, tomemos lo siguiente:

```
escriba MensajeDe<T> = T["mensaje"];
El tipo "mensaje" no se puede utilizar para indexar el tipo 'T'.
```

En este ejemplo, se producen errores de TypeScript porque no se sabe que T tenga una propiedad llamada mensaje . Podríamos restringir T , y TypeScript ya no se quejaría:

```
type MessageOf<T extends { mensaje: desconocido }> = T["mensaje"];
```

interfaz Correo

```
electrónico { mensaje: cadena;
}
```

escriba EmailMessageContents = MessageOf<Email>;

escriba EmailMessageContents = cadena

Sin embargo, ¿qué pasaría si quisiéramos que MessageOf tomara cualquier tipo y de forma predeterminada algo como nunca si una propiedad de mensaje no está disponible? Podemos hacer esto moviendo la restricción e introduciendo un tipo condicional:

```
type MessageOf<T> = T extends { mensaje: desconocido } ? T["mensaje"] : nunca;
```

interfaz Correo

```
electrónico { mensaje: cadena;
}
```

interfaz Perro

```
{ ladrar(): vacío;
}
```

escriba EmailMessageContents = MessageOf<Email>;

escriba EmailMessageContents = cadena

escriba DogMessageContents = MessageOf<Dog>;

escriba DogMessageContents = nunca

Dentro de la rama verdadera, TypeScript sabe que T voluntad tener una propiedad de mensaje .

Como otro ejemplo, también podríamos escribir un tipo llamado Flatten que aplana los tipos de matriz a sus tipos de elementos, pero los deja solos de lo contrario:

```
type Flatten<T> = T extiende any[] ? T[número] : T;
```

```
// Extrae el tipo de elemento. escriba Str =
Flatten<cadena[]>;
```

tipo Str = cadena

```
// Deja el tipo solo. type Num =
Flatten<número>;
```

tipo Núm = número

Cuando `Flatten` recibe un tipo de matriz, utiliza un acceso indexado con un número para obtener el tipo de elemento de `string[]`. De lo contrario, simplemente devuelve el tipo que se le dio.

Inferir dentro de tipos condicionales

Acabamos de encontrarnos usando tipos condicionales para aplicar restricciones y luego extraer tipos. Esto termina siendo una operación tan común que los tipos condicionales lo hacen más fácil.

Los tipos condicionales nos brindan una forma de inferir de los tipos con los que comparamos en la rama verdadera usando la palabra clave `infer`. Por ejemplo, podríamos haber inferido el tipo de elemento en `Flatten` en lugar de obtenerlo "manualmente" con un tipo de acceso indexado:

```
type Flatten<Type> = Type extends Array<infer Item> ? Artículo : Tipo;
```

Aquí, usamos la palabra clave `infer` para introducir declarativamente una nueva variable de tipo genérico llamada `Item` en lugar de especificar cómo recuperar el tipo de elemento de `T` dentro de la rama verdadera. Esto nos libera de tener que pensar en cómo profundizar y probar la estructura de los tipos que nos interesan.

Podemos escribir algunos alias útiles de tipo ayudante usando la palabra clave `infer`. Por ejemplo, para casos simples, podemos extraer el tipo de retorno de los tipos de funciones:

```
type GetReturnType<Type> = Type extends (...args: never[]) => infer Return
  ? Devolver
  : nunca;
```

tipo Num = GetReturnType<() => número>;

tipo Núm = número

tipo Str = GetReturnType<(x: cadena) => cadena>;

tipo Str = cadena

tipo Bools = GetReturnType<(a: booleano, b: booleano) => booleano[]>;

tipo Bools = booleano[]

Cuando se infiere de un tipo con varias firmas de llamada (como el tipo de una función sobrecargada), las inferencias se hacen a partir de la firma (que, presumiblemente, es el caso de menor rango más permisivo). No es posible realizar una resolución

declarar función cadenaOrNum(x: cadena): número; declarar función
 cadenaOrNum(x: número): cadena; declarar función cadenaOrNum(x:
 cadena | número): cadena | número;

tipo T1 = TipoRetorno< tipodecadenaONum>;

tipo T1 = cadena | número

Tipos condicionales distributivos

Cuando los tipos condicionales actúan sobre un tipo genérico, se convierten en distributivo cuando se le da un tipo de unión.

Por ejemplo, tome lo siguiente:

```
type ToArray<Type> = El tipo extiende cualquier ? Tipo[] : nunca;
```

Si conectamos un tipo de unión en `ToArr`, el tipo condicional se aplicará a cada miembro de esa unión.

```
type ToArr<Type> = El tipo extiende cualquier ? Tipo[] : nunca;
```

```
tipo StrArrOrNumArr = ToArr<cadena | número>;
```

```
escriba StrArrOrNumArr = cadena[] | número[]
```

Lo que sucede aquí es que `StrArrOrNumArr` se distribuye en:

```
cadena | número;
```

y mapas sobre cada tipo de miembro del sindicato, a lo que es efectivamente:

```
ParaArr<cadena> | AArray<número>;
```

lo que nos deja con:

```
cadena[] | número[];
```

Típicamente, la distributividad es el comportamiento deseado. Para evitar ese comportamiento, puede rodear cada lado de la palabra clave `extends` con corchetes.

```
escriba ToArrNonDist<Type> = [Type] extends [any] ? Tipo[] : nunca;
```

```
// 'StrArrOrNumArr' ya no es una unión. escriba  
StrArrOrNumArr = ToArrNonDist<cadena | número>;
```

```
escriba StrArrOrNumArr = (cadena | número)[]
```

Machine Translated by Google

Tipos asignados

Cuando no quiere repetirse, a veces un tipo debe basarse en otro tipo.

Los tipos asignados se basan en la sintaxis de las firmas de índice, que se utilizan para declarar los tipos de propiedades que no se han declarado antes:

```
type SoloBoolsAndHorses = {  
    [clave: cadena]: booleano | Caballo; };  
  
const cumple: OnlyBoolsAndHorses = { del: true,  
    rodney: false, };
```

Un tipo mapeado es un tipo genérico que usa una unión de `PropertyKey`s (frecuentemente creadas [a través de `keyof`](#)) para iterar a través de claves para crear un tipo:

```
escriba Banderas de opciones<Tipo> = {  
    [Propiedad en clave de Tipo]: booleano; };
```

En este ejemplo, `OptionsFlags` tomará todas las propiedades del tipo `Tipo` y cambiará sus valores para que sean booleanos.

```
tipo FeatureFlags = {
    modooscuro: () => vacío;
    nuevoPerfilUsuario: () => void; };
```

escriba FeatureOptions = OptionsFlags<FeatureFlags>;

```
type FeatureOptions =
    { darkMode: boolean;
      nuevoPerfilUsuario: boolean;
    }
```

Modificadores de asignación

Hay dos modificadores adicionales que se pueden aplicar durante el mapeo: `readonly` y `? que afectan la mutabilidad y la optionalidad respectivamente.`

Puede eliminar o agregar estos modificadores anteponiendo el prefijo `-`. Si no agrega un prefijo, entonces `+` o `+`.

```
// Elimina los atributos de 'solo lectura' de las propiedades de un tipo type
CreateMutable<Type> = { -readonly [Property in keyof Type]: Type[Property]; };
```

```
escriba LockedAccount = { ID
    de solo lectura : cadena;
    nombre de solo lectura :
    cadena; };
```

escriba CuentaDesbloqueada = CreateMutable<CuentaBloqueada>;

```
escriba CuentaDesbloqueada =
    { id: cadena; nombre:
      cadena;
    }
```

```
// Elimina los atributos 'opcionales' de las propiedades de un tipo type
```

```
Concrete<Type> = { [Property in keyof Type]-?: Type[Property]; };
```

```
escriba MaybeUser =
```

```
{ id: cadena; nombre?:  
cadena; edad?:  
número; };
```

```
escriba Usuario = Concreto<QuizásUsuario>;
```

```
escriba Usuario  
= { id: cadena;  
nombre: cadena;  
edad: número;  
}
```

Reasignación de teclas a través de as

En TypeScript 4.1 y posteriores, puede volver a asignar claves en tipos asignados con una cláusula as en un tipo asignado:

```
tipo MappedTypeWithNewProperties<Tipo> = {  
[Propiedades en keyof Type como NewKeyType]: Type[Properties]  
}
```

Puede aprovechar características como [tipos de literales de plantilla](#) para crear nuevos nombres de propiedad a partir de los anteriores:

```
tipo Getters<Tipo> = {
    [Propiedad en clave de Tipo como `get${Capitalize<string & Property>}]: ()]
};
```

```
interfaz Persona { nombre:
    cadena; edad:
    número; ubicación:
    cadena;
}
```

escriba LazyPerson = Getters<Persona>;

```
tipo LazyPerson =
    { getName: () => cadena;
        getEdad: () => número;
        getLocation: () => cadena;
    }
```

Puede filtrar claves produciendo nunca a través de un tipo condicional:

```
// Eliminar el tipo de propiedad 'tipo'
RemoveKindField <Type> = {
    [Propiedad en clave de Tipo como Excluir<Propiedad, "tipo">]: Tipo[Propiedad]
};

interfaz Círculo { tipo:
    "círculo"; radio:
    número;
}
```

escriba KindlessCircle = RemoveKindField<Circle>;

```
tipo KindlessCircle = { radio:
    número;
}
```

Puede mapear uniones arbitrarias, no solo uniones de cadenas | número | símbolo , pero uniones de cualquier tipo:

```
type EventConfig<Eventos extiende { tipo: cadena }> = {
    [E en Eventos como E["tipo"]]: (evento: E) => vacío;
}

type SquareEvent = { tipo: "cuadrado", x: número, y: número }; escriba CircleEvent
= { tipo: "círculo", radio: número };

type Config = EventConfig<SquareEvent | Evento del círculo>
```

```
type Config =
    { cuadrado: (evento: SquareEvent) => void; circulo:
        (evento: CircleEvent) => void;
    }
```

Exploración adicional

Los tipos asignados funcionan bien con otras funciones en esta sección de manipulación de tipos, por ejemplo, [aquí hay un tipo asignado que usa un tipo condicional](#) que devuelve verdadero o falso dependiendo de si un objeto tiene la propiedad pii establecida en el literal verdadero :

```
escriba ExtraerPII<Tipo> = {
    [Propiedad en clave de Tipo]: Tipo [Propiedad] extends { pii: true } ? verdadero : };
```

```
escriba DBFields =
    { id: { formato: "incrementando" }; nombre:
        { tipo: cadena; pii: verdadero } );
```

```
escriba ObjectsNeedingGDPRDeletion = ExtractPII<DBFields>;
```

```
escriba ObjectsNeedingGDPRDeletion = { id:
    false; nombre: verdadero;
}
```

Tipos de literales de plantilla

Los tipos de literales de plantilla se basan en [tipos de literales de cadena](#), y tienen la capacidad de expandirse en muchas cadenas a través de uniones.

Tienen la misma sintaxis que [las cadenas literales de plantilla en JavaScript](#), pero se utilizan en posiciones de tipo.

Cuando se usa con tipos de literales concretos, un literal de plantilla produce un nuevo tipo de literal de cadena mediante la concatenación de los contenidos.

```
tipo Mundo = "mundo";
```

```
escriba Saludo = `hola ${Mundo}`;
```

```
escriba Saludo = "hola mundo"
```

Cuando se utiliza una unión en la posición interpolada, el tipo es el conjunto de todos los literales de cadena posibles que podría representar cada miembro de la unión:

```
escriba EmailLocaleIDs = "welcome_email" | "email_heading"; escriba  
FooterLocaleIDs = "footer_title" | "footer_sendoff";
```

```
escriba AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
```

```
escriba AllLocaleIDs = "welcome_email_id" | "email_heading_id" | "footer_tit
```

Para cada posición interpolada en el literal de la plantilla, las uniones se multiplican en cruz:

```

escriba AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;
escriba Lang = "y" |
"sí" | "pt";

escriba LocaleMessageIDs = `${Lang}_${AllLocaleIDs}`;

```

```
escriba LocaleMessageIDs = "en_welcome_email_id" | "en_email_heading_id" | "
```

En general, recomendamos que las personas utilicen la generación anticipada para uniones de cuerdas grandes, pero esto es útil en casos más pequeños.

Uniones de cuerdas en tipos

El poder de los literales de plantilla surge cuando se define una nueva cadena basada en la información dentro de un tipo.

Considere el caso en el que una función (makeWatchedObject) agrega una nueva función llamada on() a un objeto pasado. En JavaScript, su llamada podría verse como: makeWatchedObject(baseObject) . Podemos imaginar el objeto base con el siguiente aspecto:

```

const objetoPasado = { nombre:
  "Saoirse", apellido: "Ronan",
  edad: 26, };

```

La función on que se agregará al objeto base espera dos argumentos, un eventName (una cadena) y un callBack (una función).

El nombre del evento debe tener el formato atributoEnElObjetoPasado + "Cambiado" ; por lo tanto, firstNameChanged como derivado del atributo firstName en el objeto base.

La función callBack , cuando se llama:

- Se debe pasar un valor del tipo asociado con el atributo de nombreInThePassedObject ; por lo tanto, dado que firstName se escribe como string , la devolución de llamada para el evento firstNameChanged espera que se le pase una cadena en el momento de la llamada. De manera similar, los eventos asociados con la edad deben esperar ser llamados con un argumento numérico

- Debería tener un tipo de retorno nulo (para simplificar la demostración)

La firma de función ingenua de `on()` podría ser: `on(eventName: string, callBack: (newValue: any) => void)` .

Sin embargo, en la descripción anterior, identificamos restricciones de tipo importantes que nos gustaría documentar en nuestro código. Los tipos de plantilla literal nos permiten traer estas restricciones a nuestro código.

```
const person = makeWatchedObject({ firstName:  
  "Saoirse", lastName: "Ronan", edad: 26, });  
  
// makeWatchedObject ha agregado `on` al objeto anónimo  
  
person.on("firstNameChanged", (newValue) => {  
  console.log(`firstName se cambió a ${newValue}`);});
```

Tenga en cuenta que al escuchar el evento "firstNameChanged" , no solo "firstName" . Nuestra especificación ingenua de `on()` podría hacerse más sólida si nos aseguráramos de que el conjunto de nombres de eventos elegibles estuviera restringido por la unión de nombres de atributos en el objeto observado con "Cambiado" agregado al final. Si bien nos sentimos cómodos haciendo dicho cálculo en JavaScript, es decir `Object.keys(passedObject).map(x => `${x}Changed`)` , los literales de plantilla proporcionan dentro del tipo sistema un enfoque similar para la manipulación de cadenas:

```
type PropEventSource<Type> =  
  { on(eventName: `${string & keyof Type}Changed`, callback: (newValue: an  
});  
  
/// Cree un "objeto observado" con un método 'on' /// para que pueda  
observar los cambios en las propiedades. declarar la función  
makeWatchedObject<Type>(obj: Type): Type & PropEventSourc
```

Con esto, podemos construir algo que falla cuando se le da la propiedad incorrecta:

```
const person = makeWatchedObject({ firstName:  
  "Saoirse", lastName: "Ronan", age: 26 });  
  
person.on("firstNameChanged", () => {});  
  
// Evita errores humanos fáciles (utilizando la clave en lugar del nombre del evento)  
person.on("firstName", () => {});
```

tipo "firstName" no se puede asignar al parámetro de tipo ~~afectandoCambio~~. El argumento de "edadCambiada" o "edadCambiada". "nombreCambiado" | "apellidoCambiado" |

// Es resistente a errores
tipográficos person.on("frstNameChanged", () => {});

de tipo "frstNameChanged" no se puede asignar al parámetro de ~~tipоЎfrstName~~. El argumento "edadCambiada" o "edadCambiada". escriba "nombreCambiado" | "apellidoCambiado" |

Inferencia con literales de plantilla

Tenga en cuenta que no nos beneficiamos de toda la información proporcionada en el objeto pasado original. Dado el cambio de un nombre (es decir, un evento `firstNameChanged`), deberíamos esperar que la devolución de llamada reciba un argumento de tipo `cadena`. De manera similar, la devolución de llamada para un cambio de edad debe recibir un argumento de número. Ingenuamente estamos usando `any` para escribir el argumento de `callback`. Una vez más, los tipos de literales de plantilla permiten garantizar que el tipo de datos de un atributo sea el mismo tipo que el primer argumento de devolución de llamada de ese atributo.

La idea clave que hace que esto sea posible es la siguiente: podemos usar una función con un genérico tal que:

1. El literal utilizado en el primer argumento se captura como un tipo de literal 2. Ese tipo de literal se puede validar como parte de la unión de atributos válidos en el genérico 3. El tipo del atributo validado se puede buscar en la estructura del genérico usando indexado Acceso
4. Esta información de escritura puede ser `después` aplicarse para garantizar que el argumento de la función de devolución de llamada sea del mismo tipo

```
type PropEventSource<Type> = { on<Key
  extends string & keyof Type>
  (eventName: `${Key}Changed`, callback: (newValue: Type[Key]) => void);
};
```

declarar la función makeWatchedObject<Type>(obj: Type): Type & PropEventSource

```
const person = makeWatchedObject({ firstName:
  "Saoirse", lastName: "Ronan", age: 26 });
```

```
person.on("firstNameChanged", newName => {
```

(parámetro) nuevoNombre: cadena

```
  console.log(`el nuevo nombre es ${newName.toUpperCase()}`);
});
```

```
person.on("edadCambiada", nuevaEdad => {
```

(parámetro) newAge: número

```
  if (newAge < 0)
    { console.warn("advertencia! edad negativa");
  }
})
```

Aquí nos convertimos en un método genérico.

Cuando un usuario llama con la cadena "firstNameChanged" , TypeScript intentará inferir el tipo correcto para Key . Para hacerlo, comparará la clave con el contenido anterior a "Cambiado" e inferirá la cadena. Una vez que TypeScript "primer nombre" se da cuenta de eso, el método on puede obtener el tipo de firstName en el objeto original, que es una cadena en este caso. De manera similar, cuando se llama con "ageChanged" , TypeScript encuentra el tipo para la propiedad age que es number .

La inferencia se puede combinar de diferentes maneras, a menudo para deconstruir cadenas y reconstruirlas de diferentes maneras.

Tipos de manipulación de cadenas intrínsecas

Para ayudar con la manipulación de cadenas, TypeScript incluye un conjunto de tipos que se pueden usar en la manipulación de cadenas. Estos tipos vienen integrados en el compilador para mejorar el rendimiento y no se pueden encontrar en los archivos .d.ts incluidos con TypeScript.

Mayúsculas<StringType>

Convierte cada carácter de la cadena a la versión en mayúsculas.

Ejemplo

```
escriba Saludo = "Hola mundo" escriba
```

```
GritoSaludo = Mayúsculas<Saludo>
```

```
escriba ShoutyGreeting = "HOLA, MUNDO"
```

```
escriba ASCIICacheKey<Str extends string> = `ID-${Uppercase<Str>}` escriba MainID =
```

```
ASCIICacheKey<"my_app">
```

```
escriba MainID = "ID-MY_APP"
```

Minúsculas<StringType>

Convierte cada carácter de la cadena al equivalente en minúsculas.

Ejemplo

```
escriba Saludo = "Hola, mundo" escriba
```

```
QuietGreeting = Minúsculas<Saludo>
```

```
escriba QuietGreeting = "hola, mundo"
```

```
escriba ASCIICacheKey<Str extends string> = `id-${Lowercase<Str>}` escriba MainID =
```

```
ASCIICacheKey<"MY_APP">
```

```
escriba MainID = "id-mi_aplicación"
```

Mayúsculas<StringType>

Convierte el primer carácter de la cadena en un equivalente en mayúsculas.

Ejemplo

```
type LowercaseGreeting = "hola, mundo"; escriba  
Saludo = Capitalize<LowercaseGreeting>;
```

```
escriba Saludo = "Hola, mundo"
```

Quitar mayúsculas<StringType>

Convierte el primer carácter de la cadena en un equivalente en minúsculas.

Ejemplo

```
escribe Saludo en Mayúsculas = "HOLA MUNDO";  
escriba UncomfortableGreeting = Uncapitalize<UppercaseGreeting>;
```

```
escriba UncomfortableGreeting = "HOLA MUNDO"
```

- Detalles técnicos sobre los tipos de manipulación de cadenas intrínsecas

Clases

TypeScript ofrece soporte completo para la palabra clave de clase introducida en ES2015.

Lectura de fondo:
[Clases \(MDN\)](#).

Al igual que con otras características del lenguaje JavaScript, TypeScript agrega anotaciones de tipo y otra sintaxis para permitirle expresar relaciones entre clases y otros tipos.

Miembros de clase

Aquí está la clase más básica, una vacía:

```
punto de clase {}
```

Esta clase aún no es muy útil, así que comenzemos a agregar algunos miembros.

Campos

Una declaración de campo crea una propiedad pública escribible en una clase:

```
clase Punto { x:  
    número; y:  
    número;  
}  
  
const pt = nuevo Punto(); pt.x  
= 0; pt.y = 0;
```

Al igual que con otras ubicaciones, la anotación de tipo es opcional, pero será implícita cualquiera si no se especifica.

Los campos también pueden tener `inicializadores`, estos se ejecutarán automáticamente cuando se instancia la clase:

```
clase Punto { x =
  0; y = 0;

}

const pt = nuevo Punto(); //
Imprime 0, 0 console.log(`$ ${pt.x}, ${pt.y}`);
```

Al igual que con constante , dejar , y fue , el inicializador de una propiedad de clase se utilizará para inferir su tipo:

```
const pt = nuevo Punto(); pt.x =
"0";
```

El tipo 'cadena' no se puede asignar al tipo 'número'.

--strictPropertyInitialization

[Inicialización estricta de la propiedad](#) la configuración controla si los campos de clase deben inicializarse en el constructor.

```
clase BadGreeter
{ nombre: cadena;

propiedad 'nombre' no tiene inicializador y no se asigna definitivamente en el constructor.
```

}

```
clase GoodGreeter
{ nombre: cadena;

constructor()
{ este.nombre = "hola";
}
```

}

Tenga en cuenta que el campo debe inicializarse en el mismo constructor . TypeScript no analiza métodos que invoque desde el constructor para detectar inicializaciones, porque una clase derivada podría anular esos métodos y fallar al inicializar los miembros.

Si tiene la intención de inicializar definitivamente un campo a través de otros medios que no sean el constructor (por ejemplo, tal vez una biblioteca externa esté completando parte de su clase por usted), puede usar el asignación definitiva operador de afirmación ! :

```
clase Bienvenida {
    // No inicializado, pero sin error
    nombre!: cadena;
}
```

solamente lectura

Los campos pueden tener el prefijo del modificador de solo lectura . Esto evita asignaciones al campo fuera del constructor.

```
saludador de clase {
    nombre de solo lectura : cadena = "mundo";

    constructor(otroNombre?: cadena) {
        if (otroNombre !== indefinido) {
            este.nombre = otroNombre;
        }
    }

    errar() {
        this.name = "no está bien";
    }
}
```

No se puede asignar a 'nombre' porque es una propiedad de solo lectura.

```
}
```

```
const g = nuevo Saludador();
g.name = "tampoco está bien";
```

No se puede asignar a 'nombre' porque es una propiedad de solo lectura.

Constructores

Los constructores de clases son muy similares a las funciones. Puede agregar parámetros con anotaciones de tipo, valores predeterminados y sobrecargas:

Lectura de fondo:
[Constructor \(MDN\)](#).

```
clase Punto { x:
    número; y:
    número;

    // Firma normal con constructor predeterminado
    (x = 0, y = 0) { this.x = x; esto.y = y;

    }
}
```

```
punto de clase {
    // Sobre carga
    constructor(x: número, y: cadena);
    constructor(es: cadena); constructor(xs:
    cualquiera, y?: cualquiera) { // TBD

    }
}
```

Hay solo algunas diferencias entre las firmas de constructores de clases y las firmas de funciones:

- Los constructores no pueden tener parámetros de tipo; estos pertenecen a la declaración de clase externa, de la que aprenderemos más adelante.
- Los constructores no pueden tener anotaciones de tipo de devolución: el tipo de instancia de clase es siempre lo que se devuelve

Súper llamadas

Al igual que en JavaScript, si tiene una clase base, deberá llamar a `super()`; en el cuerpo de su constructor antes de usar `este.miembros`:

```
clase Base { k
    = 4;
}
```

```
class Derived extends Base
{ constructor() { // Imprime un valor
    incorrecto en ES5; lanza una excepción en ES6 console.log(this.k);
```

debe llamar a 'super' para tener acceso a los métodos de la clase base en el constructor de una clase derivada.

```
súper();
}
```

Olvidar llamar a super es un error fácil de cometer en JavaScript, pero TypeScript te dirá cuándo es necesario.

Métodos

Una propiedad de función en una clase se denomina `método`. Los métodos pueden usar todos las anotaciones del mismo tipo que funciones y constructores:

Lectura de fondo:

[Definiciones de métodos](#)

```
clase Punto { x =
    10; y = 10;

    escala(n: número): vacío {
        esto.x *= n;
        esto.y *= n;
    }
}
```

Aparte de las anotaciones de tipo estándar, TypeScript no agrega nada nuevo a los métodos.

Tenga en cuenta que dentro del cuerpo de un método, aún es obligatorio acceder a los campos y otros métodos a través de este. Un nombre no calificado en el cuerpo de un método siempre se referirá a algo en el ámbito adjunto:

```

sea x: número = 0;

clase C { x:
    cadena = "hola";

    m() {
        // Esto está tratando de modificar 'x' desde la línea 1, no la propiedad de clase x = "mundo";
    }
}

```

El tipo 'cadena' no se puede asignar al tipo 'número'.

Compradores / Setters

Las clases también pueden tener accesorios :

```

clase C
{
    _longitud = 0;
    obtener longitud()
    {
        devuelve esto._longitud;
    }

    establecer longitud (valor) {
        esto._longitud = valor;
    }
}

```

Tenga en cuenta que un par get/set respaldado por campos sin lógica adicional rara vez es útil en JavaScript. Está bien exponer campos públicos si no necesita agregar lógica adicional durante las operaciones de obtener/establecer.

TypeScript tiene algunas reglas de inferencia especiales para los accesores:

- Si `get` existe pero no se establece , la propiedad es automáticamente de solo lectura
- Si no se especifica el tipo del parámetro `setter`, se deduce del tipo de retorno del `getter`
- Getters y setters deben tener la misma visibilidad de miembro .

Desde [TypeScript 4.3](#) es posible tener accesorios con diferentes tipos para obtener y configurar.

```

clase Cosa
{ _tamaño = 0;

obtener tamaño (): número
{ devuelve this._size;
}

establecer tamaño (valor: cadena | número | booleano) {
let num = Número (valor);

// No permitir NaN, Infinity, etc.

if (!Number.isFinite(num)) { this._size
= 0; devolver;

}
estos._tamaño = num;
}
}

```

Índice de firmas

Las clases pueden declarar firmas de índice; estos funcionan igual que las [firmas de índice para otros tipos de objetos](#) :

```

clase MiClase { [s:
cadena]: booleano | ((s: cadena) => booleano);

check(s: string)
{ devuelve estos[s] como booleano;
}
}

```

Debido a que el tipo de firma de índice también debe capturar los tipos de métodos, no es fácil usar estos tipos de manera útil. Por lo general, es mejor almacenar los datos indexados en otro lugar en lugar de en la propia instancia de la clase.

Herencia de clase

Al igual que otros lenguajes con funciones orientadas a objetos, las clases en JavaScript pueden heredar de las clases base.

implementa cláusulas

Puede usar una cláusula de implementaciones para verificar que una clase satisfaga una interfaz particular . Un error que se emitirá si una clase no la implementa correctamente:

```
interfaz Pingable { ping ():  
    vacío;  
}  
  
class Sonar implementa Pingable { ping()  
    { console.log("ping!");  
  
    }  
}  
  
clase Ball implementa Pingable {  
  
    La clase 'Ball' implementa incorrectamente la interfaz 'Pingable'.  
    propiedades adicionales requeridas por la interfaz requieren el tipo Falta la  
    'Pingable'.  
  
    pong()  
    { consola.log("¡pong!");  
    }  
}
```

Las clases también pueden implementar múltiples interfaces, por ejemplo, la clase C implementa A, B { .

Precauciones

Es importante comprender que una cláusula de implementación es solo una verificación de que la clase se puede tratar como el tipo de interfaz. No cambia el tipo de clase o sus métodos. La fuente de error es asumir que una comén cláusula de implementos cambiará el tipo de clase, ¡no es así!

interfaz Comprobable

```
{ comprobar (nombre: cadena): booleano;
}
```

```
class NameChecker implementa Checkable { check(s)
{
```

El parámetro 's' implícitamente tiene un tipo 'cualquiera'.

// No observe ningún error aquí

```
return s.toLowerCase() === "ok";
```



ningún

```
}
```

En este ejemplo, tal vez esperábamos que el tipo de s estuviera influenciado por el nombre: parámetro de cadena de verificación . No lo es: ~~clase A usa la clase implementación~~ No cambian la forma en que se verifica el cuerpo de la

De manera similar, implementar una interfaz con una propiedad opcional no crea esa propiedad:

interfaz A { x:

número; y?:

número;

```
} la clase C implementa A {
```

x = 0;

```
} const c = nueva C();
```

cy = 10;

La propiedad 'y' no existe en el tipo 'C'.

amplía las cláusulas

Las clases pueden extenderse desde una clase base. Una clase derivada tiene todas las propiedades y métodos de su clase base y también define miembros adicionales.

Lectura de antecedentes:

[extiende la palabra clave \(MDN\)](#)

```
class Animal
{ move()
  { console.log("¡Avanzando !");
}
}

clase Perro extiende Animal
{ guau(veces: número) { for (let i
= 0; i < veces; i++) { console.log("¡guau!");
}

}
}

const d = nuevo Perro(); //
Método de clase base
d.move(); // Método de
clase derivado
d.guau(3);
```

Métodos de anulación

Una clase derivada también puede anular una propiedad o un campo de clase base. Puedes usar el `súper.` sintaxis para acceder a los métodos de la clase base. Tenga en cuenta que debido a que las clases de JavaScript son un objeto de búsqueda simple, no existe la noción de un "supercampo".

Lectura de antecedentes:
[superpalabra clave \(MDN\)](#).

TypeScript exige que una clase derivada sea siempre un subtipo de su clase base.

Por ejemplo, aquí hay una forma legal de anular un método:

```
class Base
  { saludar()
    { console.log("¡Hola, mundo!");
  }
}

clase Derivado extiende Base
{ saludar ({nombre?: cadena} {
  if (nombre === indefinido)
    { super.saludo(); } else
  { console.log(`Hola, ${nombre.toUpperCase()}`);
  }
}
}

const d = nuevo Derivado();
d.saludar(); d.saludo("lector");
```

Es importante que una clase derivada siga su contrato de clase base. Recuerda que es muy común (¡y siempre legal!) referirse a una instancia de clase derivada a través de una referencia de clase base:

```
// Alias de la instancia derivada a través de una referencia de clase base const b:
Base = d; // No hay problema b.saludo();
```

¿Qué pasa si Derived no siguió el contrato de Base ?

```
class Base
{
    salutar()
    { console.log("¡Hola, mundo!");
    }
}

clase Derivado extiende Base {
    // Hacer que este parámetro sea obligatorio
    saludar (nombre: cadena) {
```

'saludo' en el tipo 'Derivado' no se puede asignar a la misma propiedad en ~~Propiedad de base 'Base'. propiedad~~
en el tipo base 'Base'.

El tipo '(nombre: cadena) => vacío' no se puede asignar al tipo '() => vacío'.

```
    console.log(`Hola, ${nombre.toUpperCase()}`);
}
}
```

Si compilamos este código a pesar del error, esta muestra fallaría:

```
const b: Base = nuevo Derivado(); // Falla
porque "nombre" no estará definido b.greet();
```

Declaraciones de campo de solo tipo

Cuando el objetivo `>= ES2022` o `useDefineForClassFields` es verdadero después de `,`, los campos de clase se inicializan que se completa el constructor de la clase principal, sobrescribiendo cualquier valor establecido por la clase principal. Esto puede ser un problema cuando solo desea volver a declarar un tipo más preciso para un campo heredado. Para manejar estos casos, puede escribir `declare` para indicar a TypeScript que no debería haber ningún efecto de tiempo de ejecución para esta declaración de campo.

```

interfaz Animal
  { fechaDeNacimiento: cualquiera;
}

interfaz Perro extiende Animal { raza:
  cualquiera;
}

clase AnimalHouse
  { residente: Animal;
  constructor(animal: Animal)
    { this.residente = animal;
  }
}

clase DogHouse extiende AnimalHouse {
  // No emite código JavaScript, // solo asegura
  que los tipos sean correctos declare resident: Dog;
  constructor(perro: Perro) { super(perro);

  }
}

```

Orden de inicialización

El orden en que se inicializan las clases de JavaScript puede ser sorprendente en algunos casos. Consideremos este código:

```

clase Base
  { nombre = "base";
  constructor()
    { console.log("Mi nombre es " + este.nombre);
  }
}

clase Derivado extiende Base
  { nombre = "derivado";
}

// Imprime "base", no "derivado" const d =
new Derivado();

```

¿Qué pasó aquí?

El orden de inicialización de clases, tal como lo define JavaScript, es:

- Los campos de la clase base se inicializan.
- El constructor de la clase base se ejecuta
- Los campos de la clase derivada se inicializan.
- El constructor de la clase derivada se ejecuta

Esto significa que el constructor de la clase base vio su propio valor para el nombre durante su propio constructor, porque las inicializaciones del campo de la clase derivada aún no se habían ejecutado.

Heredar tipos incorporados

Nota: si no planea heredar de tipos incorporados como Array , Error , Mapa , etc. o su compilación el destino está establecido explícitamente en ES6 / ES2015 o superior, puede omitir esta sección

En ES2015, los constructores que devuelven un objeto sustituyen implícitamente el valor de this por cualquier llamante de super(...). Es necesario que el código constructor generado capture cualquier potencial devuelva el valor de super(...) y reemplázalo con este .

Como resultado, es posible que la subclase Array de Error y otras ya no funcionen como se esperaba. Esto es debido a hecho de que las funciones constructoras para Error y similares Array , ECMAScript 6

new.target para ajustar la cadena de prototipos; sin embargo, no hay manera de asegurar un valor para new.target al invocar un constructor en ECMAScript 5. Otros compiladores de nivel inferior generalmente tienen la misma limitación por defecto.

Para una subclase como la siguiente:

```
class MsgError extiende Error {
    constructor(m: cadena) {
        súper(m);
    }
    decir hola() {
        devolver "hola" + este.mensaje;
    }
}
```

usted puede encontrar que:

- los métodos pueden no estar definidos en los objetos devueltos al construir estas subclases, por lo que llamar a sayHello generará un error.
- instanceof se dividirá entre las instancias de la subclase y sus instancias, por lo que (nuevo MsgError()) instancia de MsgError devolverá falso .

Como recomendación, puede ajustar manualmente el prototipo inmediatamente después de cualquier super(...) llamada.

```
class MsgError extiende Error
{ constructor(m: string) { super(m);

    // Establecer el prototipo explícitamente.
    Object.setPrototypeOf(this, MsgError.prototype);
}

decirHola()
{ devolver "hola" + este.mensaje;
}
}
```

Sin embargo, cualquier subclase de MsgError también tendrá que configurar manualmente el prototipo. Para tiempos de ejecución que no admiten [Object.setPrototypeOf](#), en su lugar, puede usar [proto](#).

Desafortunadamente, [estas soluciones alternativas no funcionarán en Internet Explorer 10 y versiones anteriores](#). Uno puede copiar manualmente los métodos del prototipo en la instancia misma (es decir, `MsgError.prototype` en `this`), pero la cadena de prototipos en sí no se puede arreglar.

Visibilidad de miembros

Puede usar TypeScript para controlar si ciertos métodos o propiedades son visibles para el código fuera de la clase.

público

La visibilidad predeterminada de los miembros de la clase es pública . Se puede acceder a un miembro público desde cualquier lugar:

```
class Greeter
  { saludo público ()
    { console.log ("¡hola!");
  }

} const g = nuevo Saludador();
g.saludo();
```

Debido a que público ya es el modificador de visibilidad predeterminado, nunca será miembro, pero necesitar para escribirlo en una clase puede optar por hacerlo por razones de estilo/legibilidad.

protegido

los miembros protegidos solo son visibles para las subclases de la clase en la que están declarados.

```
saludador de clase {
  saludo público ()
  { console.log("Hola, " + this.getName());

} getName protegido ()
  { devuelve "hola";
}

class SpecialGreeter extiende Greeter { public
  howdy () { // OK para acceder al miembro
  protegido aquí console.log ("Hola, "
  + this.getName());
}

} const g = new SaludoEspecial();
g.saludo(); // Aceptar g.getName();
```

Importante La propiedad 'getName' está protegida y solo se puede acceder a ella dentro de la clase 'Greeter' y sus subclases.

Exposición de miembros protegidos

Las clases derivadas deben seguir sus contratos de clase base, pero pueden optar por exponer un subtipo de clase base con más capacidades. Esto incluye hacer públicos los miembros protegidos :

```
clase Base
{ protegida m = 10;

} clase Derivado extiende Base {
    // Sin modificador, por lo que el valor predeterminado es
    'público' m = 15;

} const d = nuevo Derivado();
consola.log(dm); // OK
```

Tenga en cuenta que Derived ya podía leer y escribir libremente en la , así que esto no altera significativamente "seguridad" de esta situación. Lo principal a tener en cuenta aquí es que en la clase derivada, debemos tener cuidado de repetir el modificador protected si esta exposición no es intencional.

Acceso protegido entre jerarquías

Distintos lenguajes de programación orientada a objetos discrepan sobre si es legal acceder a un miembro protegido a través de una referencia de clase base:

```
clase Base
{ protegido x: número = 1;

} class Derived1 extends Base
{ protected x: number = 5;

} class Derivado2 extiende Base
{ f1(otro: Derivado2) { otro.x = 10;

} f2(otro: Base) { otro.x
    = 10;
```

'x' está protegida y solo es accesible a través de una instancia de la clase 'Derived1'. Si intentas acceder a 'x' a través de una instancia de la clase 'Base', obtendrás un error.

```
}
```

Java, por ejemplo, considera que esto es legal. Por otro lado, C# y C++ optaron por que este código fuera ilegal.

TypeScript se pone del lado de C# y C++ aquí, porque acceder a x en Derived2 solo debería ser legal desde las subclases de Derived2, y Derived1 no es una de ellas. Además, si acceder a x a través de una referencia Derivada1 es ilegal (¡lo cual ciertamente debería ser!), entonces acceder a él a través de una referencia de clase base nunca debería mejorar la situación.

Consulte también [¿Por qué no puedo acceder a un miembro protegido de una clase derivada?](#) lo que explica más del razonamiento de C#.

privado

private es como protected, pero no permite el acceso al miembro ni siquiera desde las subclases:

```
clase Base { privado
    x = 0;
}
const b = nueva Base();
// No se puede acceder desde fuera de la clase
consola.log(bx);
```

La propiedad 'x' es privada y solo se puede acceder a ella dentro de la clase 'Base'.

```
class Derived extends Base { showX() { // No
    se puede acceder a las subclases
    console.log(this.x);
```

La propiedad 'x' es privada y solo se puede acceder a ella dentro de la clase 'Base'.

```
}
```

Debido a que los miembros privados no son visibles para las clases derivadas, una clase derivada no puede aumentar su visibilidad:

clase Base

```
{ privado x = 0;

} clase Derivado extiende Base {
```

La clase 'Derivado' extiende incorrectamente la clase base 'Base'.

La propiedad 'x' es privada en el tipo 'Base' pero no en el tipo 'Derivado'.

```
x = 1;
```

```
}
```

Acceso privado entre instancias

Los diferentes lenguajes de programación orientada a objetos no están de acuerdo sobre si diferentes instancias de la misma clase pueden acceder a los miembros privados de los demás. Mientras que lenguajes como Java, C#, C++, Swift y PHP lo permiten, Ruby no lo permite.

TypeScript permite el acceso privado entre instancias :

clase A

```
{ privado x = 10;

público igual que (otro: A) {
    // No hay error
    devuelve otro.x === este.x;
}
```

Advertencias

Al igual que otros aspectos del sistema de tipos de TypeScript, private y protected solo se aplican durante la verificación de tipos.

Esto significa que las construcciones de tiempo de ejecución de JavaScript como en o una búsqueda de propiedad simple aún pueden acceder a un miembro privado o protegido :

```
clase MySafe {
    clave secreta privada = 12345 ;
```

```
// En un archivo JavaScript... const
s = new MySafe(); // Imprimirá
12345 console.log(s.secretKey);
```

private también permite el acceso usando la notación de corchetes durante la verificación de tipo. Esto hace que los campos declarados privados sean potencialmente más fáciles de acceder para cosas como pruebas unitarias, con el inconveniente de que estos campos ~~se ven~~ no imponen estrictamente la privacidad.

```
clase MySafe {
    clave secreta privada = 12345 ;
}
```

```
const s = nuevo MySafe();
```

```
// No permitido durante la comprobación de tipo
console.log(s.secretKey);
```

La propiedad 'secretKey' es privada y solo se puede acceder a ella dentro de la clase 'MySafe'.

```
//
Acepta consola.log(s["secretKey"]);
```

A diferencia de los campos privados de JavaScript, los campos privados de TypeScript (#) permanecen privados después de la compilación y no proporcionan las escotillas de escape mencionadas anteriormente, como el acceso a la notación de corchetes, lo que los hace duro privado.

```
clase Perro
{ #ladrarCantidad =
  0; personalidad = "feliz";

  constructor() {}
}
```

```
"uso estricto";
clase Perro
{ #ladrarCantidad =
0; personalidad = "feliz";
constructor() { }
}
```

Al compilar a ES2021 o menos, TypeScript usará WeakMaps en lugar de #

```
"uso estricto";
var _Dog_barkAmount;
clase Perro { constructor()

{ _Dog_barkAmount.set(this, 0);
esta.personalidad = "feliz";
}
}

_Dog_barkAmount = new WeakMap();
```

Si necesita proteger valores en su clase de actores maliciosos, debe usar mecanismos que ofrezcan privacidad de tiempo de ejecución estricto, como cierres, WeakMaps o campos privados. Tenga en cuenta que estas comprobaciones de privacidad adicionales durante el tiempo de ejecución podrían afectar el rendimiento.

Miembros estáticos

Las clases pueden tener miembros estáticos . Estos miembros no están asociados con una instancia particular de la clase. Se puede acceder a ellos a través del propio objeto constructor de clases:

Lectura de fondo:
[Miembros estáticos \(MDN\)](#)

```

clase MiClase
{ estática x = 0;
printX estático ()
    { console.log (MyClass.x);
}
}

} consola.log(MiClase.x);
MiClase.printX();

```

Los miembros estáticos también pueden usar los mismos modificadores de visibilidad `public` , `protected` y `private` :

```

class MyClass
{ private static x = 0;

} consola.log(MiClase.x);

```

La propiedad 'x' es privada y solo se puede acceder a ella dentro de la clase 'MyClass'.

Los miembros estáticos también se heredan:

```

class Base
{ static getGreeting() { return
    "Hola mundo";
}

} class Derivado extiende Base
{ myGreeting = Derived.getGreeting();
}

```

Nombres estáticos especiales

Por lo general, no es seguro ni posible sobrescribir las propiedades del prototipo de función . Debido a que las clases son en sí mismas funciones que se pueden invocar con nuevas , ciertos nombres estáticos no pueden ser usó. Las propiedades de función como el nombre `deYada` no son válidas para definir como miembros estáticos ,

clase S

```
{ nombre estático = "¡S!";
```

La propiedad estática 'nombre' entra en conflicto con la propiedad integrada 'Función.nombre' de la función constructora 'S'. de la función constructora 'S'.

```
}
```

¿Por qué no hay clases estáticas?

TypeScript (y JavaScript) no tienen una construcción llamada clase estática de la misma manera que, por ejemplo, C#.

Esa clase de solamente existen porque esos lenguajes fuerzan que todos los datos y funciones estén dentro de un construcciones; debido a que esa restricción no existe en TypeScript, no hay necesidad de ellos. Una clase con una sola instancia normalmente se representa como una clase normal. objeto en JavaScript/TypeScript.

Por ejemplo, no necesitamos una sintaxis de "clase estática" en TypeScript porque un objeto normal (o incluso una función de nivel superior) hará el trabajo igual de bien:

```
// clase "estática" innecesaria class
MyStaticClass {
    hacer algo estático () {}
}

// Función preferida (alternativa 1)
hacerAlgo() {}

// Preferido (alternativa 2) const
MyHelperObject = { haceralgo() {}, };
```

Bloques estáticos en clases

Los bloques estáticos le permiten escribir una secuencia de declaraciones con su propio alcance que pueden acceder a campos privados dentro de la clase contenedora. Esto significa que podemos escribir código de inicialización con todas las capacidades de escribir sentencias, sin fugas de variables y acceso total a las funciones internas de nuestra clase.

```

clase Foo
{ static #count = 0;

    get count ()
        { return Foo.#count;
    }

    static { try
        { const
            lastInstances = loadLastInstances(); Foo.#count +=
            lastInstances.length;

        } atrapar {}
    }

}

```

Clases Genéricas

Las clases, al igual que las interfaces, pueden ser genéricas. Cuando se instancia una clase genérica con nuevos , su tipo parámetros, se infieren de la misma manera que en una llamada de función:

```

class Box<Tipo>
{ contenido: Tipo;
constructor(valor: Tipo)
    { this.contents = valor;
}

```

```
const b = new Box("hola!");
```

const b: Caja<cadena>

Las clases pueden usar restricciones genéricas y valores predeterminados de la misma manera que las interfaces.

Parámetros de tipo en miembros estáticos

Este código no es legal, y puede que no sea obvio por qué:

```
caja de clase <Tipo> {
    valor predeterminado estático : tipo;
}

Los miembros estáticos no pueden hacer referencia a parámetros de tipo de clase.

}
```

¡Recuerde que los tipos siempre se borran por completo! En tiempo de ejecución, solo hay un espacio de propiedad. Esto significa que configurar Box<string>.defaultValue (si fuera posible) además cambie Box<number>.defaultValue - no es bueno. Los miembros estáticos de una clase genérica nunca puede hacer referencia a los parámetros de tipo de la clase.

esto en tiempo de ejecución en clases

Es importante recordar que TypeScript no cambia el tiempo de ejecución comportamiento de JavaScript, y que JavaScript es algo famoso por tener algunos comportamientos de tiempo de ejecución peculiares.

El manejo de JavaScript de esto es realmente inusual:

Lectura de fondo:
[esta palabra clave \(MDN\)](#)

```
clase MiClase {
    nombre = "MiClase";
    obtenerNombre() {
        devuelve este.nombre;
    }
}
const c = new MiClase();
constante obj = {
    nombre: "obj",
    getNombre : c.getNombre,
};

// Imprime "obj", no "MiClase"
consola.log(obj.getName());
```

Para resumir, por defecto, el valor de this dentro de una función depende de como era la función llamada. En este ejemplo, debido a que la función fue llamada a través de la referencia obj, su valor de esto era obj en lugar de la instancia de clase.

¡Esto rara vez es lo que quieras que suceda! TypeScript proporciona algunas formas de mitigar o prevenir este tipo de error.

Funciones de flecha

Si tiene una función que a menudo se llamará de una manera que pierde su contexto , puede tener sentido usar una propiedad de función de flecha en lugar de una definición de método:

Lectura de fondo:

[Funciones de flecha \(MDN\)](#) .

clase MiClase

```
{ nombre = "MiClase";
obtenerNombre = () => {
    devuelve este.nombre; }

} const c = new MiClase(); const
g = c.getName; // Imprime
"MyClass" en lugar de bloquear console.log(g());
```

Esto tiene algunas compensaciones:

- Se garantiza que este valor sea correcto en tiempo de ejecución, incluso para el código no verificado con Mecanografiado
- Esto usará más memoria, porque cada instancia de clase tendrá su propia copia de cada función definida de esta manera.
- No puede usar super.getName en una clase derivada, porque no hay una entrada en la cadena de prototipos para obtener el método de la clase base.

estos parámetros

En una definición de método o función, un parámetro inicial denominado this tiene un significado especial en TypeScript. Estos parámetros se borran durante la compilación:

```
// Entrada de TypeScript con la función de parámetro
'this' fn(this: SomeType, x: number) {
    /* ... */
}
```

```
// Función de salida
JavaScript fn(x) { /* ... */
}
```

TypeScript comprueba que llamar a una función con este parámetro se hace con un contexto correcto.

En lugar de usar una función de flecha, podemos agregar este parámetro a las definiciones de métodos para hacer cumplir estáticamente que el método se llama correctamente:

```
clase MiClase
{ nombre = "MiClase";
  getName(this: MyClass) { return
    this.name;
  }

} const c = new MiClase();
// OK
c.getName();

// Error, fallaría const g =
c.getName; consola.log(g());
```

El contexto 'this' de tipo 'void' no se puede asignar al método 'this' de tipo 'MyClass'. de tipo 'MiClase'.

Este método hace las compensaciones opuestas del enfoque de la función de flecha:

- Las personas que llaman JavaScript aún pueden usar el método de clase incorrectamente sin darse cuenta
- Solo se asigna una función por definición de clase, en lugar de una por instancia de clase
- Las definiciones de métodos base todavía se pueden llamar a través de super .

este tipos

En las clases, un tipo especial llamado this se refiere a dinámicamente al tipo de la clase actual. Vamos a ver cómo esto es útil:

caja de clase

```
{ contenido: cadena = "";
  establecer (valor: cadena) {
    (método) Box.set(valor: cadena): esto
    este.contenido = valor;
    devolver esto;
  }
}
```

Aquí, TypeScript infirió que el tipo de retorno del conjunto es esta subclase , en lugar de Caja . Ahora vamos a hacer un de Box :

```
clase ClearableBox extiende Box { clear()
  { this.contents = "";
  }
}

const a = new ClearableBox(); const b
= a.set("hola");
```

constante b: ClearableBox

También puede usar esto en una anotación de tipo de parámetro:

```
caja de clase
{ contenido: cadena = "";
  mismoAs(otro: este) { return
    otro.contenido === este.contenido;
  }
}
```

Esto es diferente de escribir otro: Box : si tiene una clase derivada, su método sameAs ahora solo aceptará otras instancias de esa misma clase derivada:

caja de clase

```
{ contenido: cadena = "";
mismoAs(otro: este) { return
    otro.contenido === este.contenido;
}
}
```

class DerivedBox extiende Box

```
{ otherContent: string = "?";
}
```

const base = caja nueva ();

const derivada = nueva CajaDerivada();

derivado.igual que(base);

de tipo 'Box' no se puede asignar al parámetro de tipo **DerivedBox**. El argumento 'CuadroDerivado'.

Falta la propiedad 'otroContenido' en el tipo 'Cuadro' pero se requiere en el tipo 'CuadroDerivado'.

este tipo basado en guardias

Puede usar `this is Type` en la posición de retorno para métodos en clases e interfaces. Cuando se mezcla con un tipo de restricción (por ejemplo , declaraciones if), el tipo del objeto de destino se reduciría al Tipo especificado .

```
clase ObjetoSistemaArchivo {
    isFile(): esto es FileRep {
        devolver esta instancia de FileRep;

    } isDirectory(): este es Directory { devuelve
        esta instancia de Directory;

    } isNetworked(): esto está en red y esto {
        devuelve esto.en red;

    } constructor ( ruta pública: cadena, red privada : booleano) {}
}

class FileRep extiende FileSystemObject { constructor
(ruta: cadena, contenido público : cadena) {
    super(ruta, falso);
}
}

class Directory extiende FileSystemObject { niños:
FileSystemObject[];
}

interfaz En red { host:
cadena;
}

const fso: FileSystemObject = new FileRep("foo/bar.txt", "foo");

si (fso.isFile()) { fso.content;

    const fso: FileRep

} else if (fso.isDirectory()) { fso.children;

    const fso: Directorio

} else if (fso.isNetworked()) { fso.host;

    const fso: en red y FileSystemObject

}
```

Un caso de uso común para una protección de tipo basada en `this` es permitir la validación diferida de un campo en particular. Por ejemplo, este caso elimina un indefinido del valor que se encuentra dentro del cuadro cuando se ha verificado que `hasValue` es verdadero:

```
class Box<T>
{ valor?: T;

  hasValue(): esto es { valor: T } { devuelve este
    valor !== indefinido;
  }
}

caja const = caja nueva ();
caja.valor = "Gameboy";

caja.valor; (propiedad) Box<desconocido>.valor?: desconocido

if (caja.tieneValor())
{ caja.valor; (propiedad) valor: desconocido
}
}
```

Propiedades de parámetros

TypeScript ofrece una sintaxis especial para convertir un parámetro de constructor en una propiedad de clase con el mismo nombre y valor. Estos se llaman propiedades antepuestas al parámetro o modificadores de visibilidad y son `public`, `private`, `protected`, `readonly`. El campo resultante obtiene esos modificadores:

0

```
class Params
{ constructor
  ( public readonly x: number,
  protected y: number, private z:
  number ) {

  // No se necesita cuerpo
}
```

```
} const a = new Params(1, 2, 3);
consola.log(ax);
```

(propiedad) Params.x: número

```
consola.log(esto);
```

La propiedad 'z' es privada y solo se puede acceder a ella dentro de la clase 'Params'.

Expresiones de clase

Las expresiones de clase son muy similares a las declaraciones de clase. La única diferencia real es que las expresiones de clase no necesitan un nombre, aunque podemos referirnos a ellas a través de cualquier identificador al que terminaron vinculados:

Lectura de fondo:
[Expresiones de clase \(MDN\)](#)

```
const algunaClase = clase<Tipo>
{ contenido: Tipo; constructor(valor:
  Tipo) { this.content = valor;

} };

const m = new someClass("Hola, mundo");
```

const m: algunaClase<cadena>

clases abstractas y miembros

Las clases, métodos y campos en TypeScript pueden ser `resumen`.

Un método abstracto o campo abstracto es uno que no ha tenido una implementación proporcionada. Estas miembros deben existir dentro de un `clase abstracta`, que no se puede instanciar directamente.

El papel de las clases abstractas es servir como clase base para las subclases que implementan todos los miembros abstractos. Cuando una clase no tiene miembros abstractos, se dice que es `concreto`.

Veamos un ejemplo:

```
clase abstracta Base {  
    resumen getNombre(): cadena;  
  
    imprimirNombre() {  
        consola.log("Hola, " + this.getName());  
    }  
}  
  
const b = nueva Base();
```

No se puede crear una instancia de una clase abstracta.

No podemos instanciar `Base` con `new` porque es abstracto. En su lugar, necesitamos hacer una clase derivada e implementar los miembros abstractos:

```
clase Derivado extiende Base {  
    obtenerNombre() {  
        devolver "mundo";  
    }  
}  
  
const d = nuevo Derivado();  
d.imprimirNombre();
```

Tenga en cuenta que si olvidamos implementar los miembros abstractos de la clase base, obtendremos un error:

```
clase Derivado extiende Base {
```

implementa el miembro abstracto heredado 'getName' de la clase 'Base'.
clase no abstracta 'Derivado' no 'Base'.

```
// olvidé hacer algo  
}
```

Firmas de construcciones abstractas

A veces, desea aceptar alguna función de constructor de clase que produzca una instancia de una clase que se deriva de alguna clase abstracta.

Por ejemplo, es posible que desee escribir este código:

```
function saludar(ctor: typeof Base) { const instancia = new  
ctor();
```

No se puede crear una instancia de una clase abstracta.

```
instancia.printName();  
}
```

TypeScript le dice correctamente que está intentando crear una instancia de una clase abstracta. Después de todo, dada la definición de ~~get nombre~~, es perfectamente legal escribir este código, que terminaría construyendo

```
// ¡Malo!  
saludar(Base);
```

En su lugar, desea escribir una función que acepte algo con una firma de construcción:

```

función saludar(ctor: nuevo () => Base) { const
  instancia = nuevo ctor(); instancia.printName();

} saludo (derivado);
saludar(Base);

```

tipo 'typeof Base' no se puede asignar al parámetro de tipo 'new () => Base'. El argumento `ctor`.

abstract No se puede asignar un tipo de constructor abstracto a un tipo de constructor no abstracto. `tipo constructor`.

Ahora TypeScript le informa correctamente sobre qué funciones de constructor de clase se pueden invocar:

Derivado `puedo` porque es concreto, pero Base `no puede`.

Relaciones entre clases

En la mayoría de los casos, las clases en TypeScript se comparan estructuralmente, al igual que otros tipos.

Por ejemplo, estas dos clases se pueden usar una en lugar de la otra porque son idénticas:

```

clase Punto1 { x =
  0; y = 0;

}

clase Punto2 { x =
  0; y = 0;

}

// OK
const p: Punto1 = nuevo Punto2();

```

De manera similar, las relaciones de subtipo entre clases existen incluso si no hay una herencia explícita:

```
clase Persona
{ nombre:
  cadena; edad: número;
}

clase Empleado
{ nombre: cadena;
edad: número;
salario: número;
}

// OK
const p: Persona = nuevo Empleado();
```

Esto suena sencillo, pero hay algunos casos que parecen más extraños que otros.

Las clases vacías no tienen miembros. En un sistema de tipo estructural, un tipo sin miembros es generalmente un supertipo de cualquier otra cosa. Entonces, si escribe una clase vacía (¡no lo haga!), se puede usar cualquier cosa en su lugar:

```
clase vacía {}

function fn(x: Empty) { // no
  puedo hacer nada con 'x', así que no lo haré
}

// ¡Todo bien!
fn(ventana);
fn({}); fn(fn);
```

Módulos

JavaScript tiene una larga historia de diferentes formas de manejar el código de modularización. TypeScript, que existe desde 2012, ha implementado soporte para muchos de estos formatos, pero con el tiempo, la comunidad y la especificación de JavaScript han convergido en un formato llamado Módulos ES (o módulos ES6). Es posible que lo conozca como la sintaxis de importación / exportación .

ES Modules se agregó a la especificación de JavaScript en 2015 y, para 2020, tenía un amplio soporte en la mayoría de los navegadores web y tiempos de ejecución de JavaScript.

Para enfocarse, el manual cubrirá tanto los Módulos ES como su popular precursor CommonJS module.exports = sintaxis, y puede encontrar información sobre los otros patrones de módulos en la sección de referencia en [Módulos](#).

Cómo se definen los módulos de JavaScript

En TypeScript, al igual que en ECMAScript 2015, cualquier archivo que contenga una importación o exportación de nivel superior se considera un módulo.

Por el contrario, un archivo sin declaraciones de importación o exportación de nivel superior se trata como un script cuyo contenido está disponible en el ámbito global (y, por lo tanto, también para los módulos).

Los módulos se ejecutan dentro de su propio ámbito, no en el ámbito global. Esto significa que las variables, funciones, clases, etc. declaradas en un módulo no son visibles fuera del módulo a menos que se exporten explícitamente mediante uno de los formularios de exportación. Por el contrario, para consumir una variable, función, clase, interfaz, etc. exportada desde un módulo diferente, debe importarse utilizando uno de los formularios de importación.

no módulos

Antes de comenzar, es importante comprender lo que TypeScript considera un módulo. La especificación de JavaScript declara que cualquier archivo de JavaScript sin una exportación o una espera de nivel superior debe considerarse una secuencia de comandos y no un módulo.

Dentro de un archivo de secuencia de comandos, las variables y los tipos se declaran en el ámbito global compartido, y se supone que usará el `outFile` opción del compilador para unir varios archivos de entrada en un archivo de salida, o usar varias etiquetas `<script>` en su HTML para cargar estos archivos (en el orden correcto!).

Si tiene un archivo que actualmente no tiene ninguna importación o exportación , pero desea que se lo trate como un módulo, agregue la línea:

```
exportar {};
```

que cambiará el archivo para que sea un módulo que no exporte nada. Esta sintaxis funciona independientemente del objetivo de su módulo.

Módulos en TypeScript

Hay tres cosas principales a tener en cuenta al escribir código basado en módulos en TypeScript:

- **Sintaxis:** ¿Qué sintaxis quiero usar para importar y exportar cosas?
- **Resolución del módulo:** ¿Cuál es la relación entre los nombres de los módulos (o rutas) y los archivos en el disco?
- **Destino de salida del módulo:** ¿Cómo debería ser mi módulo de JavaScript emitido?

Lectura adicional:

[Impaciente JS \(Módulos\)](#)

[MDN: Módulos JavaScript](#)

Sintaxis del módulo ES

Un archivo puede declarar una exportación principal a través de la exportación predeterminada :

```
// @filename: hello.ts función
predeterminada de exportación helloWorld()
  { console.log("¡Hola, mundo!");
}
```

Esto luego se importa a través de:

```
importar helloWorld desde "./hello.js"; Hola
Mundo();
```

Además de la exportación predeterminada, puede tener más de una exportación de variables y funciones a través de la exportación omitiendo default :

```
// @filename: maths.ts exportar
var pi = 3.14; exportar let
squareTwo = 1.41; constante de
exportación phi = 1,61;

clase de exportación RandomNumberGenerator {}

función de exportación absoluta (num: numero) { if (num
< 0) return num -1; número de retorno ;

}
```

Estos se pueden usar en otro archivo a través de la sintaxis de importación :

```
importar { pi, phi, absoluto } desde "./maths.js";

consola.log(pi); const
absPhi = absoluto(phi);

const absPhi: número
```

Sintaxis de importación adicional

Se puede cambiar el nombre de una importación utilizando un formato como `import {old as new}` :

```
importar { pi como pi } desde "./maths.js";

consola.log(pi);
(alias) var pi: número de
importación
```

Puede mezclar y combinar la sintaxis anterior en una sola importación :

```
// @filename: maths.ts export
const pi = 3.14; exportar clase
predeterminada RandomNumberGenerator {}

// @filename: app.ts import
RandomNumberGenerator, { pi como } de "./maths.js";
```

Generador de números aleatorios;

(alias) clase RandomNumberGenerator import
RandomNumberGenerator

consola.log();

(alias) const : 3.14 import

Puede tomar todos los objetos exportados y ponerlos en un solo espacio de nombres usando *

como nombre :

```
// @filename: app.ts import *
as matemáticas from "./maths.js";

consola.log(matemáticas.pi);
const positivoPhi = matemáticas.absoluto(matemáticas.phi);
```

const positivoPhi: número

Puede importar un archivo y no incluya cualquier variable en su módulo actual a través de la importación
"./archivo" :

```
// @nombre de archivo: app.ts
import "./maths.js";

consola.log("3.14");
```

En este caso, la importación no hace nada. Sin embargo, se evaluó todo el código en `maths.ts`, lo que podría desencadenar efectos secundarios que afectarían a otros objetos.

Sintaxis del módulo ES específico de TypeScript

Los tipos se pueden exportar e importar utilizando la misma sintaxis que los valores de JavaScript:

```
// @filename: animal.ts tipo de
exportación Cat = { raza: cadena; añoDeNacimiento: número };

exportar interfaz Perro { razas:
  cadena []; añoDeNacimiento:
  número;
}

// @nombredearchivo:
app.ts import { Gato, Perro } de "./animal.js"; tipo
Animales = Gato | Perro;
```

TypeScript ha ampliado la sintaxis de importación con dos conceptos para declarar una importación de un tipo:

tipo de importación

Que es una declaración de importación que puede

solamente tipos de importación:

```
// @filename: animal.ts tipo de
exportación Cat = { raza: cadena; añoDeNacimiento: número };

puede usar como valor porque se importó usando import type 'tipo de importación' 'CatName' no se

tipo de exportación Perro = { razas: cadena []; añoDeNacimiento: número }; export
const createCatName = () => "esponjoso";

// @filename: tipo de
importación válido.ts { Gato, Perro } de "./animal.js"; tipo de
exportación Animales = Gato | Perro;

// @nombre de archivo:
tipo de importación app.ts { createCatName } de "./animal.js"; const
nombre = createCatName();
```

Importaciones de tipo en línea

TypeScript 4.5 también permite que las importaciones individuales tengan un prefijo de tipo para indicar que la referencia importada es un tipo:

```
// @filename: app.ts import
{ createCatName, type Cat, type Dog } from "./animal.js";

tipo de exportación Animales = Gato | Perro;
const nombre = createCatName();
```

Juntos, estos permiten que un transpilador que no sea TypeScript como Babel, swc o esbuild sepa qué importaciones se pueden eliminar de manera segura.

Sintaxis del módulo ES con comportamiento CommonJS

TypeScript tiene una sintaxis de módulo ES que se ~~correlaciona~~ con CommonJS y AMD require .

Las importaciones que usan ES Module para la implementación de los módulos son las mismas que las requeridas de esos entornos, pero esta sintaxis garantiza que tenga una coincidencia de 1 a 1 en su archivo TypeScript con la salida de CommonJS:

```
importar fs = require("fs"); const
código = fs.readFileSync("hello.ts", "utf8");
```

Puede obtener más información sobre esta sintaxis en la [página de referencia de los módulos](#).

Sintaxis de CommonJS

CommonJS es el formato en el que se entregan la mayoría de los módulos en npm. Incluso si está escribiendo utilizando la sintaxis de Módulos ES anterior, tener una breve comprensión de cómo funciona la sintaxis de CommonJS lo ayudará a depurar más fácilmente.

Exportador

Los identificadores se exportan configurando la propiedad de exportación en un módulo llamado global .

```
función absoluta(num: numero) { if (num < 0)
    return num * -1;
}

módulo.exportaciones =
{ pi: 3,14, squareTwo:
  1,41, phi: 1,61, absoluto, };
```

Luego, estos archivos se pueden importar a través de una declaración requerida :

```
const matemáticas = require("matemáticas");
matemáticas.pi;
```

ningún

O puede simplificar un poco usando la función de desestructuración en JavaScript:

```
const { cuadradoDos } = require("matemáticas"); cuadradoDos;
```

const squareTwo: cualquiera

Interoperabilidad de los módulos CommonJS y ES

Hay una falta de coincidencia en las características entre los módulos CommonJS y ES con respecto a la distinción entre una importación predeterminada y una importación de objeto de espacio de nombres de módulo. TypeScript tiene un indicador de compilación para reducir la fricción entre los dos conjuntos diferentes de [restricciones con esModuleInterop](#).

Opciones de resolución del módulo de TypeScript

La resolución del módulo es el proceso de tomar una cadena de la instrucción import o require y determinar a qué archivo se refiere esa cadena.

TypeScript incluye dos estrategias de resolución: Classic y Node. Clásico, el predeterminado cuando el [módulo](#) de opción del compilador no es [commonjs](#), se incluye para compatibilidad con Node.js. Sin embargo, solo se incluye para las estrategias adicionales para .ts y

.d.ts .

Hay muchos indicadores de TSConfig que influyen en la estrategia del módulo dentro de TypeScript:

[moduleResolution](#), [URL base](#), [caminos](#), [rootDirs](#)

Para conocer todos los detalles sobre cómo funcionan estas estrategias, puede consultar la [Resolución del Módulo](#).

Opciones de salida del módulo de TypeScript

Hay dos opciones que afectan la salida de JavaScript emitida:

- [objetivo](#) que determina qué funciones de JS se reducen (se convierten para ejecutarse en versiones anteriores) tiempos de ejecución de JavaScript) y que se dejan intactos
- [módulo](#) que determina qué código se usa para que los módulos interactúen entre sí

que [objetivo](#) que [usa](#) está determinado por las funciones disponibles en el tiempo de ejecución de JavaScript en el que espera ejecutar el código TypeScript. Eso podría ser: el navegador web más antiguo que admite, la versión más baja de Node.js en la que espera ejecutar o podría provenir de restricciones únicas de su tiempo de ejecución, como Electron, por ejemplo.

Toda la comunicación entre módulos ocurre a través de un cargador de módulos, el [módulo](#) de opción del compilador [determina](#) cuál se utiliza. En tiempo de ejecución, el cargador de módulos es responsable de localizar y ejecutar todas las dependencias de un módulo antes de ejecutarlo.

Por ejemplo, aquí hay un archivo de TypeScript que utiliza la sintaxis de módulos ES y muestra algunas opciones diferentes para el [módulo](#):

```
importar { valueOfPi } desde "./constants.js";
```

```
export const twoPi = valueOfPi * 2;
```

```
importar { valueOfPi } desde "./constants.js"; export const
twoPi = valueOfPi * 2;
```

ComúnJS

```
"uso estricto";
Object.defineProperty(exportaciones, "__esModule", { valor: verdadero });
exportaciones.dosPi = vacío 0; const constantes_js_1 = require("./constantes.js");
exportaciones.dosPi = constantes_js_1.valorDePi *
2;
```

UMD

```
(función (fábrica) { if (tipo
de módulo === "objeto" && tipo de módulo.exportaciones === " objeto")
    var v = fábrica (requerir, exportaciones); if (v !
== indefinido) módulo.exportaciones = v;

} else if (typeof define === "función" && define.amd) {
    define(["requerir", "exportaciones", "./constantes.js"], fábrica);
}
})(función (requerir, exportaciones) { "usar
estricto";
Object.defineProperty(exportaciones, "__esModule", { valor: verdadero });
exportaciones.twoPi = void 0; const constants_js_1 = require("./constants.js ");
exportaciones.dosPi = constantes_js_1.valorDePi *
2;
});
```

Tenga en cuenta que ES2020 es efectivamente el mismo que el index.ts original .

Puede ver todas las opciones disponibles y cómo se ve su código JavaScript emitido en la [Referencia de TSConfig para el módulo](#).

Espacios de nombres de TypeScript

TypeScript tiene su propio formato de módulo llamado espacios de nombres que es anterior al estándar ES Modules. Esta sintaxis tiene muchas funciones útiles para crear archivos de definición complejos, y todavía ve un uso activo [en DefinitelyTyped](#). Si bien no están en desuso, la mayoría de las funciones en los espacios de nombres existen en los módulos ES y le recomendamos que los use para alinearse con la dirección de JavaScript. Puede obtener más información sobre los espacios de nombres en [la página de referencia de espacios de nombres](#).