



TypeScript para Principiantes

Table of Contents

TypeScript.....	1
para Principiantes.....	1
Comenzando con TypeScript.....	4
Instalando las herramientas.....	4
Hola Mundo en TypeScript.....	5
Anotaciones de tipo.....	7
Interfaces.....	9
Expresiones de Función Flecha.....	11
Clases con Modificadores de Accesibilidad Public y Private.....	13
Recursos de TypeScript.....	18
Estamos Solamente Empezando.....	19
TypeScript para Principiantes, Parte 1: Comenzando.....	20
¿Por Qué Aprender TypeScript?.....	20
Instalación.....	21
IDEs y Editores de Texto Con Soporte TypeScript.....	21
Compilando TypeScript a JavaScript.....	22
Ideas Finales.....	24
TypeScript para Principiantes, Parte 2: Tipos Básicos de Datos.....	26
El tipo de datos Null.....	26
El Tipo de Datos Indefinido.....	27
El Tipo de Datos Void.....	27
El Tipo de Datos Boolean.....	28
El Tipo de Datos Numérico.....	28
El Tipo de Datos String.....	29
Los tipos de datos Array y Tuple.....	31
El Tipo de Datos Enum.....	32
Los tipos Any y Never.....	33
Pensamientos Finales.....	33
TypeScript Para Principiantes, Parte 3: Interfaces.....	35
Creando Nuestra Primera Interfaz.....	35
Haciendo la Propiedades de la Interfaz Opcionales.....	37
Utilizando el Índice de Firmas.....	38
Propiedades de Sólo Lectura.....	39
Funciones e Interfaces.....	40
Reflexiones Finales.....	41
TypeScript Para Principiantes, Parte 4: Clases.....	42
Creando tu primera clase.....	42
Modificadores privados y públicos.....	43
Herencia en TypeScript.....	44
Usando el Modificador Protegido.....	46
Pensamientos finales.....	47
TypeScript para Principiantes, Parte 5: Genéricos.....	48
La Necesidad de Genéricos.....	48
Usando Genéricos Podría Parecer Muy Limitante.....	50
Crea Funciones Genéricas Usando Restricciones.....	52

Ideas Finales.....53

Comenzando con TypeScript

A finales de 2012, Microsoft introdujo [TypeScript](#), un superconjunto tipado para JavaScript que compila en JavaScript plano. TypeScript se centra en proporcionar herramientas útiles para aplicaciones a gran escala implementando características, como clases, anotaciones de tipo, herencia, módulos y mucho más. En este tutorial, comenzaremos con TypeScript usando ejemplos de código pequeños, compilándolos en JavaScript y viendo el resultado de forma instantánea en el navegador.

Instalando las herramientas

Las características de TypeScript solamente se aplican en el momento de la compilación

Configurará su equipo de acuerdo con su plataforma y necesidades específicas. Los usuarios de Windows y Visual Studio simplemente deben descargar el [Plugin de Visual Studio](#). Si utiliza Windows y no tiene Visual Studio, puede probar [Visual Studio Express for Web](#). Actualmente, la experiencia de TypeScript en Visual estudio es superior a la de otros editores de código.

Si utiliza una plataforma diferente (o no desea usar Visual Studio), todo lo que se necesita para usar TypeScript es un editor de texto, un navegador y el [paquete npm TypeScript](#). Siga las siguientes instrucciones:

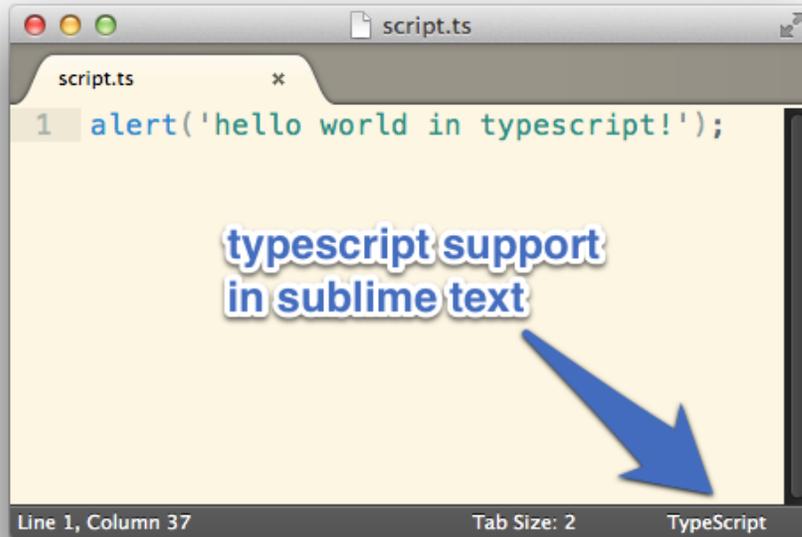
1. Instalar el [Gestor de Paquetes de Node \(npm\)](#)

```
$ curl http://npmjs.org/install.sh | sh
$ npm --version
1.1.70
```

2. Instalar el paquete npm TypeScript

```
$ npm install -g typescript
$ npm view typescript version
npm http GET https://registry.npmjs.org/typescript
npm http 304 https://registry.npmjs.org/typescript
0.8.1-1
```

3. Cualquier [navegador moderno](#): en este tutorial se usará [Chrome](#).
4. Cualquier editor de texto: en este tutorial se usará [Sublime Text](#).
5. [Plugin para resaltado de sintaxis](#) para editores de textos.



Eso es todo; ¡estamos listos para hacer una simple aplicación "Hola Mundo" ("Hello World") en TypeScript!

Hola Mundo en TypeScript

TypeScript es un superconjunto de [Ecmascript 5 \(ES5\)](#) e incorpora características propuestas para ES6. Debido a esto, cualquier programa JavaScript es ya un programa TypeScript. El compilador de TypeScript realiza transformaciones de ficheros locales en programas TypeScript. Por lo tanto, el resultado final JavaScript se asemeja bastante a la entrada en TypeScript.

En primer lugar, crearemos un fichero básico `index.html` con una referencia a un fichero de script externo.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Learning TypeScript</title>
</head>
<body>

  <script src="hello.js"></script>

</body>
</html>
```

Esta es una aplicación "Hola mundo" (Hello World) básica; así que, creemos un fichero llamado `hello.ts`. La extensión `*.ts` designa a un fichero TypeScript. Añada el siguiente código al fichero `hello.ts`:

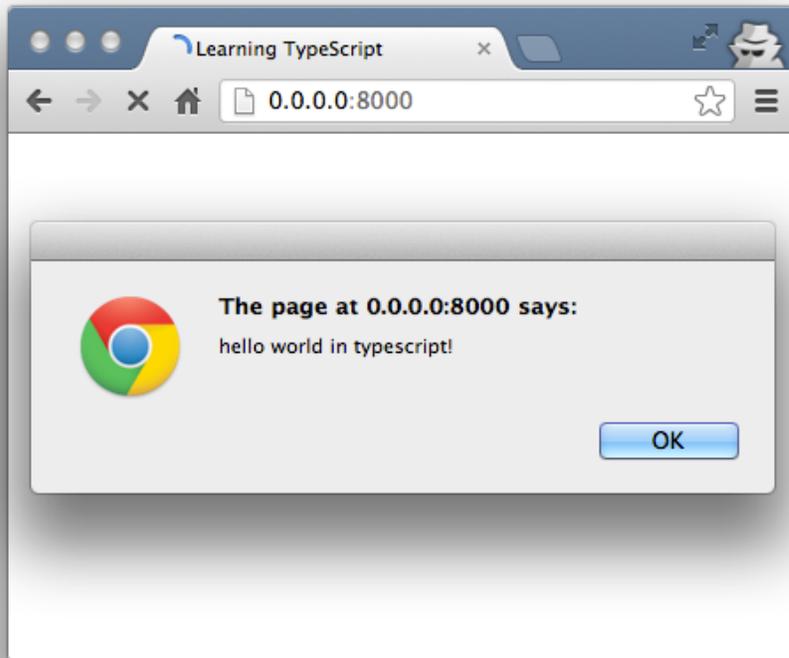
```
alert('hello world in TypeScript!');
```

En siguiente lugar, abra la interfaz de línea de comandos, navegue a la carpeta que contiene el fichero `hello.ts` y ejecute el compilador de TypeScript con el siguiente comando:

```
tsc hello.ts
```

El comando `tsc` es el compilador TypeScript, este comando generará de forma inmediata un nuevo fichero llamado `hello.js`. Nuestra aplicación TypeScript no utiliza una sintaxis específica de TypeScript por lo que veremos exactamente el mismo código JavaScript en `hello.js` que en el fichero `hello.ts` que hemos escrito.

¡Genial! Ahora podemos explorar las características de TypeScript y ver cómo puede ayudarnos a mantener y crear aplicaciones JavaScript a gran escala.

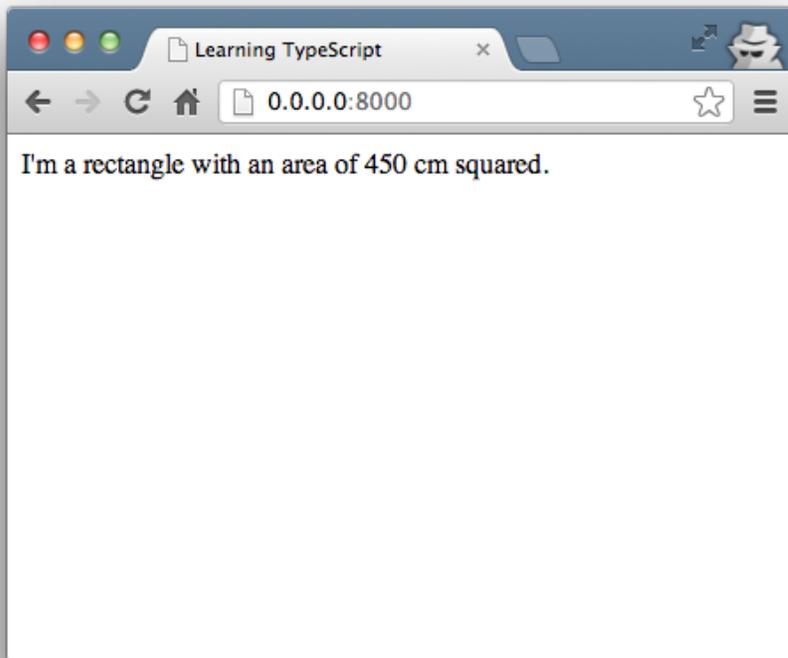


Anotaciones de tipo

Las anotaciones de tipo son una característica opcional, la cual nos permite chequear y expresar nuestras intenciones en los programas que escribimos. Creemos una función simple `area()` en un nuevo fichero TypeScript, llamado `type.ts`.

```
function area(shape: string, width: number, height: number) {  
    var area = width * height;  
    return "I'm a " + shape + " with an area of " + area + " cm squared."  
}  
  
document.body.innerHTML = area("rectangle", 30, 15);
```

Seguidamente, cambiemos el nombre del script usado en `index.html` a `type.js` y ejecutemos el compilador de TypeScript mediante `tsc type.js`. Refresque la página en el navegador y debería ver lo siguiente:



Como se muestra en el código anterior, las anotaciones de tipo se expresan como parte de los parámetros de la función; estos indican los tipos de los valores que se pueden pasar a la función. Por ejemplo, el parámetro `shape` está especificado como un valor de cadena y `width` y `height` son valores numéricos.

Las anotaciones de tipo, y otras características de TypeScript, se validan únicamente en el momento de la compilación. Si se pasa cualquier otro tipo de valor a estos parámetros, el compilador generará un error en tiempo de compilación. Este comportamiento es extremadamente útil cuando se construyen aplicaciones a gran escala. Por ejemplo, pasemos de forma intencionada un valor de cadena al parámetro `width`:

```
function area(shape: string, width: number, height: number) {
    var area = width * height;
    return "I'm a " + shape + " with an area of " + area + " cm squared.";
}

document.body.innerHTML = area("rectangle", "width", 15); // wrong width type
```

Sabemos que esto producirá un resultado no deseado, y al compilar el fichero se nos alertará del problema con el siguiente error:

```
$ tsc type.ts
type.ts(6,26): Supplied parameters do not match any signature of call target
```

Debemos tener en cuenta que, a pesar del error, el compilador genera el fichero `type.js`. El error no evita que el compilador de TypeScript genere el correspondiente fichero JavaScript, aunque el compilador nos avisa de los problemas potenciales. Teníamos la intención de que `width` fuese un número, pasando cualquier otro tipo produce un comportamiento no deseado en nuestro código. Otras anotaciones de tipo incluyen `bool` o incluso `any`.

Interfaces

Expandamos nuestro ejemplo para incluir una **interfaz** que más adelante describa el parámetro `shape` como un objeto con una propiedad `color` opcional. Creamos un nuevo fichero llamado `interface.ts` y modificamos la referencia al script en `index.html` para que incluya `interface.js`. Insertemos el siguiente código en `interface.ts`:

```
interface Shape {
  name: string;
  width: number;
  height: number;
  color?: string;
}

function area(shape : Shape) {
  var area = shape.width * shape.height;
  return "I'm " + shape.name + " with area " + area + " cm squared";
}

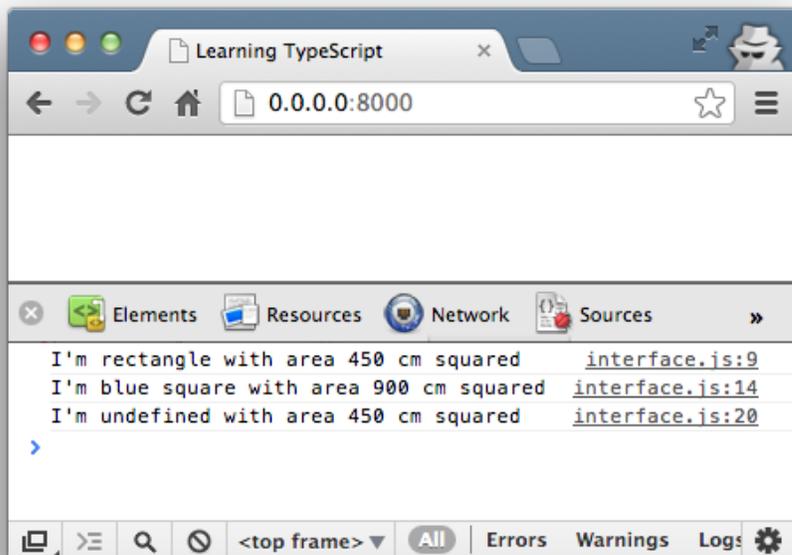
console.log( area( {name: "rectangle", width: 30, height: 15} ) );
console.log( area( {name: "square", width: 30, height: 30, color: "blue"} ) );
```

Las interfaces son nombres dados a tipos de objetos. No solamente podemos declarar una interfaz sino que también podemos usarla como anotación de tipo.

La compilación de `interface.js` no producirá errores. Para provocar un error, añadamos otra línea de código a `interface.js` usando un objeto shape que no tenga propiedad nombre y veamos el resultado en el navegador. Añadamos esta línea a `interface.js`:

```
console.log( area( {width: 30, height: 15} ) );
```

Ahora, compilemos el código con `tsc interface.js`. Recibirá un error, pero no se preocupe por eso ahora mismo. Refresque el navegador y observe la consola. Verá algo similar a la siguiente imagen:



Ahora miremos al error. Este es:

```
interface.ts(26,13): Supplied parameters do not match any signature of call target:
Could not apply type 'Shape' to argument 1, which is of type '{ width: number; height:
number; }'
```

Vemos este error debido a que el objeto pasado a `area()` no se corresponde con la interfaz `Shape`; necesita tener la propiedad `name` para que así sea.

Expresiones de Función Flecha

Entender el ámbito de la palabra `this` puede ser complejo y TypeScript lo hace un poco más fácil al soportar las expresiones de función flecha, [una nueva característica que está siendo discutida para ECMAScript 6](#). Las funciones flecha preservan el valor de `this`, haciendo mucho más fácil de escribir y usar funciones de devolución de llamada o callback. Considere el siguiente código:

```

var shape = {
  name: "rectangle",
  popup: function() {

    console.log('This inside popup(): ' + this.name);

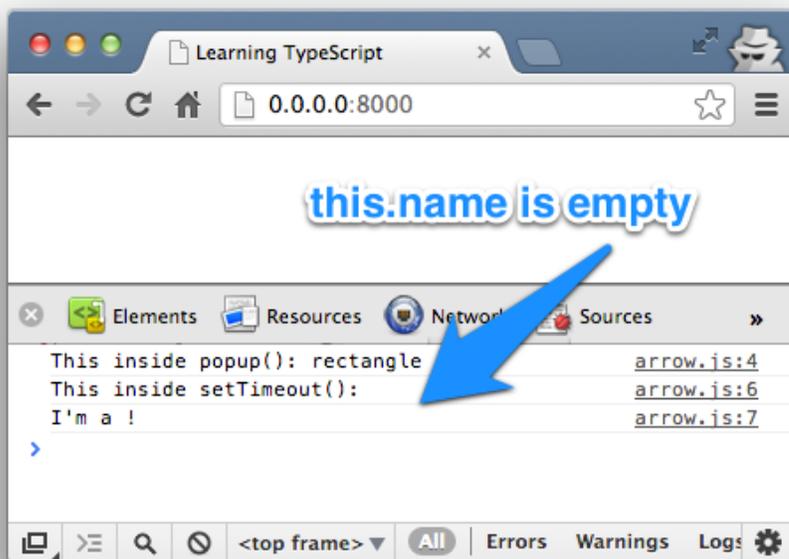
    setTimeout(function() {
      console.log('This inside setTimeout(): ' + this.name);
      console.log("I'm a " + this.name + "!");
    }, 3000);

  }
};

shape.popup();

```

El valor de `this.name` en la línea nueve estará claramente vacío, como se demuestra en la consola del navegador:



Podemos solucionar este problema fácilmente usando una función flecha de TypeScript. Simplemente debemos sustituir `function()` por `() =>`.

```

var shape = {
  name: "rectangle",
  popup: function() {

    console.log('This inside popup(): ' + this.name);

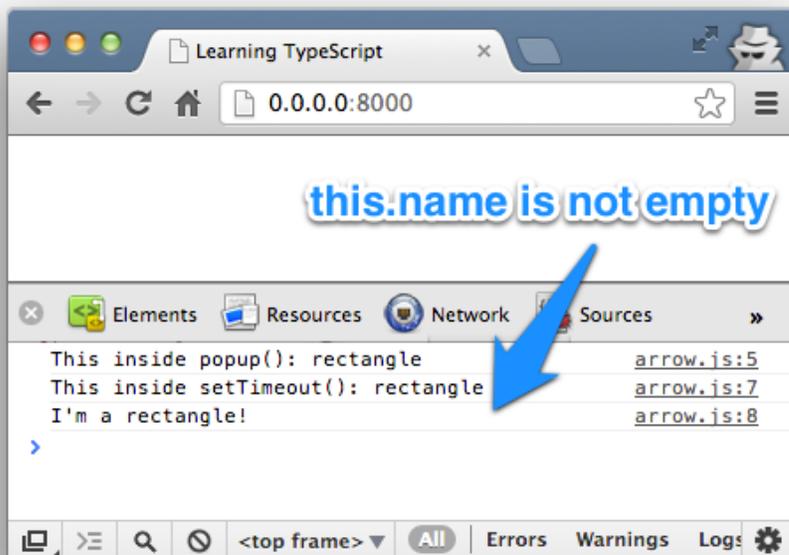
    setTimeout( () => {
      console.log('This inside setTimeout(): ' + this.name);
      console.log("I'm a " + this.name + "!");
    }, 3000);

  }
};

shape.popup();

```

Y el resultado será:



Echemos un vistazo al fichero JavaScript generado. Veremos que el compilador a inyectado una nueva variable, `var _this = this;`, y la usa en la función de devolución de llamada de `setTimeout()` para hacer referencia a la propiedad `name`.

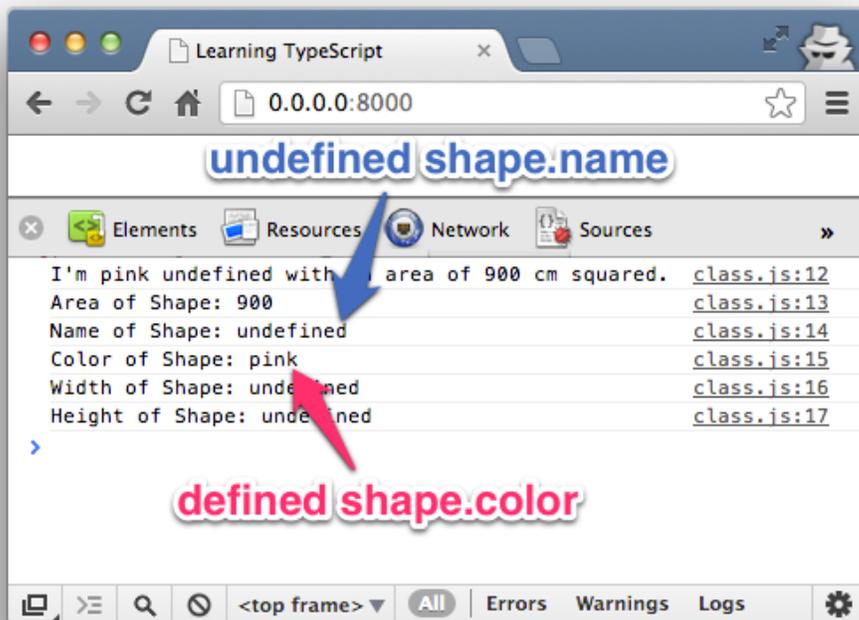
Clases con Modificadores de Accesibilidad Public y Private

TypeScript soporta clases y su implementación [está cerca de la propuesta en ECMAScript 6](#). Creemos otro fichero llamado `class.ts` y repasemos la sintaxis de las clases:

```
class Shape {  
  
    area: number;  
    color: string;  
  
    constructor ( name: string, width: number, height: number ) {  
        this.area = width * height;  
        this.color = "pink";  
    };  
};
```

La clase `Shape`, arriba definida, tiene dos propiedades, `area` y `color`, un constructor (acertadamente llamado `constructor()`), así como un método `shoutout()`. El ámbito de los argumentos del constructor (`name`, `width` y `height`) es local al constructor. Este es el motivo por el cual veremos errores en el navegador, así como en el compilador:

```
class.ts(12,42): The property 'name' does not exist on value of type 'Shape'  
class.ts(20,40): The property 'name' does not exist on value of type 'Shape'  
class.ts(22,41): The property 'width' does not exist on value of type  
'Shape'  
class.ts(23,42): The property 'height' does not exist on value of type  
'Shape'
```



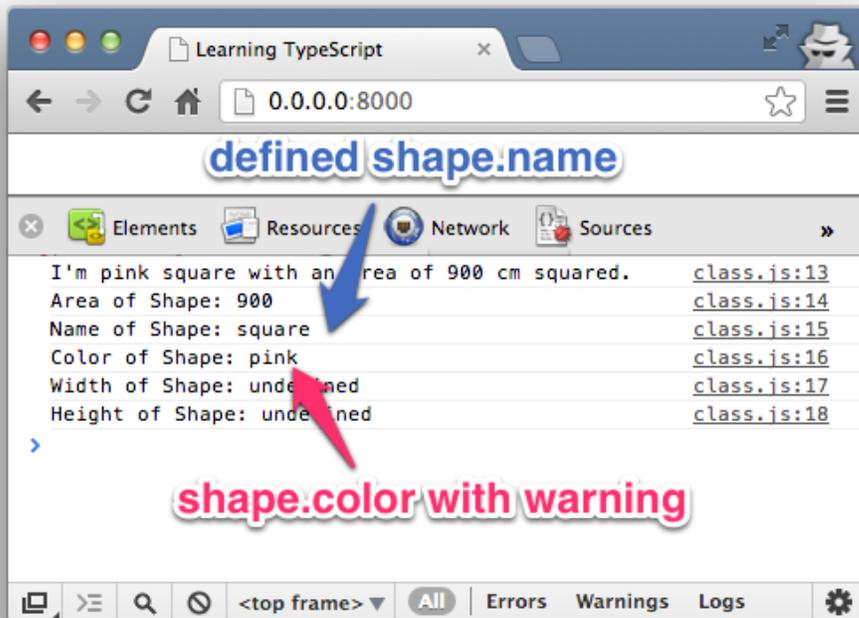
Cualquier programa JavaScript es ya un programa TypeScript.

Seguidamente, exploremos los modificadores de accesibilidad `public` y `private`. A los miembros públicos se puede acceder desde cualquier parte, mientras que a los miembros privados solamente se puede acceder dentro del ámbito de la clase. No hay, por supuesto, ninguna característica en JavaScript que valide la privacidad, por lo tanto la accesibilidad privada solamente se puede validar en el momento de la compilación y nos servirá como una advertencia a la intención original del desarrollador de hacerla privada.

Como ilustración, añadamos el modificador de accesibilidad `public` al argumento del constructor `name`, y el modificador de accesibilidad `private` a `color`. Cuando añadimos `public` o `private` a un argumento del constructor ese argumento, automáticamente, se convierte en un miembro de la clase con el modificador de accesibilidad relevante.

```
...
private color: string;
...
constructor ( public name: string, width: number, height: number )
{
...

```



```
class.ts(24,41): The property 'color' does not exist on value of type 'Shape'
```

Finalmente, podemos extender una clase existente y crear una clase derivada de ella con la palabra clave `extends`. Agreguemos el siguiente código al fichero `class.ts` existente, y compilémoslo:

```

class Shape3D extends Shape {

    volume: number;

    constructor ( public name: string, width: number, height: number, length: number ) {
        super( name, width, height );
        this.volume = length * this.area;
    };

    shoutout() {
        return "I'm " + this.name + " with a volume of " + this.volume + " cm cube.";
    }

    superShout() {
        return super.shoutout();
    }
}

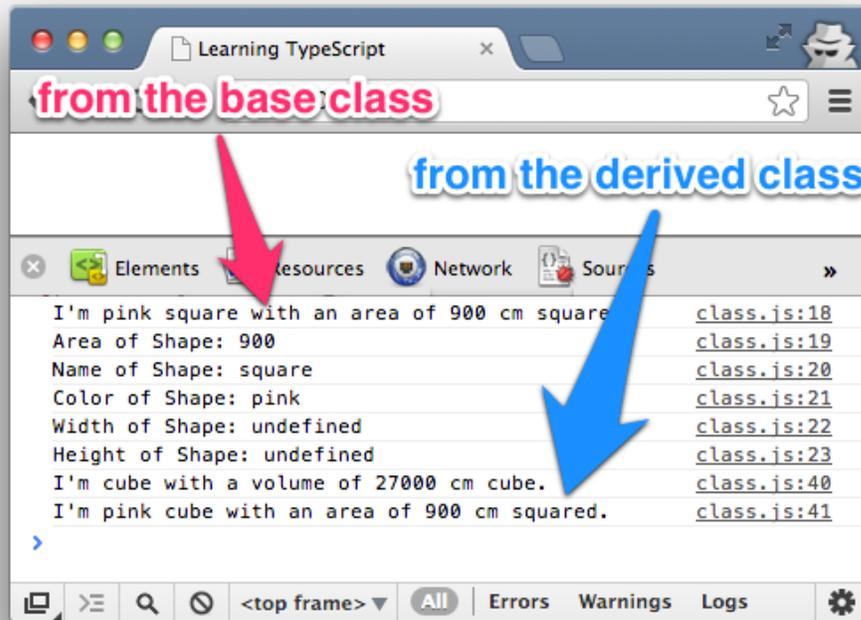
var cube = new Shape3D("cube", 30, 30, 30);
console.log( cube.shoutout() );
console.log( cube.superShout() );

```

Unas cuantas cosas están ocurriendo con la clase derivada `Shape3D`:

- Debido a que es derivada de la clase `Shape`, esta clase hereda las propiedades `area` y `color`.
- Dentro del método constructor, le método `super` llama al constructor de la clase base, `Shape`, pasándole los valores de `name`, `width` y `height`. La herencia nos permite reutilizar código de `Shape`, por lo que podemos fácilmente calcular `this.volume` con la propiedad heredada `area`.
- El método `shoutout()` sobrescribe la la implementación de la clase base y un nuevo método, `superShout()`, llama directamente al método `shoutout()` de la clase base usando la palabra clave `super`.

Con solamente unas pocas líneas adicionales de código, podemos extender fácilmente la clase base para añadir una funcionalidad específica y dar a conocer nuestra intención por medio de TypeScript.



Recursos de TypeScript

A pesar de que TypeScript tiene una edad extremadamente joven, se pueden encontrar muchos grandes recursos sobre el lenguaje en la web (¡incluyendo un curso completo que vendrá a [Tuts+ Premium!](#)) Asegúrese de mirar estos:

- [Sitio web principal](#)
- [Tutoriales y ejemplos](#)
- [Editor de código en vivo con salida JavaScript](#)
- [Repositorio de código para discusiones y seguimiento de errores](#)
- [¿Qué es TypeScript?](#) por el equipo de desarrollo
- [Lista de documentación para TypeScript](#)

Estamos Solamente Empezando

TypeScript soporta clases, y su implementación [está cerca de la propuesta en ECMAScript 6.](#)

Probar TypeScript es fácil. Si te divierte una aproximación más estáticamente tipada a aplicaciones grandes, entonces las características de TypeScript validarán un ambiente familiar y disciplinado. Aunque ha sido comparado con CoffeeScript o Dart, TypeScript es diferente ya que no reemplaza JavaScript; este añade características a JavaScript.

Tenemos todavía que ver como evoluciona TypeScript, pero Microsoft ha indicado que continuarán manteniendo sus muchas características (anotaciones de tipo aparte) alineadas con ECMAScript 6. Así que, si le gustaría probar muchas de las nuevas características de ES6, ¡TypeScript es una excelente forma de hacerlo! Adelante - ¡Dele una oportunidad!

TypeScript para Principiantes, Parte 1: Comenzando

Comencemos este tutorial con la pregunta: "¿Qué es [TypeScript](#)?"

TypeScript es un super conjunto tipado de JavaScript que compila a JavaScript plano. Como una analogía, si JavaScript fuera CSS entonces TypeScript sería SCSS.

Todo el código válido JavaScript que escribes también es válido en código TypeScript. Sin embargo, con TypeScript, usas un tipado estático y las últimas características que se compilan a JavaScript plano, el cuál es soportado por todos los navegadores. TypeScript está pensado para solucionar el problema de hacer JavaScript escalable, y hace un muy buen trabajo.

En este tutorial, comenzarás leyendo sobre diferentes características de TypeScript y por qué aprenderlo es una buena idea. El resto de las secciones en el artículo cubrirán la instalación y compilación de TypeScript junto con algunos editores populares de texto que te ofrecen soporte para la sintaxis TypeScript y otras importantes características.

¿Por Qué Aprender TypeScript?

Si nunca has usado TypeScript, podrías estarte preguntando por qué deberías molestarte sobre aprenderlo en absoluto cuando este compila a JavaScript al final.

Déjame asegurarte que no tendrás que pasar mucho tiempo aprendiendo TypeScript. Tanto TypeScript como JavaScript tienen sintaxis muy similar, y puedes solo renombrar tus archivos `.js` a `.ts` y comenzar a escribir código TypeScript. Aquí hay algunas características que deberían convencerte de comenzar a aprender TypeScript.

1. A diferencia de JavaScript, que es dinámicamente tipado, TypeScript te permite usar tipado estático. Esta característica por si misma hace al código más mantenible y reduce en gran medida la propensión de errores que podrían haber sido causados debido a asunciones incorrectas sobre el tipo de ciertas variables. Como un bono adicional, TypeScript puede determinar el tipo de una variable basado en su uso sin

que especifiques de manera explícita un tipo. Sin embargo, siempre deberías especificar los tipos para una variable de forma explícita para claridad.

2. Para ser honesto, JavaScript no fue diseñado para servir como un lenguaje de desarrollo de gran escala. TypeScript agrega todas estas características faltantes a JavaScript que lo hacen realmente escalable. Con tipado estático, el IDE que estás usando ahora podrá entender el código que estás escribiendo de mejor manera. Esto le da al IDE la habilidad de proporcionar características como completado de código y refactorización segura. Todo esto resulta en una mejor experiencia de desarrollo.

3. TypeScript también te permite usar las últimas características de JavaScript en tu código sin preocuparte sobre soporte de navegador. Una vez que has escrito el código, puedes compilarlo a JavaScript plano soportado por todos los navegadores con facilidad.

4. Muchos marcos de trabajo populares como Angular e Ionic están usando ahora TypeScript. Esto significa que si alguna vez decides usar cualquiera de los marcos de trabajo en el futuro, aprender TypeScript ahora es una buena idea.

Instalación

Antes de que comiences a escribir código TypeScript asombroso, necesitas instalar TypeScript primero. Esto puede ser hecho con la ayuda de [npm](#). Si no tienes npm instalado, tendrás que [instalar npm](#) primero antes de instalar TypeScript. Para instalar TypeScript, necesitas iniciar la terminal y ejecutar el siguiente comando:

```
npm install -g typescript
```

Una vez que la instalación está completa, puedes revisar la versión TypeScript que fue instalada ejecutando el comando `tsc -v` en tu terminal. Si todo se instaló apropiadamente, verás el número de versión TypeScript instalada en la terminal.

IDEs y Editores de Texto Con Soporte TypeScript

TypeScript fue creado por Microsoft. Así que el hecho de que [Visual Studio](#) fue el primer IDE en soportar TypeScript no debería ser una sorpresa. Una vez que TypeScript comenzó a ganar popularidad, más y más editores e IDEs agregaron soporte para este lenguaje ya sea de manera integrada o por medio de complementos. Otro ligero editor de código abierto llamado [Visual Studio Code](#), creado por Microsoft, tiene soporte integrado para TypeScript. De manera similar, [WebStorm](#) también tiene soporte integrado para TypeScript.

Microsoft también ha creado un [Complemento Sublime TypeScript](#) gratuito. NetBeans tiene un [complemento TypeScript](#) que proporciona variedad de características para facilidad de desarrollo. El resalte de sintaxis está disponible en Vim y Notepad++ con la ayuda de los complementos [typescript-vim](#) y [notepadplus-typescript](#) respectivamente.

Puedes ver una lista completa de todos los editores de texto e IDEs que soportan TypeScript en [GitHub](#). Para los ejemplos en esta serie, estaré usando Visual Studio Code para escribir todo el código.

Compilando TypeScript a JavaScript

Digamos que has escrito algún código TypeScript en un archivo `.ts`. Los navegadores no podrán ejecutar este código por si mismos. Tendrás que compilar el TypeScript a JavaScript plano que pueda ser entendido por navegadores.

Si estás usando un IDE, el código puede ser compilado a JavaScript en el IDE mismo. Por ejemplo, Visual Studio te permite directamente [compilar el código TypeScript a JavaScript](#). Tendrás que crear un archivo `tsconfig.json` en donde especifiques todas las opciones de configuración para compilar tu archivo TypeScript a JavaScript.

La aproximación más amigable para principiantes cuando no estás trabajando en un proyecto grande es usar la terminal misma. Primero, tienes que moverte a la ubicación del archivo TypeScript en la terminal y después ejecutar el siguiente comando.

```
1 tsc first.ts
```

Esto creará un nuevo archivo llamado first.js en la misma ubicación. Ten en mente que si ya tienes un archivo llamado first.js, será sobrescrito.

Si quieres compilar todos los archivos dentro del directorio actual, puedes hacerlo con la ayuda de comodines. Recuerda que esto no funcionará de manera recursiva.

```
tsc *.ts
```

Finalmente, solo puedes compilar archivos específicos proporcionando sus nombres de manera explícita en una sola línea. En tales casos, los archivos JavaScript serán creados con nombres de archivos correspondientes.

```
tsc first.ts product.ts bot.ts
```

Echemos un vistazo a siguiente programa, el cuál multiplica dos números en TypeScript.

```
let a: number = 12;
let b: number = 17;

function showProduct(first: number, second: number): void {
    console.log("The product is: " + first * second);
}

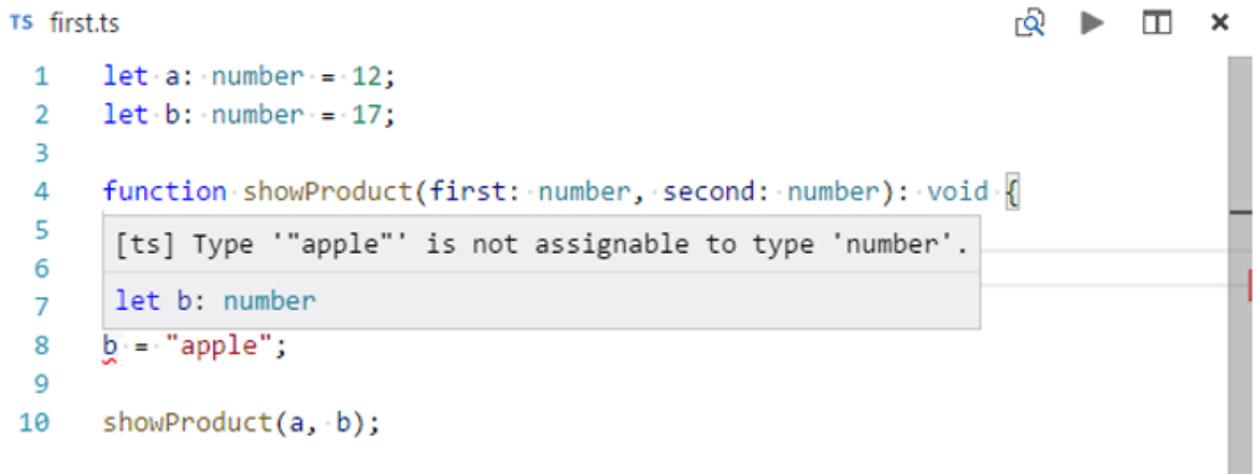
showProduct(a, b);
```

El código TypeScript de arriba compila el siguiente código JavaScript cuando la versión meta es establecida a ES6. Nota cómo todos el tipo de información que proporcionaste en TypeScript ya no está después de la compilación. En otras palabras, el código se compila a JavaScript que puede ser entendido por el navegador, pero puedes hacer el desarrollo en un ambiente mucho mejor que te puede ayudar a detectar muchos errores.

```
let a = 12;
let b = 17;
function showProduct(first, second) {
    console.log("The product is: " + first * second);
}
showProduct(a, b);
```

En el código de arriba, hemos especificado el tipo de variables `a` y `b` como números. Esto significa que si después intentas establecer el valor de `b` a una cadena como "apple", TypeScript te mostrará un error de que "apple" no es asignable al tipo `number`. Tu código aún se compilará a JavaScript, pero este mensaje de error te permitirá saber que cometiste un error y te ayudará a evitar un problema durante el tiempo de ejecución.

Aquí está una captura de pantalla que muestra el mensaje de error en Visual Studio Code:

A screenshot of the Visual Studio Code editor showing a TypeScript file named 'first.ts'. The code is as follows:

```
1 let a: number = 12;
2 let b: number = 17;
3
4 function showProduct(first: number, second: number): void {
5     [ts] Type '"apple"' is not assignable to type 'number'.
6     let b: number
7     b = "apple";
8
9
10 showProduct(a, b);
```

An error tooltip is displayed over line 5, containing the message: "[ts] Type '"apple"' is not assignable to type 'number'." The tooltip also shows the declaration of variable `b` on line 6: `let b: number`. The error is highlighted in red in the original image.

El ejemplo de arriba muestra cómo TypeScript sigue dándote pistas sobre posibles errores en el código. Este fue un ejemplo muy básico, pero cuando estás creando programas muy grandes, mensajes como este te ayudan bastante a escribir código robusto con menos probabilidad de error.

Ideas Finales

Este tutorial de inicio en la serie fue pensado para darte una breve vista general de las diferentes características de TypeScript y ayudarte a instalar el lenguaje junto con algunas sugerencias para IDEs y editores de texto que puedes usar para desarrollo. La siguiente sección cubrió diferentes maneras de compilar tu código TypeScript a JavaScript y te mostró cómo TypeScript puede ayudarte a evitar algunos errores comunes.

Espero que te haya gustado este tutorial. Si tienes alguna pregunta, por favor déjame saber en los comentarios. El siguiente tutorial de la serie discutirá diferentes tipos de variables disponibles en TypeScript.

TypeScript para Principiantes, Parte 2: Tipos Básicos de Datos

Después de haber leído el [tutorial introductorio de TypeScript](#), usted ahora debería poder escribir su propio código de TypeScript en un IDE que lo admita y luego compilarlo a JavaScript. En este tutorial, usted aprenderá sobre los diferentes tipos de tipos de datos disponibles en TypeScript.

JavaScript tiene siete tipos de datos diferentes: Null, Indefinido, Booleano, Número, Cadena, [Símbolo](#) (introducido en ES6) y Objeto. TypeScript define algunos tipos más, y todos ellos se tratarán en detalle en este tutorial.

El tipo de datos Null

Al igual que en JavaScript, el tipo de datos `null` en TypeScript solo puede tener un valor válido: `null`. Una variable null no puede contener otros tipos de datos como número y cadena de texto. Establecer una variable a null borrará su contenido si tuviese alguno.

Recuerde que cuando el indicador `strictNullChecks` se configura como `true` en `tsconfig.json`, solo el valor null se puede asignar a las variables con tipo null. Este indicador está desactivado por defecto, lo que significa que también puede asignar el valor null a variables con otros tipos como `number` o `void`.

```
// With strictNullChecks set to true
let a: null = null; // Ok
let b: undefined = null; // Error
let c: number = null; // Error
let d: void = null; // Error

// With strictNullChecks set to false
let a: null = null; // Ok
let b: undefined = null; // Ok
let c: number = null; // Ok
let d: void = null; // Ok
```

El Tipo de Datos Indefinido

Cualquier variable cuyo valor no haya especificado se establece en `undefined`. Sin embargo, usted también puede establecer explícitamente el tipo de una variable como indefinida, como se ve en el siguiente ejemplo.

Tenga en cuenta que una variable con `type` configurado como `undefined` solo puede tener `undefined` como su valor. Si la opción `strictNullChecks` está configurada como `false`, también podrá asignar `undefined` a variables de tipo numérico y cadenas de texto, etc.

```
// With strictNullChecks set to true
let a: undefined = undefined; // Ok
let b: undefined = null; // Error
let c: number = undefined; // Error
let d: void = undefined; // Ok

// With strictNullChecks set to false
let a: undefined = undefined; // Ok
let b: undefined = null; // Ok
let c: number = undefined; // Ok
let d: void = undefined; // Ok
```

El Tipo de Datos Void

El tipo de datos void se usa para indicar la falta de un `type` para una variable. Establecer variables para que tengan un tipo `void` puede no ser muy útil, pero usted puede establecer el tipo de retorno de las funciones que no devuelven nada a `void`. Cuando se usa con variables, el tipo `void` solo puede tener dos valores válidos: `null` y `undefined`.

```
// With strictNullChecks set to true
let a: void = undefined; // Ok
let b: void = null; // Error
let c: void = 3; // Error
let d: void = "apple"; // Error

// With strictNullChecks set to false
let a: void = undefined; // Ok
let b: void = null; // Ok
let c: void = 3; // Error
let d: void = "apple"; // Error
```

El Tipo de Datos Boolean

A diferencia de los tipos de datos `number` y `string`, `boolean` solo tiene dos valores válidos. Solo puedes establecer su valor en `true` o `false`. Estos valores se usan mucho en las estructuras de control donde una pieza de código se ejecuta si una condición es `true` y otra parte de código se ejecuta si una condición es `false`.

Aquí hay un ejemplo muy básico para declarar variables booleanas:

```
let a: boolean = true;
let b: boolean = false;
let c: boolean = 23; // Error
let d: boolean = "blue"; // Error
```

El Tipo de Datos Numérico

El tipo de datos `number` se usa para representar tanto enteros como valores de punto flotante en JavaScript y en TypeScript. Sin embargo, debe recordar que todos los números están representados internamente como valores de coma flotante. Los números también se pueden especificar como literales Hexadecimales, Octales o Binarios. Tenga en cuenta que las representaciones Octal y Binaria se introdujeron en ES6, y esto puede dar como resultado una salida de código JavaScript diferente en función de la versión a la que está apuntando.

Además hay tres valores simbólicos especiales adicionales que se encuentran bajo el tipo de `number`: `+Infinity`, `-Infinity` y `NaN`. Aquí hay algunos ejemplos del uso del tipo `number`.

```

// With strictNullChecks set to true
let a: number = undefined; // Error
let b: number = null; // Error
let c: number = 3;
let d: number = 0b111001; // Binary
let e: number = 0o436; // Octal
let f: number = 0xadf0d; // Hexadecimal
let g: number = "cat"; // Error

// With strictNullChecks set to false
let a: number = undefined; // Ok
let b: number = null; // Ok
let c: number = 3;
let d: number = 0b111001; // Binary
let e: number = 0o436; // Octal
let f: number = 0xadf0d; // Hexadecimal
let g: number = "cat"; // Error

```

Cuando la versión de destino se establece en ES6, el código anterior se compilará al siguiente JavaScript:

```

let a = undefined;
let b = null;
let c = 3;
let d = 0b111001;
let e = 0o436;
let f = 0xadf0d;
let g = "cat";

```

Usted debería tener en cuenta que las variables de JavaScript todavía se declaran con `let`, que se introdujo en ES6. Tampoco ve ningún mensaje de error relacionado con el `type` de variables diferentes porque el código JavaScript no tiene conocimiento de los tipos que usamos en el código TypeScript.

Si la versión de destino está configurada como ES5, el código de TypeScript que escribimos anteriormente se compilará en el siguiente JavaScript:

```

var a = undefined;
var b = null;
var c = 3;
var d = 57;
var e = 286;
var f = 0xadf0d;
var g = "cat";

```

Como puede ver, esta vez todas las ocurrencias de la palabra clave `let` han sido cambiadas a `var`. También tenga en cuenta que los números octales y binarios se han cambiado a sus formas decimales.

El Tipo de Datos String

El tipo de datos string se utiliza para almacenar información textual. Tanto JavaScript como TypeScript usan comillas dobles ("), así como comillas simples (') para rodear su información textual como una cadena. Una cadena puede contener cero o más caracteres entre comillas.

```
// With strictNullChecks set to true
let a: string = undefined; // Error
let b: string = null; // Error
let c: string = "";
let d: string = "y";
let e: string = "building";
let f: string = 3; // Error
let g: string = "3";

// With strictNullChecks set to false
let a: string = undefined; // Ok
let b: string = null; // Ok
let c: string = "";
let d: string = "y";
let e: string = "building";
let f: string = 3; // Error
let g: string = "3";
```

TypeScript también admite plantillas de cadenas de texto o plantillas de literales. Estas plantillas de literales le permiten incrustar expresiones en una cadena de texto. Las plantillas de literales están encerrados por el carácter de retroceso (```) en lugar de comillas dobles y comillas simples que encierran cadenas de texto regulares. Fueron presentados en ES6. Esto significa que usted obtendrá diferentes resultados de JavaScript en función de la versión a la que esté apuntando. Aquí hay un ejemplo de uso de plantillas de literales en TypeScript:

```
let e: string = "building";
let f: number = 300;

let sentence: string = `The ${e} in front of my office is ${f} feet tall.`;
```

En la compilación, obtendrá el siguiente JavaScript:

```
1 // Output in ES5
2 var e = "building";
3 var f = 300;
4 var sentence = "The " + e + " in front of my office is " + f + " feet
5 tall.";
6
7 // Output in ES6
8 let e = "building";
9 let f = 300;
10 let sentence = `The ${e} in front of my office is ${f} feet tall.`;
```

Como puede ver, la plantilla de literal se cambió a una cadena de texto regular en ES5. Este ejemplo muestra cómo TypeScript hace posible que use todas las funciones de JavaScript más recientes sin preocuparse por la compatibilidad.

Los tipos de datos Array y Tuple

Puede definir tipos de array de dos maneras diferentes en JavaScript. En el primer método, usted especifica el tipo de elementos de array seguidos por `[]` lo cual que denota un array de ese tipo. Otro método es utilizar el tipo de array genérico `Array<elemType>`. El siguiente ejemplo muestra cómo crear arrays con ambos métodos.

Especificar `null` o `undefined` como uno de los elementos producirá errores cuando el indicador `strictNullChecks` sea `true`.

```

// With strictNullChecks set to false
let a: number[] = [1, 12, 93, 5];
let b: string[] = ["a", "apricot", "mango"];
let c: number[] = [1, "apple", "potato"]; // Error

let d: Array<number> = [null, undefined, 10, 15];
let e: Array<string> = ["pie", null, ""];

// With strictNullChecks set to true
let a: number[] = [1, 12, 93, 5];
let b: string[] = ["a", "apricot", "mango"];
let c: number[] = [1, "apple", "potato"]; // Error

let d: Array<number> = [null, undefined, 10, 15]; // Error
let e: Array<string> = ["pie", null, ""]; // Error

```

El tipo de datos de tupla le permite crear un array donde el tipo de un número fijo de elementos se conoce de antemano. El tipo del resto de los elementos solo puede ser uno de los tipos que ya ha especificado para la tupla. Aquí hay un ejemplo que lo hará más claro:

```

let a: [number, string] = [11, "monday"];
let b: [number, string] = ["monday", 11]; // Error
let c: [number, string] = ["a", "monkey"]; // Error
let d: [number, string] = [105, "owl", 129, 45, "cat"];
let e: [number, string] = [13, "bat", "spiderman", 2];

e[13] = "elephant";
e[15] = false; // Error

```

Para todas las tuplas en nuestro ejemplo, hemos establecido el `type` del primer elemento como `number` y el `type` del segundo elemento como `string`. Como solo hemos especificado un `type` para los dos primeros elementos, el resto de ellos puede ser una cadena de texto o un número. La creación de tuplas `b` y `c` da como resultado un error porque tratamos de usar una cadena de texto como un valor para el primer elemento cuando mencionamos que el primer elemento sería un número.

Del mismo modo, no podemos establecer el valor de un elemento de tupla como `false` después de especificar que solo contendrá cadenas de texto y números. Es por eso que la última línea resulta en un error.

El Tipo de Datos Enum

El tipo de datos `enum` está presente en muchos lenguajes de programación como C y Java. Ha faltado en JavaScript, pero TypeScript le permite crear y trabajar con enums. Si no sabe qué son `enums`, le permiten crear una colección de valores relacionados con nombres memorables.

```
enum Animals {cat, lion, dog, cow, monkey}
let c: Animals = Animals.cat;
```

```
console.log(Animals[3]); // cow
console.log(Animals.monkey); // 4
```

Por defecto, la numeración de enums comienza en 0, pero también puede establecer un valor diferente para el primero u otros miembros de forma manual. Esto cambiará el valor de todos los miembros que los siguen aumentando su valor en 1. También puede establecer todos los valores manualmente en un `enum`.

```
1 enum Animals {cat = 1, lion, dog = 11, cow,
2 monkey}
3 let c: Animals = Animals.cat;
4
5 console.log(Animals[3]); // undefined
6 console.log(Animals.monkey); // 13
```

A diferencia del ejemplo anterior, el valor de `Animals[3]` es `undefined` esta vez. Esto se debe a que el valor 3 se le habría asignado a `dog`, pero establecemos explícitamente su valor en 11. El valor para `cow` permanece en 12 y no en 3 porque se supone que el valor es uno mayor que el valor del último miembro.

Los tipos Any y Never

Digamos que usted está escribiendo un programa donde el valor de una variable lo determinan los usuarios o el código escrito en una biblioteca de terceros. En este caso, no podrá establecer el tipo de esa variable correctamente. La variable podría ser de cualquier tipo, como una cadena de texto, número o booleano. Este problema se puede resolver

usando el tipo `any`. Esto también es útil cuando está creando arrays con elementos de tipos mixtos.

```
let a: any = "apple";
let b: any = 14;
let c: any = false;
```

En el código anterior, pudimos asignar un número a `b` y cambiar su valor a una cadena de texto sin obtener ningún error porque el tipo `any` puede aceptar todos los tipos de valores.

El tipo `never` se usa para representar valores que nunca se supone que ocurran. Por ejemplo, puede asignar `never` como el tipo de devolución de una función que nunca regresa. Esto puede suceder cuando una función siempre arroja un error o cuando está atrapada en un ciclo infinito.

```
let a: never; // Ok
let b: never = false; // Error
let c: never = null; // Error
let d: never = "monday"; // Error

function stuck(): never {
  while (true) {
  }
}
```

Pensamientos Finales

Este tutorial le presentó todos los tipos que están disponibles en TypeScript. Aprendimos cómo la asignación de un tipo diferente de valor a una variable mostrará errores en TypeScript. Esta comprobación puede ayudarlo a evitar muchos errores al escribir programas grandes. También aprendimos cómo orientarnos a diferentes versiones de JavaScript.

Si está buscando recursos adicionales para estudiar o usar en su trabajo, consulte lo que tenemos disponible en el [mercado de Envato](#).

En el siguiente tutorial, aprenderá sobre las interfaces en TypeScript. Si tiene alguna pregunta relacionada con este tutorial, hágamelo saber en los comentarios.

TypeScript Para Principiantes,

Parte 3: Interfaces

Comenzamos esta serie con un tutorial de introducción que te introducen a las diferentes características de mecanografiado. También le enseñó cómo instalar TypeScript y sugirió algunas IDEs que puede utilizar para escribir y compilar su propio código de TypeScript.

En el segundo tutorial, cubrimos [diferentes tipos de datos disponibles en TypeScript](#) y cómo usarlos pueden ayudarle a evitar muchos errores. Asignar a un tipo de datos como una `string` a una variable particular dice mecanografiado que desea asignar una cadena a ella. Basándose en esta información, mecanografiado puede señalarlo más tarde cuando intenta realizar una operación que no debe hacerse en las cuerdas.

En este tutorial, usted aprenderá sobre las [interfaces en TypeScript](#). Con interfaces, puede dar un paso más y definir la estructura o tipo de objetos más complejos en el código. Al igual que tipos de variables simples, estos objetos también tendrá que seguir un conjunto de reglas creadas por usted. Esto puede ayudarle a escribir código con más confianza, con menos posibilidades de error.

Creando Nuestra Primera Interfaz

Digamos que tienes un objeto de lago en el código y usarlo para almacenar información sobre algunos de los [lagos más grande por área](#) en el mundo. Este objeto de lago tiene propiedades como el nombre del lago, su área, longitud, profundidad y los países en que existe ese lago.

Los nombres de los lagos se guardará como una cadena. Las longitudes de estos lagos será en kilómetros y serán las áreas en kilómetros cuadrados, pero ambas de estas propiedades serán almacenados como números. Las profundidades de los lagos será en metros, y esto también podría ser un flotador.

Puesto que todos estos lagos son muy grandes, sus costas generalmente no se limitan a un solo país. Se utilizará una matriz de cadenas para almacenar los nombres de todos los países en la orilla de un lago particular. Un valor Boolean puede utilizarse para especificar si el lake es agua salada o agua dulce. El siguiente fragmento de código crea una interfaz para el objeto de nuestro lago.

```
interface Lakes {  
    name: string,  
    area: number,  
    length: number,  
    depth: number,  
    isFreshwater: boolean,  
    countries: string[]  
}
```

La interfaz de `Lakes` contiene el tipo de cada propiedad que vamos a utilizar al crear los objetos de nuestro lago. Si ahora intenta asignar a diferentes tipos de valores a cualquiera de estas propiedades, se producirá un error. Aquí es un ejemplo que almacena información acerca de nuestro primer lago.

```
let firstLake: Lakes = {  
    name: 'Caspian Sea',  
    length: 1199,  
    depth: 1025,  
    area: 371000,  
    isFreshwater: false,  
    countries: ['Kazakhstan', 'Russia', 'Turkmenistan', 'Azerbaijan', 'Iran']  
}
```

Como se puede ver, no importa el orden en que asigna un valor a estas propiedades. Sin embargo, no puede omitir un valor. Usted tendrá que asignar un valor a cada propiedad con el fin de evitar errores al compilar el código.

Esta manera, mecanografiado se asegura de que usted no faltó ninguno de los valores deseados por error. Aquí hay un ejemplo donde olvidamos para asignar el valor de la propiedad de `depth` de un lago.

```

let secondLake: Lakes = {
  name: 'Superior',
  length: 616,
  area: 82100,
  isFreshwater: true,
  countries: ['Canada', 'United States']
}

```

La captura de pantalla siguiente muestra el mensaje de error en código de Visual Studio después de que olvidé de especificar la `depth`. Como se puede ver, el error claramente indica que estamos perdiendo la propiedad `depth` para nuestro objeto de lago.

```

10 |
11 | let
12 | [ts]
13 | Type '{ name: string; length: number; area: number; isFreshwater:
14 | true; countries: string[]; }' is not assignable to type 'Lakes'.
15 | Property 'depth' is missing in type '{ name: string; length:
16 | number; area: number; isFreshwater: true; countries: string[];
17 | }'.
18 |
19 | let secondLake: Lakes
20 | let secondLake: Lakes = {
21 |   name: 'Superior',
22 |   length: 616,
23 |   area: 82100,
24 |   isFreshwater: true,
25 |   countries: ['Canada', 'United States']
26 | }

```

Haciendo la Propiedades de la Interfaz Opcionales

A veces, puede que necesite una propiedad sólo para algunos objetos específicos. Por ejemplo, supongamos que desea agregar una propiedad para especificar los meses en los que un lago se congela. Si agrega la propiedad directamente a la interfaz, como lo hemos hecho hasta ahora, obtendrá un error para otros lagos que no congelan y por lo tanto no tener ninguna propiedad `frozen`. Asimismo, si agrega esa propiedad a los lagos que se congelan pero no en la declaración de interfaz, se da un error.

En tales casos, puede añadir un signo de interrogación (?) después del nombre de una propiedad para establecer como opcional en la declaración de interfaz. Esta manera, no conseguirá un error por falta de propiedades o propiedades desconocidas. En el ejemplo siguiente, se debe dejar claro.

```
interface Lakes {
  name: string,
  area: number,
  length: number,
  depth: number,
  isFreshwater: boolean,
  countries: string[],
  frozen?: string[]
}

let secondLake: Lakes = {
  name: 'Superior',
  depth: 406.3,
  length: 616,
  area: 82100,
  isFreshwater: true,
  countries: ['Canada', 'United States']
}

let thirdLake: Lakes = {
  name: 'Baikal',
  depth: 1637,
  length: 636,
  area: 31500,
  isFreshwater: true,
  countries: ['Russia'],
  frozen: ['January', 'February', 'March', 'April', 'May']
}
```

Utilizando el Índice de Firmas

Propiedades opcionales son útiles cuando un buen número de los objetos van a utilizarlos. Sin embargo, ¿qué pasa si cada lago también tenía su propio conjunto de propiedades como las actividades económicas, la población de diferentes tipos de flora y fauna floreciente en ese lago, o los poblados alrededor del lago? Agregar muchos diferentes propiedades dentro de la declaración de la propia interfaz y hacerlos opcional no son ideal.

Como solución, TypeScript permite añadir propiedades extras a objetos específicos con la ayuda de las firmas del índice. Agregar una firma de índice a la declaración de interfaz le

permite especificar cualquier número de propiedades de diferentes objetos que están creando. Necesita realizar los siguientes cambios en la interfaz.

En este ejemplo, he utilizado una firma de índice para agregar información acerca de diferentes asentamientos alrededor de los lagos. Ya que cada lago tendrá sus propios asentamientos, no utilizando propiedades opcionales habría sido una buena idea.

```
interface Lakes {
  name: string,
  area: number,
  length: number,
  depth: number,
  isFreshwater: boolean,
  countries: string[],
  frozen?: string[],
  [extraProp: string]: any
}

let fourthLake: Lakes = {
  name: 'Tanganyika',
  depth: 1470,
  length: 676,
  area: 32600,
  isFreshwater: true,
  countries: ['Burundi', 'Tanzania', 'Zambia', 'Congo'],
  kigoma: 'Tanzania',
  kalemie: 'Congo',
  bujumbura: 'Burundi'
}
```

Como otro ejemplo, supongamos que está creando un juego con diferentes tipos de enemigos. Todos estos enemigos tienen algunas características comunes como su tamaño y la salud. Estas propiedades se pueden incluir en la declaración de interfaz directamente. Si cada categoría de estos enemigos tiene un conjunto único de las armas, esas armas pueden ser incluidas con la ayuda de una firma de índice.

Propiedades de Sólo Lectura

Cuando se trabaja con objetos diferentes, puede que necesite trabajar con propiedades que sólo pueden modificarse cuando creamos primero el objeto. Puede marcar estas propiedades

como `readonly` en la declaración de interfaz. Esto es similar al uso de la palabra clave `const`, sino `const` se supone para ser utilizado con variables, mientras que `readonly` se entiende por propiedades.

Escrito permite realizar arreglos de discos de sólo lectura mediante `ReadOnlyArray<T>`. Crear una matriz de sólo lectura resultará en la eliminación de todos los métodos mutando desde eso. Esto se hace para asegurarse de que no puede cambiar el valor de los elementos individuales más adelante. Aquí es un ejemplo del uso de matrices y propiedades de sólo lectura en las declaraciones de interfaz.

```
interface Enemy {
    readonly size: number,
    health: number,
    range: number,
    readonly damage: number
}

let tank: Enemy = {
    size: 50,
    health: 100,
    range: 60,
    damage: 12
}

// This is Okay
tank.health = 95;

// Error because 'damage' is read-only.
tank.damage = 10;
```

Funciones e Interfaces

También puede utilizar interfaces para describir un tipo de función. Esto requiere que la función de una firma llamada con su lista de parámetros y tipo de valor devuelto. También debe proporcionar un nombre y un tipo para cada uno de los parámetros. Aquí está un ejemplo:

```
interface EnemyHit {  
    (name: Enemy, damageDone: number): number;  
}  
  
let tankHit: EnemyHit = function(tankName: Enemy, damageDone: number) {  
    tankName.health -= damageDone;  
    return tankName.health;  
}
```

En el código anterior, hemos declarado una interfaz de función y utiliza para definir una función que resta el daño infligido a un tanque de su salud. Como se puede ver, no tienes que utilizar el mismo nombre para los parámetros en la declaración de la interfaz y la definición del código para trabajar.

Reflexiones Finales

Este tutorial le presenta a interfaces y cómo usted puede utilizar para asegurarse de que está escribiendo código más robusto. Ahora debe ser capaz de crear sus propias interfaces con características opcionales y de sólo lectura.

También aprendió a utilizar índice firmas para agregar una variedad de otras propiedades a un objeto que no están incluidos en la declaración de interfaz. Este tutorial fue significado para comenzar con interfaces por escrito, y usted puede leer más sobre este tema en [la documentación oficial](#).

En el siguiente tutorial, usted aprenderá acerca de las clases en TypeScript. Si usted tiene alguna pregunta relacionada con las interfaces, hágamelo saber en los comentarios.

TypeScript Para Principiantes, Parte 4: Clases

Hemos recorrido un largo camino en el aprendizaje de TypeScript desde el inicio de esta serie. El primer tutorial le proporcionó una breve [introducción de TypeScript](#) y sugirió algunos IDE que puede usar para escribir TypeScript. El segundo tutorial se centró en [los tipos de datos](#) , y el tercer tutorial discutió los conceptos básicos de las [interfaces en TypeScript](#) .

Como ya sabrá, JavaScript ha agregado recientemente soporte nativo para clases y programación orientada a objetos. Sin embargo, TypeScript ha permitido a los desarrolladores utilizar clases en su código durante mucho tiempo. Este código luego se compila en JavaScript que funcionará en todos los principales navegadores. En este tutorial, aprenderá sobre las clases en TypeScript. Son similares a sus homólogos de ES6 pero son más estrictos.

Creando tu primera clase

Empecemos con lo básico. Las clases son una parte fundamental de la programación orientada a objetos. Usas clases para representar cualquier entidad que tenga algunas propiedades y funciones que puedan actuar sobre propiedades determinadas. TypeScript le brinda control total sobre las propiedades y funciones que son accesibles dentro y fuera de su propia clase contenedora. Aquí hay un ejemplo muy básico de crear una clase de `Person` .

```

class Person {
  name: string;
  constructor(theName: string) {
    this.name = theName;
  }
  introduceSelf() {
    console.log("Hi, I am " + this.name + "!");
  }
}

let personA = new Person("Sally");

personA.introduceSelf();

```

El código anterior crea una clase muy simple llamada `Person`. Esta clase tiene una propiedad llamada `name` y una función llamada `introduceSelf`. La clase también tiene un constructor, que también es básicamente una función. Sin embargo, los constructores son especiales porque se llaman cada vez que creamos una nueva instancia de nuestra clase. También puede pasar parámetros a los constructores para inicializar diferentes propiedades. En nuestro caso, estamos usando el constructor para inicializar el nombre de la persona que estamos creando usando la clase `Person`. La función `introduceSelf` es un método de la clase `Person`, y la estamos usando aquí para imprimir el nombre de la persona en la consola. Todas estas propiedades, métodos y el constructor de una clase se denominan colectivamente miembros de la clase.

Debe tener en cuenta que la clase `Person` no crea automáticamente una persona por sí misma. Actúa más como un plano con toda la información sobre los atributos que una persona debería haber creado una vez. Con eso en mente, creamos una nueva persona y la llamamos Sally. Al llamar al método `introduceSelf` en esta persona, se imprimirá la línea "Hola, soy Sally". a la consola.

Modificadores privados y públicos

En la sección anterior, creamos una persona llamada Sally. En este momento, es posible cambiar el nombre de la persona de Sally a Mindy en cualquier lugar de nuestro código, como se muestra en el siguiente ejemplo.

```

class Person {
  name: string;
  constructor(theName: string) {
    this.name = theName;
  }
  introduceSelf() {
    console.log("Hi, I am " + this.name + "!");
  }
}

let personA = new Person("Sally");

// Prints "Hi, I am Sally!"
personA.introduceSelf();

personA.name = "Mindy";

// Prints "Hi, I am Mindy!"
personA.introduceSelf();

```

Es posible que haya notado que pudimos usar tanto la propiedad de `name` como el método de `introduceSelf` fuera de la clase contenedora. Esto se debe a que todos los miembros de una clase en TypeScript son `public` por defecto. También puede especificar explícitamente que una propiedad o método es público agregando la palabra clave `public` before. A veces, no desea que una propiedad o método sea accesible fuera de su clase contenedora. Esto se puede lograr haciendo que esos miembros sean privados usando la palabra clave `private`. En el código anterior, podríamos hacer que la propiedad del nombre sea `private` e impedir que se modifique fuera de la clase contenedora. Después de este cambio, TypeScript le mostrará un error que dice que la propiedad del `name` es `private` y que solo puede acceder a ella dentro de la clase `Person`. La captura de pantalla a continuación muestra el error en Visual Studio Code.

```
s
map
ig.json
1 class Person {
2   ... private name: string;
3   ... constructor(theName: string) {
4     ... this.name = theName;
5   }
6   ... introduceSelf() {
7     ... console.log("Hi, I am " + this.name + "!");
8   }
9 }
10
11 let pers
12 (property) Person.name: string
13 personA.name = "Mindy";
14 personA.introduceSelf();
15
```

Herencia en TypeScript

La herencia le permite crear clases más complejas a partir de una clase base. Por ejemplo, podemos usar la clase `Person` de la sección anterior como base para crear una clase `Friend` que tendrá todos los miembros de la `Person` y agregar algunos miembros propios. Del mismo modo, también puede agregar una clase de `Family` o `Teacher`.

Todos heredarán los métodos y las propiedades de la `Person` y agregarán algunos métodos y propiedades propios para diferenciarlos. El siguiente ejemplo debería hacerlo más claro. También agregué el código para la clase `Person` aquí para que pueda comparar fácilmente el código tanto de la clase base como de la clase derivada.

```

class Person {
  private name: string;
  constructor(theName: string) {
    this.name = theName;
  }
  introduceSelf() {
    console.log("Hi, I am " + this.name + "!");
  }
}

class Friend extends Person {
  yearsKnown: number;
  constructor(name: string, yearsKnown: number) {
    super(name);
    this.yearsKnown = yearsKnown;
  }
  timeKnown() {
    console.log("We have been friends for " + this.yearsKnown + " years.")
  }
}

let friendA = new Friend("Jacob", 6);

// Prints: Hi, I am Jacob!
friendA.introduceSelf();

// Prints: We have been friends for 6 years.
friendA.timeKnown();

```

Como puede ver, debe usar la palabra clave `extend` para que la clase `Friend` herede todos los miembros de la clase `Person`. Es importante recordar que el constructor de una clase derivada siempre debe invocar al constructor de la clase base con una llamada a `super()`.

Es posible que haya notado que el constructor de `Friend` no necesitaba tener la misma cantidad de parámetros que la clase base. Sin embargo, el primer parámetro de nombre se pasó a `super()` para invocar al constructor del elemento principal, que también aceptó un parámetro. No tuvimos que redefinir la función de `introduceSelf` Sí Mismo dentro de la clase `Friend` porque se heredó de la clase `Person`.

Usando el Modificador Protegido

Hasta este punto, solo hemos hecho que los miembros de una clase sean `private` o `public`. Al hacerlos públicos, nos permite acceder a ellos desde cualquier lugar, haciendo que los miembros los limiten a su propia clase. En ocasiones, es posible que desee que los miembros de una clase base estén accesibles dentro de todas las clases derivadas.

Puede usar el modificador `protected` en tales casos para limitar el acceso de un miembro solo a las clases derivadas. También puede usar la palabra clave `protected` con el constructor de una clase base. Esto evitará que alguien cree una instancia de esa clase. Sin embargo, aún podrá extender clases basadas en esta clase base.

```
class Person {
    private name: string;
    protected age: number;
    protected constructor(theName: string, theAge: number) {
        this.name = theName;
        this.age = theAge;
    }
    introduceSelf() {
        console.log("Hi, I am " + this.name + "!");
    }
}

class Friend extends Person {
    yearsKnown: number;
    constructor(name: string, age: number, yearsKnown: number) {
        super(name, age);
        this.yearsKnown = yearsKnown;
    }
    timeKnown() {
        console.log("We have been friends for " + this.yearsKnown + " years.")
    }
    friendSince() {
        let firstAge = this.age - this.yearsKnown;
        console.log(`We have been friends since I was ${firstAge} years old.`)
    }
}

let friendA = new Friend("William", 19, 8);

// Prints: We have been friends since I was 11 years old.
friendA.friendSince();
```

En el código anterior, puede ver que hemos `protected` propiedad de `age` . Esto evita el uso de la `age` fuera de cualquier clase derivada de `Person` . También usamos la palabra clave `protected` para el constructor de la clase `Person` . Declarar el constructor como `protected` significa que ya no podremos instanciar directamente la clase `Person` . La siguiente captura de pantalla muestra un error que aparece al intentar crear una instancia de una clase con el constructor `protected` .

```
1 class Person {
2   ... private name: string;
3   ... protected age: number;
4   ... protected constructor(theName: string, theAge: number) {
5     ... this.name = theName;
6     ... this.age = theAge;
7   ... }
8   ... introduceSelf() {
9     ... console
10  ... }
11 }
12
13 let personA = new Person("Amanda", 22);
14
```

[ts] Constructor of class 'Person' is protected and only accessible within the class declaration.

constructor Person(theName: string, theAge: number): Person

Pensamientos finales

En este tutorial, he tratado de cubrir los conceptos básicos de las clases en TypeScript. Comenzamos el tutorial creando una clase de `Person` muy básica que imprimió el nombre de la persona en la consola. Después de eso, aprendí sobre la palabra clave `private` , que puede usarse para evitar el acceso de los miembros de una clase en cualquier punto arbitrario del programa.

Finalmente, aprendí cómo extender diferentes clases en su código usando una clase base con herencia. Hay mucho más que puedes aprender sobre las clases en la [documentación oficial](#) .

Si tiene alguna pregunta relacionada con este tutorial, hágame saber en los comentarios.

TypeScript para Principiantes, Parte 5: Genéricos

El segundo tutorial en nuestra serie [TypeScript para Principiantes](#) se centró en [tipos de datos básico disponibles en TypeScript](#). La revisión de tipo en TypeScript nos permite asegurar que las variables en nuestro código solo pueden tener tipos específicos de valores asignados a estas. De esta manera podemos evitar muchos errores mientras escribimos código debido a que el IDE será capaz de decirnos cuando estamos realizando una operación sobre un tipo que no soporta. Esto hace a la revisión de tipo una de las mejores características de TypeScript.

En este tutorial, nos centraremos en otra importante característica de este lenguaje--- genéricos. Con genéricos, TypeScript te permite escribir código que puede actuar sobre una variedad de datos en vez de ser limitado a solo uno. Aprenderás sobre la necesidad de genéricos a detalle y cómo es mejor que solo usar el tipo de dato `any` disponible en TypeScript.

La Necesidad de Genéricos

Si no estás familiarizado con genéricos, podrías estar preguntándote por qué los necesitamos en absoluto. En esta sección, responderé esta pregunta por ti. Comencemos escribiendo una función que devolverá un elemento aleatorio de un arreglo de números.

```
function randomIntElem(theArray: number[]): number {
    let randomIndex = Math.floor(Math.random()*theArray.length);
    return theArray[randomIndex];
}

let positions: number[] = [103, 458, 472, 458];
let randomPosition: number = randomIntElem(positions);
```

La función `randomElem` que acabamos de definir toma un arreglo de números como su único parámetro. El tipo de retorno de la función también ha sido especificado como un número.

Estamos usando la función `Math.random()` para devolver un número de punto flotante aleatorio entre 0 y 1. Multiplicarlo con la longitud del arreglo y llamando a `Math.floor()` sobre el resultado nos da un índice aleatorio. Una vez que tenemos el índice aleatorio, devolvemos un elemento en ese índice específico.

Algún tiempo después, digamos que necesitas un elemento aleatorio de cadena de un arreglo de cadenas. En este punto, podrías decidir crear otra función que ataque específicamente cadenas.

```
function randomStrElem(theArray: string[]): string {
  let randomIndex = Math.floor(Math.random()*theArray.length);
  return theArray[randomIndex];
}

let colors: string[] = ['violet', 'indigo', 'blue', 'green'];
let randomColor: string = randomStrElem(colors);
```

¿Qué pasa si necesitas seleccionar un elemento aleatorio de un arreglo de una interfaz que tu definiste? Crear una nueva función cada vez que quieras obtener un elemento aleatorio de un arreglo de diferentes tipos de objetos no es viable.

Una solución para este problema es establecer el parámetro de tipo de arreglo siendo pasado a las funciones como `any[]`. De este modo, puedes solo escribir la función una vez, y funcionará con un arreglo de todos los tipos.

```
function randomElem(theArray: any[]): any {
  let randomIndex = Math.floor(Math.random()*theArray.length);
  return theArray[randomIndex];
}

let positions = [103, 458, 472, 458];
let randomPosition = randomElem(positions);

let colors = ['violet', 'indigo', 'blue', 'green'];
let randomColor = randomElem(colors);
```

Como puedes ver, podemos usar la función de arriba para obtener posiciones aleatorias así como colores aleatorios. Un problema mayor con esta solución es que perderás la información sobre el tipo de valor que está siendo devuelto.

Anteriormente, estábamos seguros de que `randomPosition` sería un número y `randomColor` sería una cadena. Esto nos ayudó a usar estos valores de manera acorde. Ahora, todo lo que sabemos es que el elemento devuelto podría ser de cualquier tipo. En el código de arriba, podríamos especificar el tipo de `randomColor` para ser un `number` y aún así no obtener ningún error.

```
// This code will compile without an error.
let colors: string[] = ['violet', 'indigo', 'blue', 'green'];
let randomColor: number = randomElem(colors);
```

Una mejor solución para evitar duplicación de código mientras aún conservamos la información de tipo es usar genéricos. Aquí está una función genérica que devuelve elementos aleatorios de un arreglo.

```
function randomElem<T>(theArray: T[]): T {
  let randomIndex = Math.floor(Math.random()*theArray.length);
  return theArray[randomIndex];
}

let colors: string[] = ['violet', 'indigo', 'blue', 'green'];
let randomColor: string = randomElem(colors);
```

Ahora, obtendré un error si intento cambiar el tipo de `randomColor` de `string` a `number`. Esto prueba que usar genéricos es mucho más seguro que usar el tipo `any` en tales situaciones.

```
33
34 function randomElem<T>(theArray: T[]): T {
35   ... let randomIndex = Math.floor(Math.random()*theArray.length);
36   ... return theArray[randomIndex];
37 }
38
39 let colors: string[] = ['violet', 'indigo', 'blue', 'green'];
40
41 [ts] Type 'string' is not assignable to type 'number'.
42   let randomColor: number
43 let randomColor: number = randomElem(colors);
44 console.log(randomColor);
```

Usando Genéricos Podría Parecer Muy Limitante

La sección previa discutió cómo puedes usar genéricos en vez del tipo `any` para poder escribir una sola función y evitar sacrificar beneficios de revisión de tipo. Un problema con la función genérica que hemos escrito en la sección previa es que TypeScript no nos dejará realizar muchas operaciones sobre variables pasadas a esta.

Esto es porque TypeScript no puede hacer ninguna suposición sobre el tipo de variable que será pasada a nuestra función genérica de antemano. Como resultado, una función genérica solo puede usar aquellas operaciones que son aplicables en todos los tipos de datos. El siguiente ejemplo debería hacer este concepto más claro.

```
function removeChar(theString: string, theChar: string): string {
    let theRegex = new RegExp(theChar, "gi");
    return theString.replace(theRegex, '');
}
```

La función de arriba removerá todas las ocurrencias del caracter especificado desde la cadena dada. Podrías querer crear una versión genérica de esta función para que también puedas remover dígitos específicos de un número dado así como caracteres de una cadena. Aquí está la función genérica correspondiente.

```
function removeIt<T>(theInput: T, theIt: string): T {
    let theRegex = new RegExp(theIt, "gi");
    return theInput.replace(theRegex, '');
}
```

La función `removeChar` no te mostró un error. Sin embargo, si usas `replace` dentro de `removeIt`, TypeScript te dirá que `replace` no existe para el tipo 'T'. Esto es debido a que TypeScript ya no puede asumir que `theInput` va a ser una cadena.

Esa restricción sobre usar diferentes métodos en una función genérica podría llevarte a pensar que el concepto de genéricos no va a ser de mucha utilidad después de todo. Eso no es realmente mucho cuando puedes hacerlo con un puñado de métodos que deben ser aplicables en todos los tipos de datos para que los uses dentro de una función genérica.

Una cosa importante que deberías recordar en este punto es que no creas generalmente funciones que serán usadas con todos los tipos de datos. Es mucho más común crear una

función que será usada con un conjunto específico o rango de tipos de datos. Esta restricción sobre tipos de datos hace a las funciones genéricas mucho más útiles.

Crea Funciones Genéricas Usando Restricciones

La función genérica `removeIt` de la sección anterior mostró un error porque el método `replace` dentro de esta debe ser usado con cadenas, mientras que los parámetros pasados a este podría tener cualquier tipo de dato.

Puedes usar la palabra clave `extends` para restringir los tipos de datos que son pasados a una función genérica en TypeScript. Sin embargo, `extends` está limitado a solo interfaces y clases. Esto significa que la mayoría de funciones genéricas que creas tendrán parámetros que extienden una interfaz o clase base.

Aquí está la función genérica que imprime el nombre de la gente, familiares o celebridades pasados a esta.

```
interface People {
  name: string
}

interface Family {
  name: string,
  age: number,
  relation: string
}

interface Celebrity extends People {
  profession: string
}

function printName<T extends People>(theInput: T): void {
  console.log(`My name is ${theInput.name}`);
}

let serena: Celebrity = {
  name: 'Serena Williams',
  profession: 'Tennis Player'
}

printName(serena);
```

En el ejemplo de abajo, hemos definido tres interfaces, y cada una de ellas tiene una propiedad `name`. La función genérica `printName` que creamos aceptará cualquier objeto que extienda a `People`. En otras palabras, puedes pasar ya sea un objeto de familiar o celebridad a esta función, e imprimirá su nombre sin ninguna queja. Puedes definir muchas más interfaces, y mientras tengan una propiedad `name`, podrás usar la función `printName` sin ningún problema.

Este fue un ejemplo muy básico, pero puedes crear más funciones genéricas útiles una vez que estés más cómodo con todo el proceso. Por ejemplo, puedes crear una función genérica que calcule el valor total de diferentes artículos vendidos en un mes dado siempre y cuando cada artículo tenga una propiedad `price` para almacenar el precio y una propiedad `sold` que almacene el número de artículos vendidos. Usando genéricos, podrás usar la misma función siempre que los elementos extiendan la misma interfaz o clase.

Ideas Finales

En este tutorial, he intentado cubrir los básicos de genéricos en TypeScript de manera amigable con principiantes. Comenzamos el artículo discutiendo la necesidad de genéricos. Después de eso, aprendimos sobre la manera correcta de usar genéricos para evitar duplicación de código sin sacrificar la disponibilidad de revisión de tipo. Una vez que entiendas los básicos discutidos aquí, puedes leer más sobre [genéricos en la documentación oficial](#).

Si tienes preguntas relacionadas a este tutorial, estaré feliz de responderlas en los comentarios.