

# TYPESCRIPT

DESARROLLO DE APLICACIONES WEB II

# ¿QUÉ ES TYPESCRIPT?

- Elegir TypeScript como lenguaje de programación es una excelente decisión ya que nos permite construir software mas robusto. Posiblemente ya hayas escuchado que TypeScript es una variante de JavaScript con características que permiten trabajar de mejor forma con este lenguaje. Por ello es importante tener en cuenta que ambos JavaScript (JS) como TypeScript (TS) tienen una relación estrecha como lenguajes de programación, y que es importante entender JS para poder entender TS.

# ¿QUÉ ES JAVASCRIPT?

- JavaScript también conocido como ECMAScript inicio su vida como un lenguaje de programación enfocado en los navegadores. Cuando fue creado se utilizó para crear bloques de código que permitieran dar cierto nivel de interactividad a las páginas web. Sin embargo en sus inicios los navegadores eran software bastante lento por lo que agregar mucha interactividad no era recomendable. Con el tiempo tanto los equipos de cómputo, como los navegadores y los intérpretes de lenguajes como JavaScript han evolucionado, y hoy podemos crear desde páginas web sencillas hasta complejas aplicaciones en JavaScript.

- Además de evolucionar en el navegador, JavaScript también lo ha echo en otros rubros, hoy es posible ejecutar JavaScript dentro de un sistema como nodejs y crear código destinado a la parte del backend. El poder ejecutar JavaScript en cualquier entorno hace del lenguaje una opción muy atractiva para crear aplicaciones multiplataforma al punto que hoy en día muchos programadores eligen JavaScript como su opción predilecta para todo su stack de programación.
- JavaScript ha evolucionado de ser un lenguaje del navegador a un lenguaje de multipropósito. Como cualquier lenguaje de programación tiene sus pros y sus contras.

# EL TIPADO

- La detección de errores dentro del código que se realiza sin ejecutarlo se conoce como static checking. El determinar que es y que no es un error mediante la determinación del tipo de valores que se están utilizando se conoce como static type checking.
- TypeScript verifica un programa antes de que este se ejecute, y lo hace verificando los tipos de valores (static type checking). Para el siguiente ejemplo se ha introducido un error al escribir la propiedad de forma intencional.



ts Ejemplo1.ts > ...

```
1  const rectangulo = { altura: 10, anchura: 15 };  
2  const area = rectangulo.altura * rectangulo.anchuRa;
```

Property 'anchuRa' does not exist on type '{ altura: number; anchura: number; }'. Did you mean 'anchura'? ts(2551)

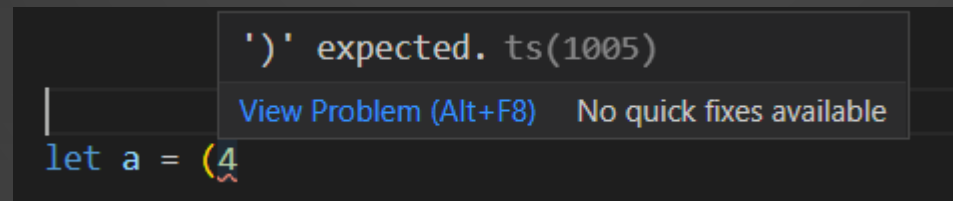
Ejemplo1.ts(1, 34): 'anchura' is declared here.

any

[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

# ¿CUÁLES SON LAS CARACTERÍSTICAS QUE TYPESCRIPT AGREGA A JAVASCRIPT?

- Sintaxis
- TypeScript es un superset de JavaScript por lo cual podemos escribir JavaScript dentro de un archivo TypeScript. Por sintaxis nos referimos a la manera en que se escribe el código para un programa. Por ejemplo el siguiente código tiene un error debido a que no se ha cerrado el paréntesis.



```
let a = (4
```

`) ' expected. ts(1005)

[View Problem \(Alt+F8\)](#) No quick fixes available

- *TypeScript no considera ningún bloque de código de JavaScript como un error de su sintaxis. Esto quiere decir que se puede utilizar cualquier bloque de JavaScript y ejecutarlo dentro de TypeScript sin preocuparnos de como ha sido escrito.*



# TIPOS

- TypeScript es un superset tipado (typed superset), lo que quiere decir que añade reglas acerca de como deben ser utilizados los valores. En el ejemplo anterior observamos que `rectangulo.anchuRa` no es un error de sintaxis sino que se ha escrito de forma incorrecta el nombre de la propiedad.

The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type. ts(2363)

[View Problem \(Alt+F8\)](#) No quick fixes available

```
console.log(4/[ ])
```

- Si bien es posible que la intención sea realmente dividir un número entre un arreglo, sin embargo la mayoría de las veces este es un error de programación. El verificador de tipos de TypeScript está diseñado para permitir que los programas puedan capturar errores comunes tanto como sea posible.

# ERRORES EN TIEMPO DE EJECUCIÓN

- TypeScript también es un lenguaje de programación que preserva el comportamiento de en ejecución de JavaScript. Por ejemplo, dividir entre cero en JavaScript produce un error Infinity en lugar de disparar un error en tiempo de ejecución (runtime exception).
- Esto quiere decir que si copias código JavaScript a TypeScript, esta garantizado que se ejecutara, incluso si TypeScript determina que existen errores en los tipos.
- El mantener el comportamiento en ejecución es uno de los fundamentos de TypeScript debido a que permite una transición sencilla entre los dos lenguajes sin tener que preocuparnos por aquellas diferencias que pudieran evitar que se empiece a migrar a este nuevo lenguaje.

# ELIMINACIÓN DEL TIPADO

- Cuando el compilador termina de revisar el código, elimina los tipos para producir un resultado “compilado”. Esto quiere decir que cuando el código ha sido compilado, el JavaScript resultante no contiene información acerca de los tipos.
- TypeScript nunca cambia el comportamiento de un programa que esta basado en tipos inferidos. Puede ser que aparezcan errores durante la compilación, pero el sistema de tipado no tiene ingerencia en como se ejecuta el programa.
- TypeScript no provee ninguna librería runtime adicional. Los programas utilizan las mismas librerías estándar (o librerías externas) que el resto de programas escritos en JavaScript, así que no se requiere volver a aprender nuevas librerías.

# ¿DEBO APRENDER TYPESCRIPT O JAVASCRIPT?

- Esta pregunta es muy común. La realidad es que no se puede aprender TypeScript sin implícitamente aprender JavaScript ya que TypeScript comparte la sintaxis y comportamiento de JavaScript, de tal forma que aprender JavaScript implica entender como TypeScript funciona.
- Si te encuentras en la necesidad de buscar información acerca de como escribir un algoritmo en TypeScript recuerda que: TypeScript es JavaScript con un compilador que revisa el tipado. La forma en la que por ejemplo se ordena un arreglo en TypeScript es la misma forma en la que se hace en JavaScript.
- Muchas de las preguntas que puedan aparecer en relación a como hacer algo en TypeScript pueden son en realidad preguntas de como hacerlo en JavaScript.



# ¿CÓMO INSTALAR TYPESCRIPT?

- Existen dos formas de instalar TypeScript:
  - Utilizando npm.
  - Instalando el plugin de TypeScript en Visual Studio Code.
- Visual Studio Code incluye TypeScript por default. Si no tienes Visual Studio Code puede descargarlo desde <https://code.visualstudio.com/>.

# ¿CÓMO INSTALAR EL COMPILADOR DESDE LA LÍNEA DE COMANDOS?

- Para instalar TypeScript mediante npm:

```
C:\Users\Geovany> npm i -g typescript
```

# COMPILAR UN ARCHIVO DE TYPESCRIPT

- Abre tu editor de código, crea un archivo llamado main.ts

```
main.ts > ...  
1  function saludar(nombre){  
2      return "Hola, " + nombre;  
3  }  
4  
5  console.log(saludar("Geovany"));  
6  
7  |
```

# ¿CÓMO COMPILAR EL PROGRAMA ESCRITO EN TYPESCRIPT?

- El compilador de TypeScript transforma el código TypeScript en código JavaScript, para ello desde la línea de comando invocamos el compilador y enviamos el nombre del archivo como parámetro.

```
PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> tsc main.ts
PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> node main.js
Hola, Geovany
PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> █
```

# TIPOS, BOOLEAN

- Boolean
  - El primer tipo que vamos a explorar es el tipo Boolean (booleano). En TypeScript los valores que puede tener boolean son true (verdadero) y false (falso).

```
let esVerdadero = true;  
console.log(esVerdadero);
```



# TIPOS, NUMBER

- Así como en JavaScript, todos los números en TypeScript son flotantes o enteros. Estos valores flotantes obtienen el valor `number` mientras que los `BigIntegers` obtienen el valor `bigint`. Además de las literales hexadecimales y decimales, TypeScript también soporta binarios y octales que han sido agregados en ECMAScript 2015.

```
let entero: number = 6;  
let hexadecimal: number = 0xf00d;  
let binario: number = 0b1010;  
let octal: number = 0o744;
```

# TIPOS, STRING

- Un tipo de valor fundamental para poder construir programas en JavaScript es el tipo string. Estas cadenas de texto se definen de dos formas: por bloques de texto con comillas sencillas o dobles.

```
let marca: string = 'toyota';  
let modelo: string = "tacoma";
```

- Además de las comillas sencillas y dobles también es posible utilizar template strings, los cuales permiten crear cadenas de caracteres de múltiples líneas mediante el carácter backtick, además de poder embeber variables dentro de este template mediante la expresión `${variable}`.

```
let nombre: string = "Raul";  
let apellido: string = "Jimenez";  
let impresion: string = `  
Nombre: ${nombre}  
Apellido: ${apellido}  
`;
```

# TIPOS, ARRAYS

- TypeScript al igual que JavaScript, permite trabajar con arreglos de valores. Los arreglos pueden ser escritos de dos formas posibles. La primera es usando los paréntesis [] para denotar un arreglo del tipo definido.

```
let listaDeNumeros : number[] = [1, 2, 3];
```

- En el ejemplo anterior hemos definido que la variable listaDeNumeros puede solo contener números. En TypeScript si intentamos agregar un tipo diferente dentro de un arreglo nos generará un error.

```
let listaDeNumeros : number[] = [1, 2, 3];
```

Argument of type 'string' is not assignable to parameter of type 'number'. ts(2345)

[View Problem \(Alt+F8\)](#) No quick fixes available

```
listaDeNumeros.push('a');
```



- Otra forma de declarar el mismo arreglo es el uso del constructor `Array<tipo>`.

```
let listaDeNumeros: Array<number> = [1, 2, 3];
```

- *Ambas formas de declarar el arreglo producen el mismo efecto.*

# TIPOS, TUPLES

- Las tuplas permiten expresar un arreglo con un número fijo conocido de elementos, pero que no requieren ser los mismos. Por ejemplo para representar una tupla que contenga un string y un number.

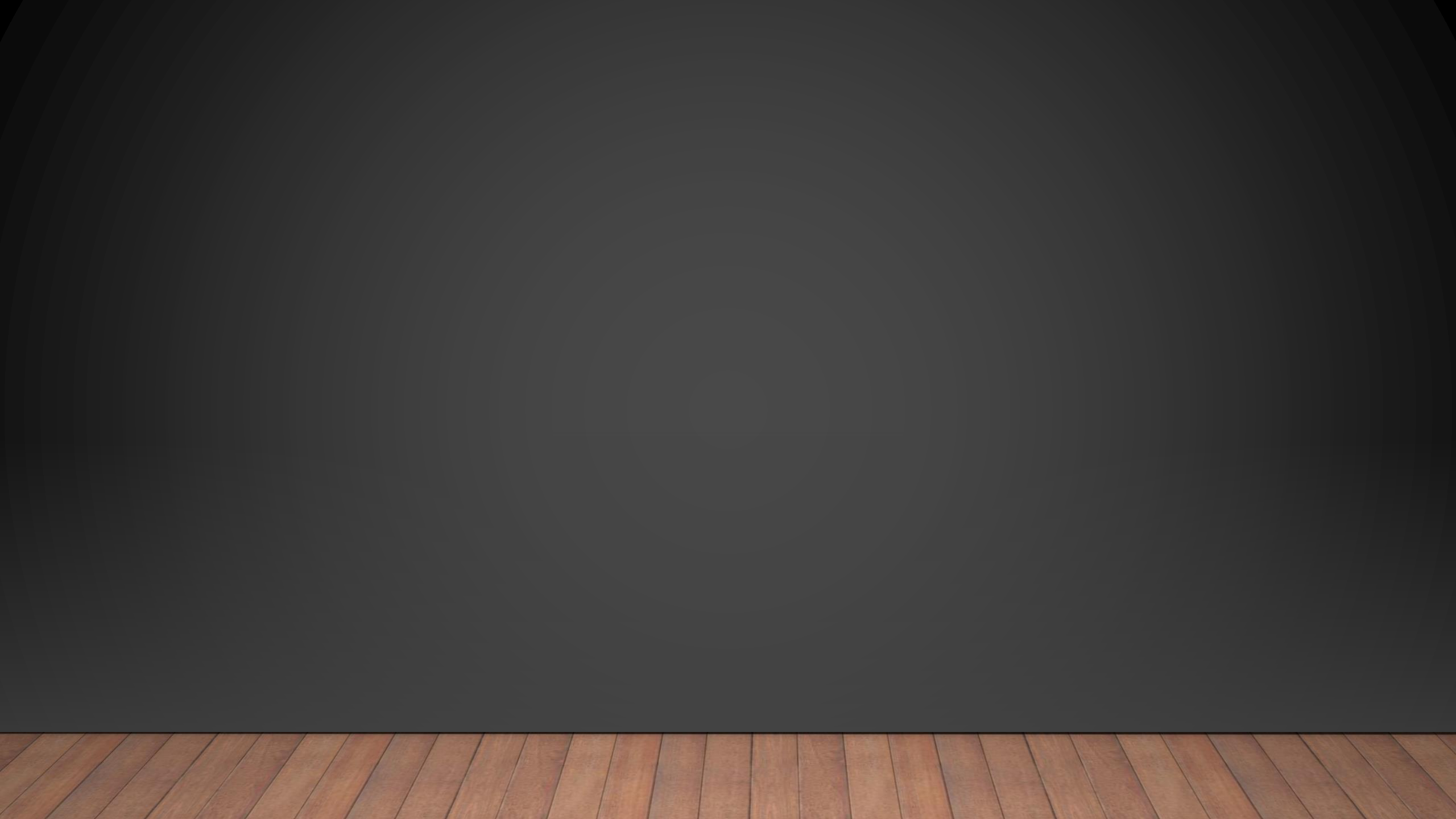
```
let futbolista: [string, number];  
  
futbolista = ['Raul Jimenez', 28]
```

- *Si intentamos capturar otro tipo de valores que no sean consistentes con los que se han definido en la tupla, TypeScript nos generará un error.*

- Para acceder a los elementos de una tupla tenemos que utilizar su índice.

```
console.log(`El nombre es ${futbolista[0]}`)  
console.log(`Su edad es ${futbolista[1]}`)
```

- *Si se intenta acceder un índice fuera del rango de la dimensión de la tupla, se generará un error.*



# TIPOS, ENUMS

- Un agregado bastante útil que proporciona TypeScript a JavaScript es la posibilidad de utilizar enum. Como en cualquier lenguaje similar a C#, un enum es una forma de dar nombres mas amigables a un grupo de sets de valores numéricos.

```
enum MarcasDeAutos {  
    Toyota,  
    Chevrolet,  
    Ford  
}  
  
let tacoma: MarcasDeAutos = MarcasDeAutos.Toyota;  
  
console.log(tacoma);
```



- En el ejemplo anterior se imprimiría 0 ya que los enums empiezan en dicho valor. Sin embargo este comportamiento puede ser modificado de forma que inicie por ejemplo en 100.

```
enum MarcasDeAutos {  
    Toyota = 100,  
    Chevrolet,  
    Ford  
}  
  
let tacoma: MarcasDeAutos = MarcasDeAutos.Toyota;  
  
console.log(tacoma);
```

- También es posible asignar un valor a cada uno de los elementos de un enum.
- En los ejemplos anteriores hemos accedido el valor de un enumerador, pero que pasa si quisieramos obtener el nombre del índice donde este se encuentra, para ello hacemos referencia de la posición numérica de dicho índice.

```
console.log(MarcasDeAutos[0])
```

# TIPOS, ANY

- En ciertas ocasiones no toda la información se encuentra disponible o la declaración de un tipo nos puede llevar a tener un error inesperado, esto puede ocurrir para valores del código que han sido escritos sin utilizar TypeScript o con una librería de un tercero.

```
let variableSinTipo: any = 'hola pedrito';  
variableSinTipo = 100;
```

# TIPOS, UNKNOWN

- En ocasiones necesitamos describir un tipo de variable el cual no conocemos. Estos valores pueden provenir de fuentes cuyos valores son dinámicos como una API. En estos casos, necesitamos indicarle al compilador y a los programadores que actualicen el código en un futuro, que esta variable puede tomar cualquier tipo de valor, para ello le asignamos el tipo unknown.

```
let valorDesconocido: unknown = 4;  
valorDesconocido = true;
```

- <https://blogs.msdn.microsoft.com/typescript/2018/07/12/announcing-typescript-3-0-rc/>

# TIPOS, VOID

- El tipo void puede considerarse como el opuesto a any, mientras any representa cualquier tipo, void representa ningún tipo. Algunas funciones como las de impresión son un ejemplo del uso de void ya que estas no retornan ningún valor.

```
function saludar2(): void {  
    console.log("Hola Mundo... Cruel");  
}  
  
saludar2();
```



# TIPOS, NULL Y UNDEFINED

- En TypeScript, los tipos undefined y null pueden solo tener el valor undefined y null respectivamente. Así como el tipo void, los tipos undefined y null no resultan tan útiles.

```
let variableSinDefinir: undefined = undefined;  
let variableNula: null = null;
```

# TIPOS, NEVER

- El tipo never representa el tipo de valores que nunca ocurren. Por ejemplo, never es retornado por la expresión de una función que siempre lanza una excepción o alguna que nunca retorna valores.
- El tipo never es un subtipo, que es assignable a cualquier tipo, sin embargo ningún tipo es un subtipo de never, un tipo never solo puede ser asignado con el valor never. Incluso any no puede ser asignado a never.

```
// esta funcion no tiene un punto final ya que dispara una excepcion
function error(mensaje: string): never {
    |   throw new Error(mensaje);
}

// esta funcion no tiene un punto final ya que dispara un error
function fallo(): never {
    |   return error("Reportar fallo");
}

// esta funcion no finaliza ya que posee un loop infinito
function loopInfinito() : never {
    |   while(true){}
}
```

# TIPOS, OBJECTS

- El tipo object representa un tipo no primitivo, cualquier cosa que no sea number, string, boolean, bigint, symbol, null o undefined.
- Con el tipo object, las APIs como Object.create pueden ser mejor representadas.

```
declare function crear(o: object): void;

crear({ prop: 0 })
crear(null);
crear(undefined);
crear([]);

// false es un tipo primitivo, por lo cual se generara un error
crear(false);
```

# TIPOS, UNIONS

- El sistema de tipos de TypeScript permite crear nuevos tipos utilizando una variedad de operadores. Ahora que sabemos como escribir algunos tipos es tiempo de combinarlos en formas interesantes.



# ¿CÓMO DEFINIR UTILIZAR UNION EN TYPESCRIPT?

- La primera forma de combinar tipos es utilizando uniones. En una unión los tipos que forman parte de esta se le llaman miembros.
- Supongamos que tenemos una función que imprime el valor de un identificador, este puede ser un número o un string.

```
function imprimirId(id: number | string) {  
  console.log(`El id es ${id}`);  
}  
  
imprimirId(1);  
imprimirId('abc');
```

# ¿CÓMO TRABAJAR CON UNIONES?

- Supongamos ahora que dentro de nuestra función deseamos utilizar los métodos asociados a id que puede ser number o string. Como cada uno de estos tipos tiene diferentes métodos se hace necesario primero hacer una verificación del tipo para poder interactuar con este.

```
function imprimirId(id: number | string) {  
  if (typeof id === "string") {  
    console.log(`El id es ${id as string.toUpperCase()}`);  
  } else {  
    console.log(`El id es ${id as number.toFixed(2)}`);  
  }  
}  
  
imprimirId('este_es_mi_id');  
imprimirId(100.234234123);
```

# TIPOS, TYPE ASSERTION

- En algunas ocasiones te encontraras en una situación en la cual tendrás mas información acerca de que tipo esperar que el propio TypeScript. Usualmente, esto pasa cuando sabes que el tipo de una entidad podría ser mas específica que el tipo que tiene en ese momento.
- Los type assertions son una forma de decirle al compilador que debe confiar en ti, porque entiendes lo que estas haciendo. Un type assertion es como el type cast de otros lenguajes, pero se ejecuta sin validar ninguna verificación o reestructurando datos. No tiene impacto durante la ejecución y es manejado exclusivamente por el compilador. TypeScript asume que el programador ha realizado las verificaciones necesarias para cerciorarse que el tipo corresponde al que dice ser.

- Los type assertions tienen dos formas.
  - Uno es la sintaxis usando as.

```
let algunValor: unknown = "esto es un string";  
let longitudDelString: number = (algunValor as string).length;
```

- Otra forma de llevar a cabo la misma operación es utilizando <tipo> antes de la variable.

```
let algunValor2: unknown = "este es un string";  
let longitudDelString2: number = (<string>algunValor).length;
```



# TIPOS, FUNCTIONS

- Las funciones son el componente mas básico del como los datos son transferidos a través de JavaScript. TypeScript permite especificar los tipos de los parámetros recibidos así como del tipo de valor retornado.

# ANOTACIONES PARA LOS PARÁMETROS DE UNA FUNCIÓN

- Cuando se declara una función es posible añadir anotaciones después de cada uno de los parámetros de manera que estos indiquen el tipo de cada uno de estos.

```
function saludar3(nombre: string){  
    console.log(`Hola ${nombre}`);  
}  
  
saludar3("Nubia");
```

# TIPOS DE VALOR DE RETORNO DE LA FUNCIÓN

- Es posible también definir el tipo de valor que retorna la función.

```
function elevarAlCuadrado(base: number): number {  
    return base * base;  
}  
  
let numeroBase = 10;  
let numeroAlCuadrado = elevarAlCuadrado(numeroBase);  
console.log(numeroAlCuadrado);
```

# FUNCIONES ANÓNIMAS

- Las funciones anónimas son un poco diferentes de las funciones declarativas. Cuando una función aparece en un lugar en donde TypeScript determina como la función va a ser invocada, los parámetros de esta función son asignados automáticamente.

```
const nombres = ["Juan", "Pedro", "Luis"];

nombres.forEach(function (s) {
  console.log(s.toUpperCase());
});

nombres.forEach((s) => {
  console.log(s.toUpperCase());
});
```

- En el ejemplo anterior hemos iterado la variable nombres mediante funciones anónimas. En el segundo caso utilizando una función que además de ser anónima es una función tipo flecha o arrow function. En ambos ejemplos TypeScript recibe la función y puede determinar el tipo de los argumentos, en este caso a partir del arreglo que contiene strings.
- Al igual que en otros escenarios en donde el tipo puede ser determinado es posible omitir las anotaciones, pero si considera que estas ayudaran a la legibilidad y a entender mejor el algoritmo, se recomienda que se utilicen.



# TIPOS, ALIASES

- Hemos estado utilizando tipos directamente en anotaciones. Esto es conveniente, pero es común querer utilizar el mismo tipo mas de una ocasión y referirse a el con un nombre. Un **alias de tipo** es un nombre dado a cualquier tipo.

```
type Punto = {  
  x: number;  
  y: number;  
}  
  
function imprimirCoordenada(punto: Punto) {  
  console.log(`La coordenada x es : ${punto.x}`);  
  console.log(`La coordenada y es : ${punto.y}`);  
}  
  
imprimirCoordenada({ x: 10, y: 25 });
```

- En este ejemplo al utilizar un alias podemos proporcionar una lista de propiedades de las cuales consta el parámetro punto.
- Para crear un alias usamos type, en este caso hemos creado un alias con dos propiedades tipo number pero esto no quiere decir que deban ser ambas iguales.

# TIPOS, INTERFACES

- Uno de los principios fundamentales de TypeScript es el que la verificación del tipado se enfoca en la forma que tienen los valores. Esto es a veces conocido como **duck typing (tipado pato)** o **structural subtyping (tipado subestructurado)**. En TypeScript, las interfaces juegan el rol de nombrar estos tipos, y son una forma poderosa de definir contratos dentro del código así como contratos fuera del proyecto.

```
function imprimirEtiqueta(etiqueta: { label: string }) {  
    console.log(etiqueta.label);  
}  
  
let miEtiqueta = { numero: 10, label: "Esta es mi etiqueta" };  
imprimirEtiqueta(miEtiqueta);
```

- El verificador de tipos revisa el proceso que invoca la función imprimirEtiqueta. Esta función tiene un solo parámetro que requiere que el objeto que se envía llamado etiqueta tiene una propiedad llamada label del tipo string. Si bien el objeto tiene otra propiedad llamada numero, como podemos ver no es necesario que el objeto miEtiqueta tenga exactamente las mismas propiedades del contrato del argumento etiqueta: {label: string}, pero si que implementa aquellas que forman parte del contrato.

- Podemos reescribir el ejemplo anterior definiendo una interface que describe los requerimientos anteriores.

```
interface Etiqueta {  
  label: string;  
}  
  
function imprimirEtiqueta(etiqueta: Etiqueta) {  
  console.log(etiqueta.label);  
};  
  
let miEtiqueta = { numero: 10, label: "Esta es mi etiqueta" };  
  
imprimirEtiqueta(miEtiqueta);
```



- La interface Etiqueta es un nombre que podemos utilizar para describir los requerimientos del ejemplo previo. Contiene una propiedad llamada label del tipo string. Toma en cuenta que el objeto que enviamos a la función no implementa de forma implícita esta interface. La única parte relevante es que la composición del valor enviado coincida para que se considere valido.
- El type checker no requiere que las propiedades se definan en el orden de la interface.

# PROPIEDADES OPCIONALES DE LAS INTERFACES

- No todas las propiedades de una interface tienen que ser requeridas. Algunas existen solo durante ciertas condiciones o incluso pueden ni siquiera estar presentes.
- Las interfaces con propiedades opcionales son escritas de forma similar a otras interfaces pero incluyen propiedades que llevan incluido el sufijo ?.

```
interface Cuadrado {  
  color?: string;  
  ancho: number;  
}  
  
function crearCuadrado(cuadrado: Cuadrado): { area: number } {  
  const area = cuadrado.ancho * cuadrado.ancho;  
  return { area: area };  
}  
  
crearCuadrado({ ancho: 10 });
```

# PROPIEDADES DE SOLO LECTURA DE LAS INTERFACES

- Algunas propiedades pueden ser solo modificables cuando se crean los objetos. Para especificar este comportamiento se deben definir las clases como readonly (de solo lectura).

```
interface Punto2 {  
    readonly x: number;  
    readonly y: number;  
}  
  
let punto1: Punto = { x: 10, y: 20 };  
punto1.x = 20;
```

# READONLY VS CONST

- La forma mas sencilla de entender readonly es si lo comparamos con const.
- Las variables utilizan const mientras que las propiedades usan readonly.

# TIPOS, INTERFACES VS TYPES

- El uso de type e interface es muy similar, y en la mayoría de los casos es posible elegir cualquiera de ellos de forma indistinta. Casi todas las características de interface están disponibles en type, la clave para distinguir entre cuando usar una y otra es que una vez que se define un type no se le pueden agregar mas propiedades, mientras que interface es siempre extendible.



```
interface Transporte {  
  |  nombre: string;  
}
```

```
type Figura = {  
  |  nombre: string;  
}
```

# ¿CÓMO EXTENDER UNA INTERFACE?

```
interface Auto extends Transporte {  
    |   ruedas: number;  
}
```

# ¿CÓMO EXTENDER UN TYPE?

```
type Cuadrado2 = Figura & {  
  | lados: 4;  
}
```

# TIPOS, LITERALES

- Los literals (literales) son bastante útiles cuando se combinan con el uso de funciones y unions.
- Un ejemplo es cuando deseamos restringir un parámetro a una serie de valores posibles, por ejemplo una serie de strings o numbers.

```
function imprimir(estadoCivil: 'soltero' | 'casado') {  
  | console.log(estadoCivil);  
}  
  
imprimir('soltero');
```

# FUNCIONES COMO EXPRESIONES

- La manera mas simple de describir una función es como una función que opera como un tipo de expresión. Estos tipos son sintácticamente como funciones de flecha (arrow functions).

```
function saludar4(fn: (a: string) => void) {  
    fn("Hola Mundo")  
}  
  
function imprimirEnConsola(s: string) {  
    console.log(s);  
}  
  
saludar(imprimirEnConsola);
```



- En el anterior ejemplo la función saludar recibe el parámetro fn es una función (a: string) => null que recibe un string y retorna null, y esta se envía y cuya invocación se realiza dentro de la función que la recibe fn("Hola Mundo"), que a su vez pasa el argumento "Hola Mundo".

# FUNCIONES, CONSTRUCTOR SIGNATURE

- Las funciones de JavaScript pueden ser también invocadas con el operador `new`. TypeScript hace referencia a estos operadores como `constructors` (constructores) porque estos sirven para crear un objeto. Por ello se puede escribir un `constructor signature` (firma para el constructor) añadiendo la palabra `new` enfrente de la firma.

TS main.ts > ...

```
1 interface Transporte {
2     nombre: string;
3 }
4 class Caballo implements Transporte {
5     constructor(public nombre: string) {}
6 }
7 class Automovil implements Transporte {
8     constructor(public nombre: string) {}
9 }
10 type ConstructorDeTransporte = {
11     new (nombre: string): Transporte;
12 };
13 function construirTransporte(ctr: ConstructorDeTransporte, nombre: string)
14     return new ctr(nombre);
15 }
16
17 const miCaballo = construirTransporte(Caballo, "Paso Fino");
18 const miAutomovil = construirTransporte(Automovil, "Toyota");
19
20 console.log("Mi caballo se llama " + miCaballo.nombre);
21 console.log("Mi automovil es un " + miAutomovil.nombre);
```

TERMINAL

PROBLEMS

OUTPUT

DEBUG CONSOLE

```
PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> tsc main.ts
PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> node main.js
Mi caballo se llama Paso Fino
Mi automovil es un Toyota
PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> []
```

- En el ejemplo hemos construido una función a la cual se le pasa un objeto que posee un constructor como parte de su firma. En este caso dicho constructor debe retornar un objeto que implemente la interface Transporte.

# FUNCIONES, PARÁMETROS OPCIONALES

- Las funciones en JavaScript usualmente tienen un número de argumentos variable. Por ejemplo el método `toFixed` del tipo `number` toma como segundo parámetro un número que indica la cantidad de dígitos a los cuales debe redondear.



```
function f(n: number) {  
  console.log(n.toFixed()); // no se especifican argumentos  
  console.log(n.toFixed(3)); // se especifica un argumento  
}
```

- Para poder reproducir esta característica dentro de TypeScript utilizamos el signo de interrogación ? que indica que el parámetro es opcional.

```
function f2(n?: number) {  
  // ...  
}
```

- Si bien el parámetro ha sido especificado como un tipo number, cuando no se envía un argumento a la función este tiene el tipo undefined, esto se debe a que los parámetros no especificados en JavaScript tienen por default el valor undefined.
- También es posible asignar un valor por default y omitir el tipo que es inferido desde su asignación.

```
function f3(n = 10) {  
  // ...  
}
```

# PARÁMETROS OPCIONALES EN LOS CALLBACKS

- Una vez que se ha aprendido como utilizar los parámetros opcionales, es muy común cometer el siguiente tipo de errores.

```
function miIterador(arr: any[], callback: (arg: any, index?: number) => void) {  
    for(let i=0; i<=arr.length; i++){  
        callback(arr[i], i);  
    }  
}
```

- El error que por lo general se comete al hacer el parámetro index opcional, es que cualquiera de las dos siguientes llamadas a la función sean validas.

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

- Pero esto no funciona así, ya que lo que TypeScript entiende es que la función callback puede ser invocada desde el iterador también con un solo argumento.

```
function miIterador2(arr: any[], callback: (arg: any, index?: number) => void) {  
    for(let i=0; i<=arr.length; i++){  
        // aquí es en donde index es opcional o no  
        callback(arr[i]);  
    }  
}
```

- Incluso dentro de la definición del callback TypeScript envía un mensaje indicando que index puede ser undefined.



- En este último caso veremos que el compilador tira el error `Object is possibly 'undefined'`.
- En JavaScript si se llama a una función con mas argumentos de los parámetros que existen, los argumentos extras son simplemente ignorados. El comportamiento de TypeScript es el mismo. Las funciones con menos parámetros (con los mismos tipos) pueden siempre sustituir a aquellas con mas parámetros.

# FUNCIONES, OVERLOAD

- Algunas funciones de JavaScript pueden ser invocadas con diferentes tipos y número de argumentos. Por ejemplo, es posible escribir una función que produce un Date que toma un timestamp (un argumento) o un mes/día/año (tres argumentos).
- En TypeScript, podemos especificar una función que puede ser llamada de formas diferentes utilizando overload signatures o sobrecarga de funciones. Para poder hacer esto escribe un número de posibles definiciones distintos de una función (usualmente dos o mas), seguido del cuerpo de la función.

- Al igual que con los generics, existen algunas guías para el buen uso de la escritura de las funciones con sobrecarga. Al seguir estos principios nos aseguramos que sean sencillas de invocar, sencillas de entender y sencillas de implementar.

```
function longitud(a: any[]): number;  
function longitud(x: string): number;  
function longitud(x: any): number {  
    return x.length;  
}  
  
console.log(longitud("hola mundo"));  
console.log(longitud([1, 2, 3, 4, 5]));
```

- En este ejemplo tenemos una función con sobrecarga que imprime la longitud de un texto o un string. Si bien cumple con su función, podemos simplificar esta función con sobrecarga de una forma mucho mas sencilla usando unions.

```
function calcularLongitud(x: any[] | string) {  
    return x.length;  
}  
  
console.log(calcularLongitud("hola mundo"));  
console.log(calcularLongitud([1, 2, 3, 4, 5]));
```

# FUNCIONES, USO DE 'THIS'

- TypeScript infiere cual será el elemento al cual `this` hace referencia, por ejemplo:

```
main.ts > ...
1  const usuario = {
2      id: 123,
3      admin: false,
4      volverseAdmin: function() {
5          this.admin = true;
6      },
7  };
8
9  console.log(usuario.admin);
10 usuario.volverseAdmin();
11 console.log(usuario.admin);
```

TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

Windows PowerShell

Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma <https://aka.ms/pscore6>

PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> **tsc** main.ts

PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> **node** main.js

**false**

**true**

PS C:\Users\Geovany\Desktop\Angular\Ejemplos TypeScript> █

- Cuando ejecutamos la función `volverseAdmin()` el valor de `admin` cambia de `false` a `true`.
- Ahora supongamos que en lugar de utilizar una función utilizamos una función tipo flecha.

```
1  const usuario2 = {  
2    id: 123,  
3    admin: false,  
4    volverseAdmin: () => {  
5      this.admin = true;  
6    },  
7  };  
8  
9  console.log(usuario2.admin);  
10 usuario2.volverseAdmin();  
11 console.log(usuario2.admin);
```



# FUNCIONES, REST PARAMETERS

- Además de utilizar parámetros opcionales o sobrecarga para hacer que las funciones acepten una variedad de argumentos, también es posible definir funciones que tomen un número indeterminado de argumentos utilizando rest parameters.
- Un parámetro rest aparece al final de la lista de parámetros con un prefijo de tres puntos seguido del nombre del parámetro ...nombre.

```
1 function multiplicar(n: number, ...m: number[]): number {  
2     return m.reduce((p, c) => {  
3         return p * c;  
4     }, n);  
5 }  
6  
7 console.log(multiplicar(2, 2));  
8 console.log(multiplicar(2, 2, 3));  
9 console.log(multiplicar(2, 2, 3, 4));
```

- La función multiplicar recibe un segundo parámetro llama m, este almacena todos los números adicionales como parte de un arreglo tipo number y despues los multiplica uno a uno.

# FUNCIONES, PARAMETER DESTRUCTURING

- La destructuración de parámetros permite romper objetos en una o mas variables locales. En el caso de la función permite romper los argumentos para crear variables con un scope al nivel del cuerpo de la función.

```
function sumar(num) {  
    return num.a + num.b + num.c;  
}  
  
const numeros = { a: 1, b: 2, c: 3 };  
  
console.log(sumar(numeros));
```

- Ahora supongamos que deseamos descomponer el parámetro num en variables independientes para realizar la suma. Para ello podemos usar la deestructuración.

```
function sumar({ a, b, c }): number {  
    return a + b + c;  
}  
  
console.log(sumar({ a: 1, b: 2, c: 3 }));
```

- TypeScript nos permite definir los tipos que el objeto a desestructurar posee.

```
function sumar({ a, b, c }: { a: number; b: number; c: number }): number {  
    return a + b + c;  
}  
  
console.log(sumar({ a: 1, b: 2, c: 3 }));
```

- tomar en cuenta que la sintáxis de deestructuración es similar a la de definición de tipos. Pero a diferencia de esta separa los valores por comas , (deestructuración) y no por punto y coma ; (definición de tipos).



# OBJECT TYPES

- En JavaScript, la forma mas básica en la que agrupamos y enviamos datos es a través de objetos. En TypeScript representamos estos a traves de **object types (tipos de objeto)**.
- Éstos object types pueden ser anónimos.

```
function saludar(persona: { nombre: string; edad: number }) {  
    return `Hola ${persona.nombre}`;  
}  
  
console.log(saludar({ nombre: "Luis", edad: 22 }));
```

- o pueden ser nombradas como una interface

```
interface Persona {  
  nombre: string;  
  edad: number;  
}  
  
function saludar(persona: Persona) {  
  return `Hola ${persona.nombre}`;  
}  
  
console.log(saludar({ nombre: "Elena", edad: 25 }));
```

- o puede también ser nombradas como un alias

```
type Persona = {  
  nombre: string;  
  edad: number;  
}  
  
function saludar(persona: Persona) {  
  return `Hola ${persona.nombre}`;  
}  
  
console.log(saludar({ nombre: "Anabel", edad: 30 }));
```

- En cualquiera de los ejemplos que acabamos de mostrar, es posible obtener la propiedad nombre (que es de tipo string) y edad (que es de tipo number).

# OBJECT TYPES, PROPERTY MODIFIERS

- Cada propiedad que existe en un objeto especifica un grupo de cosas: el tipo, si la propiedad es opcional, y si la propiedad es de solo lectura (no escribible) o no.

# PROPIEDADES OPCIONALES

- La mayoría de las veces, nos vamos a encontrar lidiando con objetos que pueden tener o no una propiedad definida. En estos casos, marcamos esta propiedad como opcional añadiendo el signo de interrogación ? al final del nombre de la propiedad.

```
interface Computadora {
  os: 'windows' | 'linux' | 'mac';
  monitor?: 'crt' | 'led';
  memoria: number;
  procesador: 'intel' | 'amd'
}

function imprimir(computador: Computadora) {
  console.log(`Sistema operativo: ${computador.os}`);
  console.log(`Memoria: ${computador.memoria}`);
  console.log(`Procesador: ${computador.procesador}`);
}

imprimir({
  os: 'windows',
  memoria: 8,
  procesador: 'intel',
});
```



- En este ejemplo, la propiedad monitor es considerada opcional. Podemos elegir si enviarla o no como parte del objeto que se envía como argumento a la función imprimir. Si no declaramos un valor para esta su valor sera undefined, es decir no definida.
- Si deseamos definir unos valores por default a las propiedades del parámetro computador, podemos hacer utilizando la deestructuración.

# OBJECT TYPES, READONLY PROPERTIES

- Las propiedades pueden ser marcadas como propiedades de solo lectura o readonly dentro de TypeScript. En una propiedad tipo readonly solo podemos asignarle un valor cuando la instancia es creada, pero a partir de ese momento no podemos modificarlo.

```
interface Perro {  
    readonly raza: string;  
}  
  
const miCachorro: Perro = { raza: "Shitzu" };  
  
console.log(`La raza de mi cachorro es: ${miCachorro.raza}`);
```

- Sin embargo si complementamos el programa intentando cambiar la raza de nuestro cachorro (lo cual en teoría no es posible), el compilador nos generará un error.

```
interface Perro {  
  readonly raza: string;  
}  
  
const miCac  
console.log  
miCachorro.raza = 'pitbull'
```

Cannot assign to 'raza' because it is a read-only property. ts(2540)

(property) Perro.raza: any

[View Problem \(Alt+F8\)](#) No quick fixes available

- Es importante entender las expectativas de lo que implica el uso de readonly. Es una señal de alerta durante el desarrollo en TypeScript acerca de como los objetos deben de ser utilizados. TypeScript no toma en cuenta las propiedades de dos tipos si estos son readonly al momento de verifica su compatibilidad, de tal forma que las propiedades readonly pueden ser cambiadas mediante el uso de un alias.

```
interface Persona {  
  edad: number;  
}  
  
interface EdadNoEscribible {  
  readonly edad: number;  
}  
  
const Luis: Persona = { edad: 20 };  
  
const Pedro: EdadNoEscribible = Luis;  
  
Luis.edad++;  
Pedro.edad++; // <-- esto generara un error
```

# OBJECT TYPES, EXTENDER TIPOS

- Es muy común tener tipos que pueden ser versiones de otros tipos. Por ejemplo podemos tener el tipo Direccion que describe los campos que son necesarios para el envío de correspondencia.

```
interface Direccion {  
    nombre: string;  
    calle: string;  
    numero: number;  
    ciudad: string;  
    pais: string;  
    codigoPostal: string;  
}
```



- Sin embargo en algunos casos es necesario agregar alguna información adicional, por ejemplo los departamentos comparten el mismo número sobre la calle, suelen tener un identificador interno que puede ser una combinación de números y letras.

```
interface DireccionDeApartamento {  
    nombre: string;  
    calle: string;  
    numero: number;  
    unidad: string;  
    ciudad: string;  
    pais: string;  
    codigoPostal: string;  
}
```

- Esto implica que tengamos que repetir todas las propiedades de la interface Direccion. Para evitar esto podemos usar el concepto de extensión, que permite utilizar la interface Base y solo agregar aquellos que hacen falta.

```
interface Direccion {  
    nombre: string;  
    calle: string;  
    numero: number;  
    ciudad: string;  
    pais: string;  
    codigoPostal: string;  
}  
  
interface DireccionDeUnDepartamento extends Direccion {  
    unidad: string;  
}
```

# OBJECT TYPES, EXTENDER MÚLTIPLES TIPOS

- Las interfaces también permiten extender desde múltiples interfaces. Supongamos que tenemos una computadora MacbookPro, la cual esta conformada por los valores de las interfaces Computadora, SistemaOperativo y Portatil. Por ende podemos combinar todas estas interfaces para asignar valores a nuestro nuevo objeto.

```
interface Computadora {
    memoria: string;
    procesador: string;
    hdd: string;
}

interface SistemaOperativo {
    so: string;
    version: string;
}

interface Portatil extends Computadora, SistemaOperativo {
    bateria: string;
    monitor: string;
    teclado: string;
}

interface Servidor extends Computadora, SistemaOperativo {
    conexion: string;
}
```

```
const macbookPro: Portatil = {
    memoria: "16G",
    procesador: "intel",
    hdd: "1TB",
    so: "osx",
    version: "catalina",
    bateria: "litio",
    monitor: "17 pulgadas",
    teclado: "español",
};

const ubuntuServer: Servidor = {
    memoria: "64G",
    procesador: "intel",
    hdd: "4TB",
    so: "ubuntu",
    version: "trusty",
    conexion: "ethernet",
};
```

- Teniendo como base las interfaces Computadora y Sistema Operativo, podemos crear otras 2 interfaces Portatil y Servidor. En el caso de Portatil requerimos de un monitor y una batería, un monitor y un gabinete de cierta dimensión. Los servidores por su lado solo se administran desde la distancia, por lo que solo requerimos conectarlo para utilizarlo, por ello especificamos solo la configuración de la conexión.

# OBJECT TYPES, INTERSECTION TYPES

- Las interfaces permiten construir nuevos tipos a partir de extender otros. TypeScript permite esta construcción a partir de la intersección que se utiliza para combinar tipos de datos existentes.
- Una tipo de intersección esta definida por el uso del operador &.



```
interface Computadora {  
    memoria: string;  
    procesador: string;  
    hdd: string;  
}  
  
interface SistemaOperativo {  
    so: string;  
    version: string;  
}  
  
type Portatil = Computadora & SistemaOperativo;  
  
const macbookPro: Portatil = {  
    memoria: "16G",  
    procesador: "intel",  
    hdd: "1TB",  
    so: "osx",  
    version: "catalina",  
};
```

- Retomado el ejemplo en donde combinamos las interfaces, podemos ver que podemos utilizar la intersección para crear un nuevo alias type. En este caso el alias Portatil solo contiene las propiedades de Computadora y SistemaOperativo.

# OBJECT TYPES, INTERFACES VS INTERSECTIONS

- Hemos visto dos formas en las cuales se pueden combinar tipos que tienen cierta similitud y simultáneamente diferentes. Con las interfaces utilizamos la clausula `extends` para poder extender desde otros tipos, algo similar a lo que sucede con las intersecciones. En ambos casos es posible dar un nombre al resultado de extender o interseccionar tipos.
- La principal diferencia entre extensión e intersección es la forma en la que cada una de estas formas manejan la resolución de conflictos, y esta suele ser la razón por la cual se elige entre una forma u otra.

# OBJECT TYPES, GENERICS

- Imaginemos que tenemos un tipo Caja que contiene cualquier valor posible como string, number, etc.

```
interface Caja {  
  contenido: any;  
}
```

- En esta ocasión, la propiedad contenido es definida como any, lo que permite trabajar con cualquier valor, pero que puede conducir a escenarios no deseados.

- Si por el contrario utilizamos unknown, esto implicaría que para aquellos casos en los cuales ya conocemos el tipo de contenido, se requeriría hacer comparaciones de precaución, o utilizar aserciones (asserts) para prevenir algunos errores.

```
interface Caja {  
    contenido: unknown;  
}  
  
let x: Caja = {  
    contenido: "hola mundo",  
};  
  
// mediante typeof podemos verificar si el tipo es string  
if (typeof x.contenido === "string") {  
    console.log(x.contenido.toLocaleLowerCase());  
}  
  
// mediante "as tipo" podemos decirle al compilador que esto es siempre string  
console.log((x.contenido as string).toLocaleLowerCase());
```

- Otro posible enfoque es poder utilizar un tipo para cada uno de los casos.

```
interface CajaNumber {  
    contenido: number;  
}  
  
interface CajaString {  
    contenido: string;  
}  
  
interface CajaBoolean {  
    contenido: boolean;  
}
```



- Sin embargo esto implica que tendremos que crear diferentes funciones/sobrecarga, para poder operar con cada uno de estos tipos.

```
function setContenido(caja: CajaNumber, nuevoContenido: string): void;  
function setContenido(caja: CajaString, nuevoContenido: number): void;  
function setContenido(caja: CajaBoolean, nuevoContenido: boolean): void;  
function setContenido(caja: { contenido: any }, nuevoContenido: any) {  
    caja.contenido = nuevoContenido;  
}
```

- *Utilizar sobrecarga para cubrir los escenarios de cada uno de los posibles tipos, no resulta ser la solución mas adecuada para solucionar este problema.*

# ¿QUÉ SON LOS GENERICS EN TYPESCRIPT?

- Una herramienta muy útil para construir soluciones que respondan a tipos dinámicos; es el uso de **generics**.

```
interface Caja<T> {  
  |   contenido: T;  
}
```

- Piensa en la Caja como una plantilla que recibe un tipo, en donde T es un contenedor que será reemplazado con algún tipo. Cuando TypeScript ve Caja<string>, va a reemplazar cada instancia de T dentro de Caja<T> con el tipo string, para que se genere así un contenido: string.

```
interface Caja<T> {  
    contenido: T;  
}  
  
let cajaDeString: Caja<string> = { contenido: "hola mundo" };  
let cajaDeNumero: Caja<number> = { contenido: 100 };  
let cajaDeFecha: Caja<Date> = { contenido: new Date() };
```

- Podemos ver que nuestra interface Caja<T> se convierte en una interface reutilizable para diferentes tipos. De igual forma podemos crear alias genéricos.

```
type Cajita<T> = {  
    contenido: T;  
};  
  
let cajaDeString2: Cajita<string> = { contenido: "hola mundo" };  
let cajaDeNumero2: Cajita<number> = { contenido: 100 };  
let cajaDeFecha2: Cajita<Date> = { contenido: new Date() };
```

# OBJECT TYPES, ARRAY TYPE

- Los generics son por lo general un contenedor de cierto tipo que trabajar de forma independiente del tipo de elementos que contiene. Es ideal para estructuras de datos trabajar de esta forma, de manera que sean reutilizables a través de diferentes tipos de datos.
- Hemos estado trabajando con el tipo `Array`. Cuando hacemos referencia a `string[]` o `number[]`, usamos la sintaxis alternativa a `Array<string>` y `Array<number>`.



```
const imprimirTareas = (v: Array<string>) => {  
  v.forEach((v) => {  
    console.log(v);  
  });  
};  
  
const misTareas: string[] = [  
  "levantarse",  
  "lavarse los dientes",  
  "sacar al perro",  
];  
  
imprimirTareas(misTareas);
```



# OBJECT TYPES, READONLYARRAY TYPE

- El ReadonlyArray es un tipo especial que describe arreglos que no deberían cambiar.

```
const miLista : ReadonlyArray<string> = ["a", "b", "c"];  
  
miLista.push("d"); // <- esta línea generaría un error
```

- El tipo ReadonlyArray es el equivalente para los arreglos de readonly para las propiedades. Cuando vemos una función que consume ReadonlyArray, nos dice que podemos pasar un arreglo a la función sin preocuparnos que este arreglo va a cambiar dentro de ella.

- A diferencia de Array, no existe el constructor ReadonlyArray que pueda ser utilizado. Por lo cual no se puede crear un arreglo ReadonlyArray de la siguiente forma.

```
const miLista2 = new ReadonlyArray('a','b','c');
```

- En lugar de ello podemos asignar un arreglo normal a uno de solo lectura.

```
const miLista3: ReadonlyArray<string> = ['a','b','c'];
```

- Como ya comentamos, sintaxis alternativa de `Array<Type>` es `Type[]`. Para el caso de `ReadOnlyArray<Type>` hay que agregar el prefijo `readonly` `readonly Type[]` que indica que el arreglo es de solo lectura.

# OBJECT TYPES, TUPLES

- Una tupla es otro tipo de Array que conoce exactamente cuantos elementos, de que tipo y en que posición contiene el arreglo.

```
type Auto = [string, number];
```

- Aquí tenemos el tipo Auto que es una tupla que contiene un string y un number. Para TypeScript la tupla Auto describe un arreglo que contiene en la posición 0 un string y en la posición 1 un number.

```
type Auto = [string, number]
```

```
const prius : Auto = ['Toyota', 2015]
```

```
const civic : Auto = ['Honda', 2016]
```

```
console.log('El Prius es marca: ', prius[0], ' y modelo: ', prius[1])
```

```
console.log('El Civic es marca: ', civic[0], ' y modelo: ', civic[1])
```

- Si intentamos escribir un tipo invalido, el compilador de TypeScript generará un error. A diferencia de los arreglos de solo lectura, las tuplas si permiten que el valor de este tipo de arreglos sea modificado siempre que este se ajuste a la definición de la tupla. También soporta operaciones como push.
- Las tuplas son muy utilizadas dentro de las convenciones de API's, cuando cada uno de los elementos tiene un sentido de obviada. Esto brinda una buena flexibilidad cuando deseamos asignar nombres a partir de la deestructuración.

```
const prius: [string, number] = ["Toyota", 2015];  
  
const [marca, modelo] = prius;  
  
console.log("La marca del prius es: ", marca);  
console.log("El modelo del prius es: ", modelo);
```



# REST TUPLES

- Las tuplas pueden contener elementos rest, este puede utilizarse solo para el último elemento de una tupla.

```
type StringNumberBooleans = [string, number, ...boolean[]];  
const a: StringNumberBooleans = ["a", 1, true, false, true];
```

# READONLY TUPLES

- Un detalle final acerca de las tuplas es que estas también pueden ser de solo lectura readonly, y esto puede ser especificado de la siguiente forma.

```
type Auto = readonly [string, number];  
  
const prius: Auto = ["Toyota", 2014];  
  
prius[0] = 'Honda'; // <- esta linea generaria un error
```

# CLASSES

- TypeScript ofrece soporte para el uso de clases `class` que fueron introducidas en ES2015.
- Así como con otras características de JavaScript, TypeScript agrega anotaciones y características adicionales a la sintaxis que permiten crear relaciones entre las clases y otros tipos.

# ELEMENTOS DE LAS CLASES

- La estructura mas básica de una clase es la siguiente...

```
class Punto {}
```

- Esta clase Punto si bien no es muy útil en esta expresión básica, va obteniendo mayor utilidad conforme se van agregando elementos a la clase.

# CAMPOS DE UNA CLASE

- La declaración de un campo en una clase crea por default una propiedad escribible.

```
class Punto {  
    x: number;  
    y: number;  
}  
  
const miPunto = new Punto();  
miPunto.x = 0;  
miPunto.y = 0;
```

- Al igual que con otros elementos de TypeScript, como las variables, si en una clase no se especifica el tipo del campo, este por default toma el valor `any`.

```
class Punto {  
  x; // <- any  
  y; // <- any  
}
```

- Otra posibilidad es asignar el tipo utilizando inferencia, asignando un valor al campo.

```
class Punto {  
  x = 0; // <- number  
  y = 0; // <- number  
}
```



# CLASSES, READONLY

- Los campos propiedades de una clase pueden ser precedidos por readonly. Esto previene que se asigne un valor al campo fuera del constructor de la clase.

```
class Saludo {  
    readonly nombre: string = "mundo";  
  
    constructor(nuevoNombre: string) {  
        if (!!nuevoNombre) {  
            this.nombre = nuevoNombre; // <- correcto, asignación es valida dentro del constructor  
        }  
    }  
  
    asignarNuevoNombre(nuevoNombre: string) {  
        this.nombre = nuevoNombre; // <- error, no se puede asignar valor fuera del constructor  
    }  
}  
  
const miNombre = new Saludo("Elio"); // <- correcto, asignación mediante el constructor  
miNombre.nombre = "Alejandro"; // <- error, no se puede asignar valor fuera del constructor
```

# CLASSES, CONSTRUCTORS

- Los constructores son muy similares a las funciones. Se pueden agregar parámetros que incluyen anotaciones acerca de su tipo, valor por default y sobrecarga.

```
class Punto {  
  x: number;  
  y: number;  
  
  //  asignatura normal utilizando valores por default  
  constructor(x = 10, y = 10) {  
    this.x = x;  
    this.y = y;  
  }  
}  
  
let miPunto = new Punto();  
console.log(miPunto.x);  
console.log(miPunto.y);
```

# SOBRECARGA DEL CONSTRUCTOR

- También podemos usar la sobrecarga para definir varias formas de trabajar del constructor.

```
class Punto {  
    // uso de sobrecarga  
    constructor(x: number, y: number);  
    constructor(s: string);  
    constructor(xs: number | string, y?: number) {  
  
    }  
}
```

Los constructores no pueden retornar anotaciones de tipo, la instancia de la clase siempre es retornada.

# CLASSES, SUPER

- Así como en JavaScript, si se tiene una clase base, es necesario invocar `super()`; dentro del cuerpo del constructor, antes de realizar cualquier llamada a `this`.

```
class Figura {  
    lados = 0;  
}  
  
class Circulo extends Figura {  
    constructor() {  
        this.lados = 2; // <- esto generará un error  
        super();  
        // <- a partir de este punto se puede utilizar this  
    }  
}
```

# CLASSES, METHODS

- Los métodos son propiedades que actúan como funciones dentro de las clases. Al igual que en las funciones y los constructores, es posible utilizar anotaciones dentro de los métodos.

```
class Video {  
  titulo: string;  
  
  constructor(titulo: string) {  
    this.titulo = titulo;  
  }  
  
  reproducir(): void {  
    console.log(`${this.titulo} se esta reproduciendo`);  
  }  
}  
  
const miVideo = new Video("año nuevo");  
miVideo.reproducir();
```



- Además de las anotaciones generales, TypeScript no agrega nada nuevo a los métodos.
- Toma en cuenta que dentro del cuerpo de un método, es necesario utilizar `this` para hacer referencia a otros métodos y/o propiedades.

```
let titulo = "mi graduación"; // (1)

class Video {
  titulo: string; // (2)

  asignarTitulo(nuevoTitulo: string) {
    titulo = nuevoTitulo; // <- esto hace referencia a (1)
    this.titulo = nuevoTitulo; // <- esto hace referencia a (2)
  }
}
```

# CLASSES, SETTERS Y GETTERS

- Las clases también pueden tener setters (métodos que asignan) y getters (métodos que retraen valores), a estos se les llama accesoros.

```
class Desfile {  
  private _participantes = 0;  
  
  get participantes(): number {  
    return this._participantes;  
  }  
  
  set participantes(v: number) {  
    this._participantes = v;  
  }  
}  
  
const desfileHoy = new Desfile();  
desfileHoy.participantes = 100;  
console.log(desfileHoy.participantes); // <- 100
```

- TypeScript tiene algunas reglas de inferencia para el uso de accessors (setters/getters).
  - Si no existe set, la propiedad es automáticamente readonly.
  - El tipo del setter es inferido del tipo de retorno del getter.
  - Si el setter tiene definido el tipo, debe coincidir con el tipo de retorno del getter.
  - No se puede tener setters y getters asociados a diferentes tipos.

# CLASSES, HERENCIA

- Así como sucede en otros lenguajes orientados a objetos, las clases en JavaScript pueden heredar de otras clases utilizando la clausula `implements`.
- Se puede utilizar `implements` para verificar que una clase satisface una interface particular. Un error será disparado si no se ha implementado correctamente.

- Por ejemplo si tenemos la siguiente interface.

```
interface Encendible {  
    encender(): void;  
}
```

- Al implementar la clase podemos ver que esto nos genera un error mientras no se haya definido el método de la interface encender.

```
class Television implements Encendible {  
}
```

- Por ello requerimos que se implementen todos los métodos de la clase encender.

```
class Television implements Encendible {  
    encender(): void {  
        console.log("El televisor se ha encendido");  
    }  
}
```

- Las clases pueden implementar diferentes interfaces, por ejemplo Class Television implements Encendible, Apagable, Sintonizable {}.



# PRECAUCIONES

- Es importante entender que la clausula implements solo verifica que la clase sea tratada como un tipo equivalente a la interface. No cambia el tipo de la clase o sus métodos de ninguna forma. Un error común es asumir que la clausula implements cambiará el tipo de la clase, pero esto no sucede.

```
interface Verificable {  
    verificar(nombre: string): boolean;  
}  
  
class NombreVerificable implements Verificable {  
    verificar(nombre): boolean { // <- nombre se convierte en un parámetro tipo any  
        return nombre.toLowerCase();  
    }  
}
```

# CLASSES, EXTENDS

- Las clases pueden extender de otra clase base. Una clase derivada tiene todas las propiedades y métodos de una clase base, y también define métodos adicionales.

```
class Animal {  
    moverse() {  
        console.log("El animal se mueve");  
    }  
}  
  
class Perro extends Animal {  
    ladrar() {  
        console.log("El perro ladra");  
    }  
}  
  
const miPerro = new Perro();  
miPerro.moverse();  
miPerro.ladrar();
```

# CLASSES, SOBRECARGA

- Una clase que extiende a otra puede reescribir sus propiedades. Si deseamos acceder a los elementos de una clase padre, cuyo clase hijo ha sobrescrito sus propiedades, podemos utilizar la sentencia `super` para acceder a ellos.
- TypeScript fuerza a una clase derivada a tener siempre un subtipo de la clase base.

```
class Padre {  
  saludar() {  
    console.log("Hola");  
  }  
}  
  
class Hijo extends Padre {  
  saludar(nombre?: string) {  
    if (!!nombre) {  
      console.log(`Hola ${nombre}`);  
    } else {  
      super.saludar();  
    }  
  }  
}  
  
const hijo = new Hijo();  
hijo.saludar();  
hijo.saludar("Luis");
```

- Es importante que una clase derivada siga las bases de la clase padre (o clase contrato). Recuerda que es muy común que se haga una referencia a una instancia de la clase derivada a través de una clase base.

```
const h : Padre = new Hijo();|
```

- Es importante seguir las indicaciones del contrato, si cambiamos la definición de alguno de los métodos heredados, de forma que se vuelven incompatibles, el compilador de TypeScript generará un error.

# CLASSES, ORDEN DE INICIALIZACIÓN DE LAS CLASES

- El orden de inicialización de las clases de JavaScript puede sorprender en algunos casos. Consideremos el siguiente código.

```
class Definicion {  
  nombre = "definicion";  
  constructor() {  
    console.log(`Mi nombre es ${this.nombre}`);  
  }  
}  
  
class Implementacion extends Definicion {}  
  
const d = new Implementacion();
```



# ¿CUÁL ES EL ORDEN DE LA INICIALIZACIÓN DE LAS CLASES?

- El orden de la inicialización de las clases es el siguiente.
  - Los campos de la clase base son inicializados.
  - El constructor de la clase base se ejecuta.
  - Los campos de la clase derivada son inicializados.
  - El constructor de la clase derivada se ejecuta.

# CLASSES, VISIBILITY, PUBLIC

- Se pueden utilizar TypeScript para controlar la visibilidad de ciertos métodos o propiedades fuera de la clase a la que pertenecen.

# VISIBILIDAD PUBLIC

- La visibilidad pública permite que las propiedades y/o métodos sean accedidos desde cualquier parte.

```
class Saludo {  
    public saludar() {  
        console.log("Saludar!");  
    }  
}  
  
const inst = new Saludo();  
inst.saludar();
```

- Debido a que public es la visibilidad por default, no es necesario declararlo, es decir si no se indica se asume que el método o la propiedad son public.
- Pero si se desea hacer agregar la palabra public se puede hacer por cuestiones como por ejemplo la legibilidad del código.

# CLASSES, VISIBILITY, PROTECTED

- Las propiedades y métodos protected solo son visibles para subclases de la clase en donde son declaradas.

```
class Saludo {  
    protected getDestinatario() {  
        return "amigos";  
    }  
}  
  
class SaludoEspecial extends Saludo {  
    saludar() {  
        console.log(`Hola ${this.getDestinatario()}`); // <- accedemos al método protected  
    }  
}  
  
const saludo: SaludoEspecial = new SaludoEspecial();  
saludo.saludar();  
saludo.getDestinatario(); // <- error, no se tiene acceso de forma publica
```

# HABILITAR LOS MÉTODOS PROTEGIDOS

- Las clases derivadas siguen el contrato definido por la clase padre, pero cuando pueden habilitar un subtipo de la clase base para brindar mas permisos sobre este. Esto incluye el convertir un método protected en public.

```
class Base {  
    protected m = 10;  
}  
  
class Derivada extends Base {  
    m = 15;  
}  
  
const d = new Derivada();  
console.log(d.m);
```



- Tome en cuenta que Derivada desde el momento que extiende Base tiene acceso a m, sin embargo al no incluir protected la propiedad m pasa a ser public y se vuelve accesible desde fuera de la definición de la clase.

# CROSS-HIERARCHY PROTECTED ACCESS

- Diferentes tipos de lenguajes de Languages de POO (Programación Orientados a Objetos) estan en desacuerdo en relación a si es valido o no acceder a elementos protected a través de diferentes clases que extienden la clase base.
- Java por ejemplo considera que esto debe ser posible, mientras tanto otros lenguajes como C#, C++ y TypeScript consideran que esto no debe ser así.

```
class Base {  
    protected x: number = 1;  
}  
  
class Derivada1 extends Base {  
    protected x: number = 5;  
}  
  
class Derivada2 extends Base {  
    imprimirX(c1: Derivada2){ // <-  
        console.log(c1.x);  
    }  
}
```

- Si reemplazamos Derivada2 por Derivada1 esto generará un error ya que esta fuera de su scope.

# CLASSES, VISIBILITY, PRIVATE

- La visibilidad private funciona como protected en el sentido que no permite que el elemento sea accedido fuera de la clase. Sin embargo a diferencia de protected, no permite que las clases que hereden de la clase base lo utilicen, por lo que solo puede ser utilizado por la clase en la cual fue definido.

```
class Base {  
    private x = 0;  
}  
  
class Derivada1 extends Base {  
    imprimirX() {  
        console.log(this.x); // <- esta linea generará un error  
    }  
}  
  
const b = new Base();  
console.log(b.x); // <- esta linea tambien generará un error
```

- En este ejemplo estamos intentando acceder a la propiedad `x`, pero al ser `private` no puede ser accedida fuera de la clase.
- Tampoco es posible acceder a `x` desde una clase que extiende `Base` como es el caso de `Derivada1`.

# CROSS-INSTANCE PRIVATE ACCESS

- Diferentes lenguajes de POO (Programación Orientada a Objetos) tienen visiones diferentes en relación a como deben manejarse los permisos de `private` entre instancias de la misma clase.
- Lenguajes como Java, C#, C++, Swift y PHP lo permiten, mientras que para otros como Ruby no.



```
class A {  
  private x = 10;  
  
  public imprimirX(otra: A) {  
    console.log(otra.x); // <- podemos acceder a una propiedad privada de otra instancia  
  }  
}  
  
const b = new A();  
  
b.imprimirX(new A());
```

# CONSIDERACIONES

- Como otros aspectos de TypeScript, `private` y `protected` solo son analizados durante la revisión del tipado.

# CLASSES, STATIC MEMBERS

- Las clases tienen componentes estáticos. Estos elementos no están asociados con una instancia particular de la clase. Pueden ser accedidos a través del constructor de la clase.

```
1  class MiClase {
2      static x = 10;
3
4      static imprimirX() {
5          // para acceder a una propiedad estática utilizamos this dentro de un método estático
6          console.log(`El valor de x es: ${this.x}`);
7      }
8
9      imprimirX() {
10         // para acceder a una propiedad estática usamos el nombre de la clase dentro de un método de una instancia
11         console.log(`El valor de x en una instancia es: ${MiClase.x}`);
12     }
13 }
14
15 // para acceder a un método lo hacemos directamente desde la clase
16 MiClase.imprimirX();
17
18 // para acceder a una propiedad estática lo hacemos directamente desde la clase
19 console.log(`El valor obtenido de x es: ${MiClase.x}`);
20
21 const miClase = new MiClase();
22 miClase.imprimirX();
```

- Los elementos estáticos pueden ser también public, protected y private.

```
1  class MiClase {  
2      |   private static x = 10;  
3  }  
4  
5  console.log(MiClase.x); // <- esta linea generará un error  
6  // Property 'x' is private and only accessible within class 'MiClase'.  
7
```

- Los métodos estáticos también se heredan.

```
class Base {  
    static saludar() {  
        console.log("Hola mundo");  
    }  
}  
  
class Derivada extends Base {}  
  
Derivada.saludar();
```



# PALABRAS RESERVADAS EN CLASES

- Debido a que las clases son funciones que pueden ser invocadas con new, algunos nombres no pueden ser definidos como static, algunos ejemplos son: name, length y call.

```
class Base {  
  static name = "S!"; // <- esta línea generara un error  
  // Static property 'name' conflicts with built-in property 'Function.name' of constructor function 'Base'  
}
```

# CLASSES, GENERICS

- Las clases al igual que las interfaces, pueden ser genéricas. Cuando una clase es instanceada con el constructor new, su tipo es inferido de la misma forma que cuando se llama a una función.

```
1 class Caja<T> {
2     contenido: T;
3
4     constructor(value: T) {
5         this.contenido = value;
6         console.log(this.contenido);
7     }
8 }
9
10 type Juguete = {
11     nombre: string;
12 };
13
14 const misJuguetes: Juguete[] = [];
15 misJuguetes.push({ nombre: "Pelota" });
16 misJuguetes.push({ nombre: "Consola" });
17
18 const miCajaDeJuguetes: Caja<Juguete[]> = new Caja(misJuguetes);
19
20 type Maquillaje = {
21     nombre: string;
22 };
23
24 const miMaquillaje: Maquillaje[] = [];
25 miMaquillaje.push({ nombre: "Sombras" });
26 miMaquillaje.push({ nombre: "Labial" });
27
28 const miCajaDeMaquillaje: Caja<Maquillaje[]> = new Caja(miMaquillaje);
```

# CLASSES, THIS TYPES

- Dentro de las clases, un tipo especial llamado this hace referencia de forma dinámica a la clase que lo utiliza.

```
class Caja {  
  contenido = "";  
  set(valor: string) {  
    this.contenido = valor;  
    return this;  
  }  
}  
  
const miCaja: Caja = new Caja();  
const valorRetornado = miCaja.set("Joyas");  
console.log(miCaja);  
console.log(valorRetornado);
```

- También es posible utilizar `this` como anotación dentro de los parámetros.

```
class Caja {  
    contenido = "";  
  
    constructor(contenido: string) {  
        this.contenido = contenido;  
    }  
  
    igualQue(otro: this) {  
        return otro.contenido === this.contenido;  
    }  
}  
  
const caja1 = new Caja("joyas");  
const caja2 = new Caja("joyas");  
const caja3 = new Caja("maquillaje");  
  
console.log(caja1.igualQue(caja2));  
console.log(caja1.igualQue(caja3));
```

- Esto es diferente de escribir otro: Caja si se tiene una clase derivada, entonces igualQue solo aceptara instancias derivadas de la misma clase.

```
class Caja {
  contenido = "";

  igualQue(otro: this) {
    return this.contenido === otraInstancia.contenido;
  }
}

class CajaDerivada extends Caja {
  otroContenido = "";
}

const base = new Caja();
const derivada = new CajaDerivada();
derivada.igualQue(base); // <- base no extiende de derivada, esto generará un error
// Argument of type 'Caja' is not assignable to parameter of type 'CajaDerivada'.
```



# CLASSES, PARAMETER PROPERTIES

- TypeScript ofrece un tipo de sintaxis especial para convertir un parámetro de un constructor en una propiedad de una clase con el mismo nombre y valor. Estos son llamados parameter properties (parámetros como propiedades) y son creados utilizando como prefijo al argumento del constructor uno de los modificadores de visibilidad como public, private, protected y readonly.

```
class Video {  
  constructor(  
    public readonly nombre: string,  
    public readonly duracion: number,  
    public readonly formato: "mp4" | "mkv" | "web"  
  ) {}  
}  
  
const miVideo: Video = new Video("vacaciones", 60, "mp4");  
  
console.log(`Mi video de: ${miVideo.nombre}`);  
console.log(`Tiene una duración de: ${miVideo.duracion} segundos`);  
console.log(`Y el formato es: ${miVideo.formato}`);
```

# CLASSES, CLASS EXPRESSIONS

- Las class expressions (clases como expresiones) son muy similares a las class declarations (clases como declaraciones). La única diferencia real es que las class expressions no necesitan tener un nombre, sin embargo podemos referirnos a ellas a través de cualquier identificador al que esten relacionados.

```
const miClase = class<T> {  
  contenido: T;  
  constructor(v: T) {  
    this.contenido = v;  
  }  
};  
  
const miInstancia = new miClase("Un video de 12 minutos");  
  
console.log(`El contenido del video es: ${miInstancia.contenido}`);
```