



TypeScript. Nivel I

junio, 2018



Objetivos del nivel

- Conocer los aspectos básicos del lenguaje TypeScript.
- Aprender a usar módulos.
- Aprender a dibujar en el canvas de HTML5.
- Aplicar el paradigma orientado a objetos en TypeScript.

Prerrequisitos del nivel

- JavaScript Nivel I
- Programación Orientada a Objetos Nivel II

Acerca de este manual

Este manual pertenece al Centro de Asesoramiento y Desarrollo Informático C.A. (CADI F1). Para obtener más información sobre este u otros cursos visite nuestra sitio Web www.cadif1.com, escribanos a la dirección de correo cadi@cadif1.com o visítenos en nuestra sede ubicada en la Av. Pedro León Torres con calle 59, Centro Comercial Sotavento, piso 2 oficina 27, Barquisimeto estado Lara, Venezuela. Tf. 0251-7179247, 0251-4410268.

Las marcas mencionadas en este manual son propiedad de sus respectivos dueños. Copyright 2018. Todos los derechos reservados.

ACADEMIA DE SOFTWARE



Contenido del nivel

Capítulo 1. Introducción a Typescript

- 1.1.- Qué es Typescript?.
- 1.2.- Requisitos.
- 1.3.- Compilación.

Capítulo 2. Conociendo Typescript

- 2.1.- Tipos de Datos.
- 2.2.- Declaración de Variables.
- 2.3.- Asignación.

Capítulo 3. Funciones

- 3.1.- Tipos de Datos de Parámetros.
- 3.2.- Parámetros Opcionales.
- 3.3.- Valor de Retorno.

Capítulo 4. El Canvas. Parte 1

- 4.1.- Concepto.
- 4.2.- Javascript y el Canvas.
- 4.3.- Dibujar en el Canvas.



Capítulo 5. El Canvas. Parte 2

- 5.1.- Pintar Círculos.
- 5.2.- Pintar Cuadros.
- 5.3.- Escribir Texto.

Capítulo 6. Enums

- 6.1.- Concepto.
- 6.2.- Declaración de Enums.
- 6.3.- Uso de Enums.

Capítulo 7. Interfaces

- 7.1.- Concepto.
- 7.2.- Declaración de Interfaces.
- 7.3.- Uso de Interfaces.

Capítulo 8. Clases

- 8.1.- Definición.
- 8.2.- Instanciación.
- 8.3.- Métodos.

Capítulo 9. Modificadores de Acceso

- 9.1.- Public.
- 9.2.- Private.
- 9.3.- Readonly.

Capítulo 10. Gettes y Setter

- 10.1.- Get.
- 10.2.- Set.
- 10.3.- Uso.



Capítulo 11. Constructores

- 11.1.- Definición.
- 11.2.- Uso del constructor.

Capítulo 12. Herencia

- 12.1.- Sub Clases.
- 12.2.- Protected.
- 12.3.- Super.

Capítulo 13. Clases Abstractas

- 13.1.- Sobre Escritura de Métodos.
- 13.2.- Clases Abstractas.
- 13.3.- Métodos Abstractos.

Capítulo 14. Métodos y Atributos Estáticos

- 14.1.- Métodos Estáticos.
- 14.2.- Atributos Estáticos.

Capítulo 15. Módulos

- 15.1.- Concepto.
- 15.2.- Exportar.
- 15.3.- Importar.

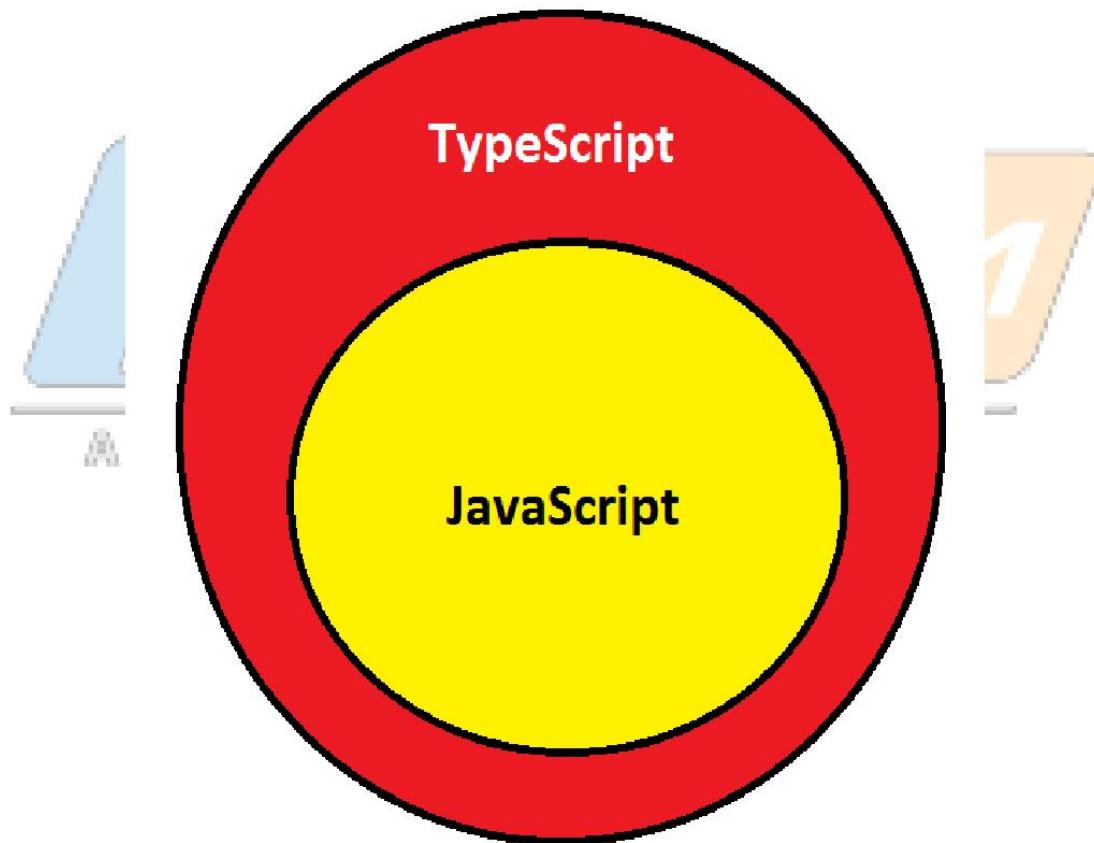


ACADEMIA DE SOFTWARE

Capítulo 1. INTRODUCCIÓN A TYPESCRIPT

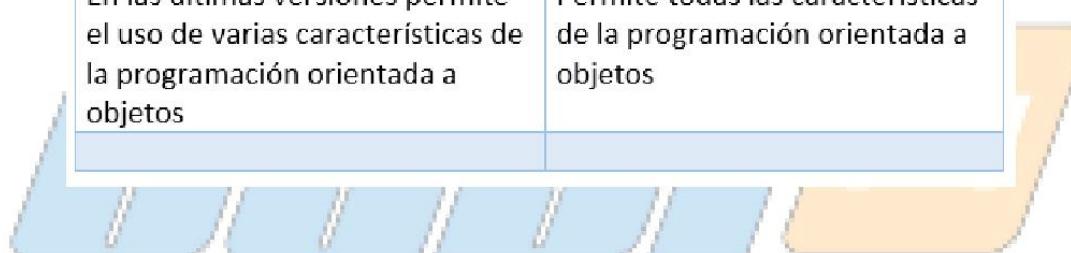
1.1.- Qué es Typescript ?

TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un súper conjunto de JavaScript, que esencialmente añade tipado estático y objetos basados en clases. TypeScript extiende la sintaxis de JavaScript, por tanto cualquier código JavaScript existente debería funcionar sin problemas.



TypeScript es un súper conjunto de JavaScript, por lo tanto, todas las instrucciones de JavaScript son válidas en TypeScript. Aun así, TypeScript tiene algunas diferencias con JavaScript, tales como:

JavaScript	TypeScript
Es un lenguaje interpretado por el navegador	Es un lenguaje compilado
Al declarar una variable no se debe indicar el tipo de dato	Al declarar una variable se puede indicar el tipo de dato
No se verifican los tipos de datos	Verificación de tipos de datos
Las funciones pueden retornar valores de cualquier tipo de dato	El valor de retorno de una función debe coincidir con el tipo de dato de su definición
No se verifica tipo de dato de parámetros al llamar funciones	Verificación de tipos de datos en el paso de parámetros de funciones
En las últimas versiones permite el uso de varias características de la programación orientada a objetos	Permite todas las características de la programación orientada a objetos



1.2.- Requisitos

ACADEMIA DE SOFTWARE

Para usar TypeScript es necesario instalar Node.js, que es el responsable de ejecutar el compilador, que convierte el código TypeScript a JavaScript. El instalador de Node.js se puede encontrar en la página oficial nodejs.org. Al instalar Node.js se instala también el NodePackage Manager (NPM), a través del cual se instala TypeScript.



Para verificar si está instalado Node.js se debe abrir una ventana de consola y ejecutar el comando `node --version`. Si se instaló correctamente debe aparecer la versión de Node.js que está instalada. Luego, debe ejecutarse la instrucción para instalar TypeScript: `npm install -g typescript`.

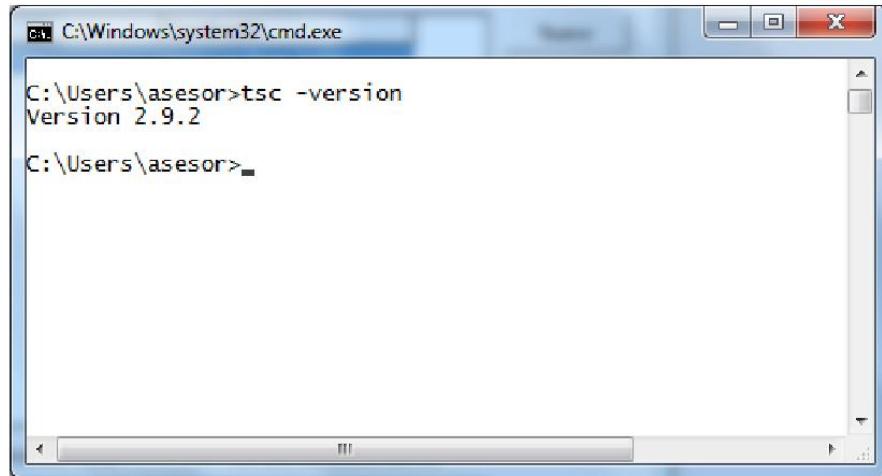
```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\asesor>node --version
v8.11.3

C:\Users\asesor>npm install -g typescript
C:\Users\asesor\AppData\Roaming\npm\tsc -> C:\Users\asesor\AppData\Roaming\npm\node_modules\typescript\bin\tsc
C:\Users\asesor\AppData\Roaming\npm\tsserver -> C:\Users\asesor\AppData\Roaming\npm\node_modules\typescript\bin\tsserver
+ typescript@2.9.2
updated 1 package in 9.525s

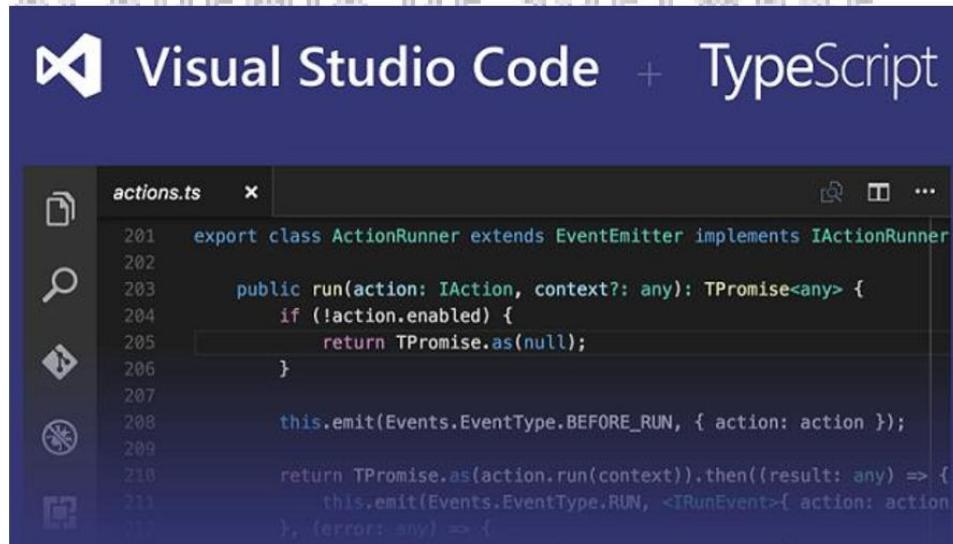
C:\Users\asesor>
```

Si la instalación se realiza exitosamente, se muestra la versión que se instala de TypeScript. Para comprobar la instalación de TypeScript se ejecuta el comando: `tsc -version`, como se muestra en la siguiente imagen:

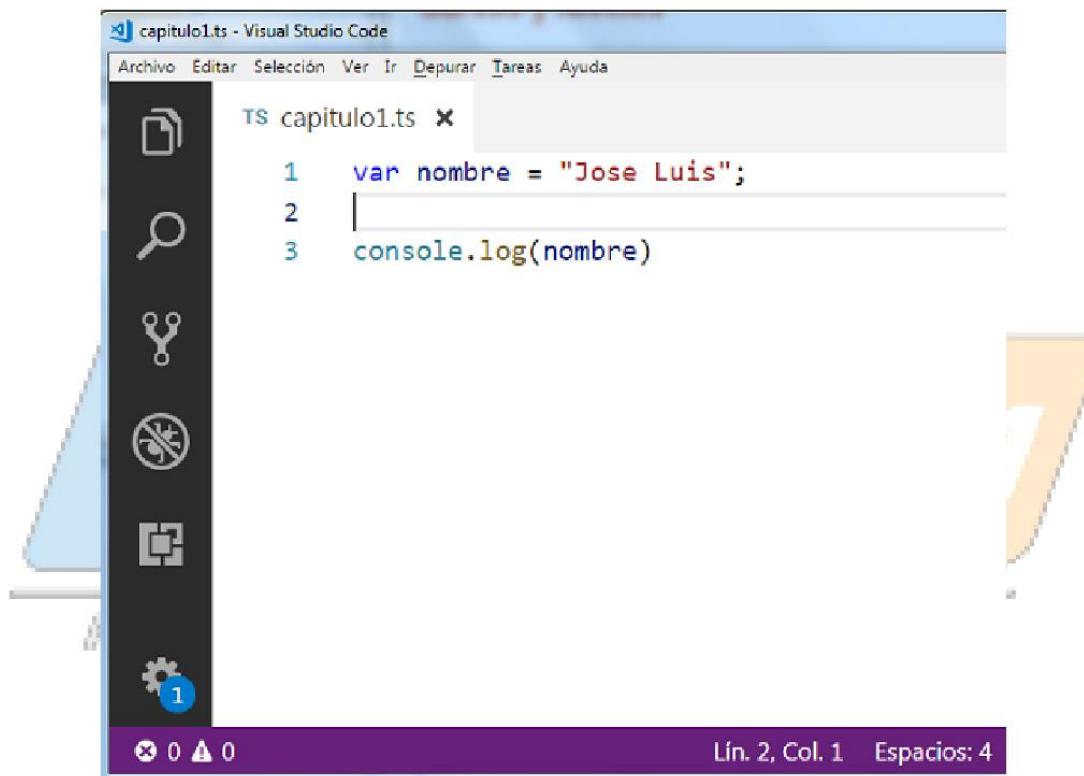


1.3.- Compilación

El código TypeScript puede escribirse en cualquier editor de texto así como se escribe el código JavaScript. Sin embargo, el código TypeScript al tener que compilarse, se ha hecho posible que algunos editores de texto como SublimeText y entornos de desarrollo como Visual Studio Code, implementen verificación sintáctica mientras se escribe el código, lo que supone una gran ventaja para el programador.



El compilador de TypeScript (también llamado transpilador) se llama tsc y se ejecuta en la línea de comandos. Los archivos TypeScript se deben guardar con la extensión .ts. Al compilar exitosamente un archivo .ts el compilador genera un archivo .js con las instrucciones equivalentes. Una de las ventajas de usar TypeScript es que el compilador mostrará la existencia de errores antes de la ejecución del código en el navegador.



```
TS capitulo1.ts ×
1  var nombre = "Jose Luis";
2
3  console.log(nombre)
```

Lín. 2, Col. 1 Espacios: 4

La forma general de compilar es la siguiente: tscfuente.ts. Esta instrucción debe ejecutarse en la línea de comandos. Visual Studio Code tiene incorporada una ventana de consola que facilita el trabajo:

The screenshot shows the Visual Studio Code interface. On the left, there are two tabs: 'TS capitulo1.ts' and 'JS capitulo1.js'. The 'capitulo1.ts' tab is active, showing the following code:

```
1 var nombre="Jose Luis R."
2
3 console.log(nombre)
```

The 'capitulo1.js' tab shows the generated JavaScript code:

```
1 var nombre = "Jose Luis R.";
2 console.log(nombre);
3
```

Below the tabs, there is a terminal window titled 'cmd' with the following text:

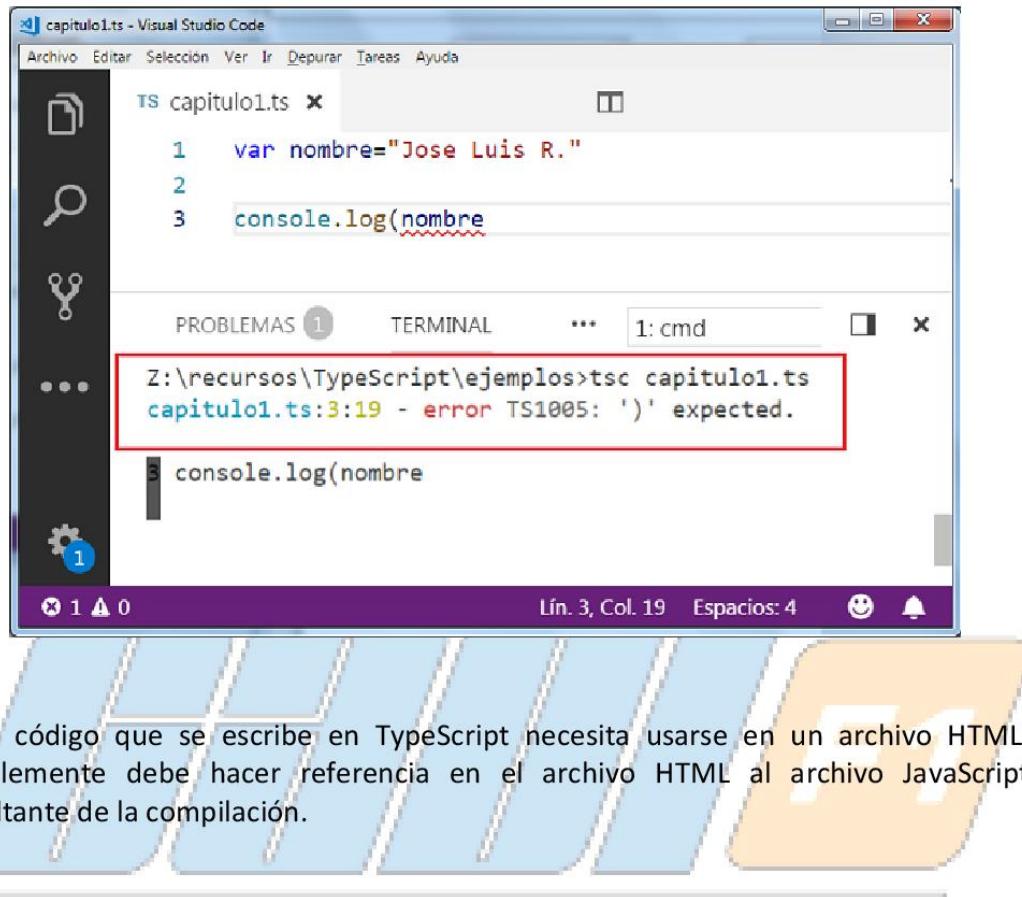
```
Z:\recursos\TypeScript\ejemplos>tsc capitulo1.ts Compilación
Z:\recursos\TypeScript\ejemplos>
```

The status bar at the bottom indicates: Lín. 1, Col. 25 Espacios: 4 UTF-8 CRLF TypeScript 2.9.1.

Se puede compilar manualmente el código cada vez que se hace un cambio en los archivos fuentes, o se puede iniciar el modo "watch". El modo Watch hace que el compilador esté constantemente observando si hay cambios en el archivo fuente y se guardan, en cuyo caso realiza la compilación automáticamente. El modo "watch" se puede activar ejecutando el mismo comando de compilación agregando el parámetro -w. Por ejemplo:

tsfuente.ts -w

Si existe algún error de sintaxis es mostrado tanto por el editor del IDE como durante la compilación:



The screenshot shows a Visual Studio Code window with the title "capítulo1.ts - Visual Studio Code". The code editor contains the following TypeScript code:

```
1 var nombre="Jose Luis R."
2
3 console.log(nombre)
```

The terminal tab shows the command: `Z:\recursos\TypeScript\ejemplos>tsc capítulo1.ts`. Below it, an error message is displayed: `capítulo1.ts:3:19 - error TS1005: ')' expected.`. This error is highlighted with a red box. The status bar at the bottom indicates "Lín. 3, Col. 19 Espacios: 4".

Below the code editor, there is a decorative graphic featuring blue and orange abstract shapes.

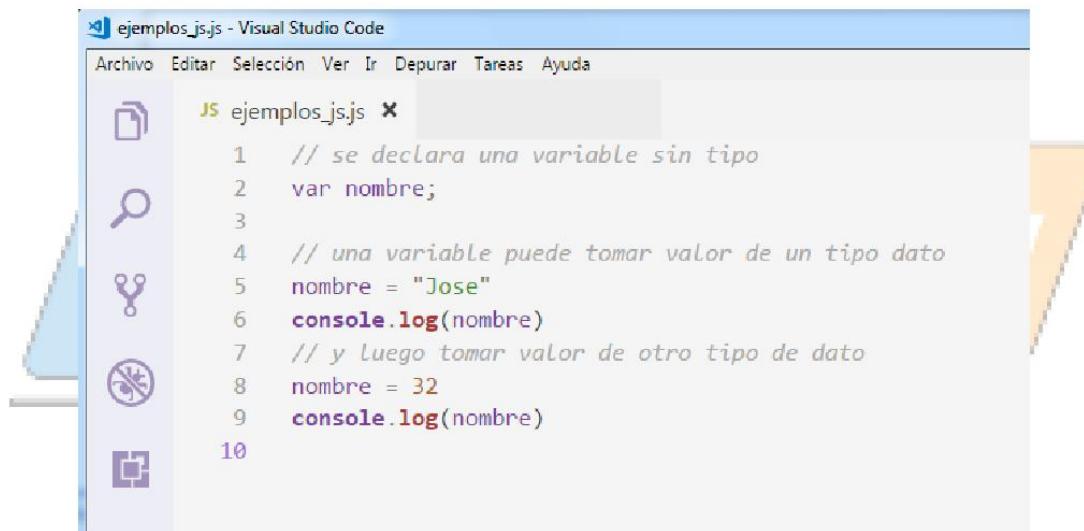
Si el código que se escribe en TypeScript necesita usarse en un archivo HTML, simplemente debe hacer referencia en el archivo HTML al archivo JavaScript resultante de la compilación.

ACADEMIA DE SOFTWARE

Capítulo 2. CONOCIENDO TYPESCRIPT

2.1.- Tipos de Datos

JavaScript maneja el concepto de tipo de dato de una forma muy flexible. En cambio, una de las características más resaltantes de TypeScript es que permite establecer tipos de datos a las variables al momento de declararlas, de tal forma que el compilador verificará si una variable declarada de cierto tipo de dato recibe valores que correspondan a ese tipo. La siguiente imagen muestra código de ejemplo de JavaScript:



```
ejemplos_js.js - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
JS ejemplos_js.js ✘
1 // se declara una variable sin tipo
2 var nombre;
3
4 // una variable puede tomar valor de un tipo dato
5 nombre = "Jose"
6 console.log(nombre)
7 // y luego tomar valor de otro tipo de dato
8 nombre = 32
9 console.log(nombre)
10
```

En JavaScript existen los tipos de datos implícitos al momento de asignar valor a las variables. Los tipos de datos de JavaScript son: números, arreglos, objetos, booleanos y string. En TypeScript, se agregaron nuevos tipos de datos, que son:

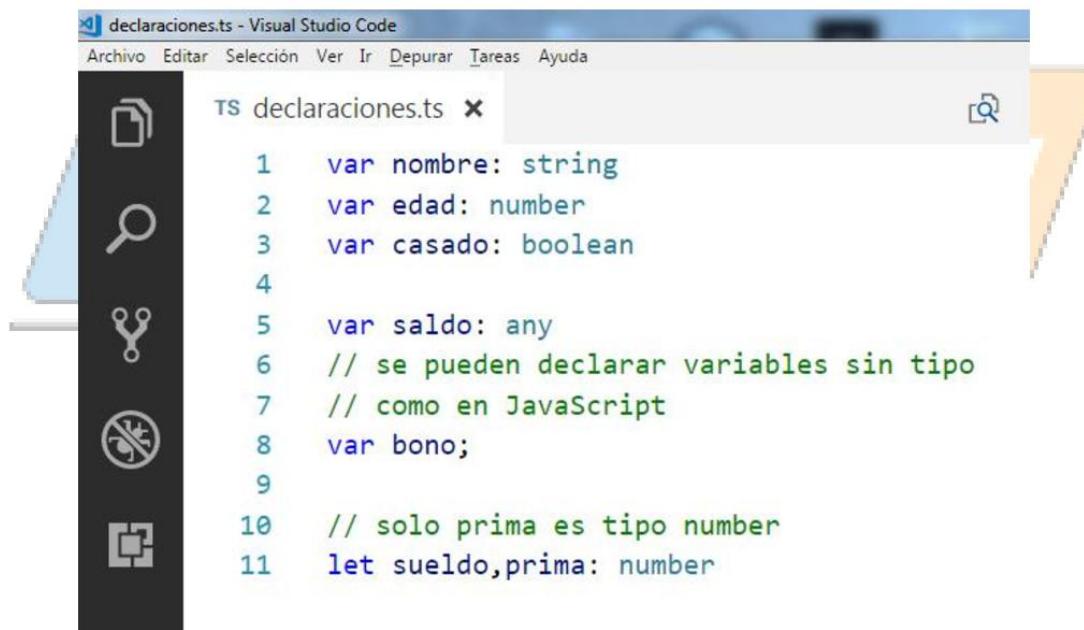
- boolean: true o false.
- number: números de cualquier rango.
- string: cadenas de caracteres.
- array: arreglos
- Any: se utiliza cuando no se
- object: para objetos.
- enum: enumerados (se explicará más adelante).
- tuplas: mapas.

2.2.- Declaración de Variables

En TypeScript es obligatorio declarar las variables. Se pueden declarar usando las mismas palabras reservadas de JavaScript (var, let y const) sin tipo de dato, pero se perdería una de las características más distintivas del lenguaje. Las variables se declaran en forma general:

```
Var variable:tipo  
Let variable:tipo
```

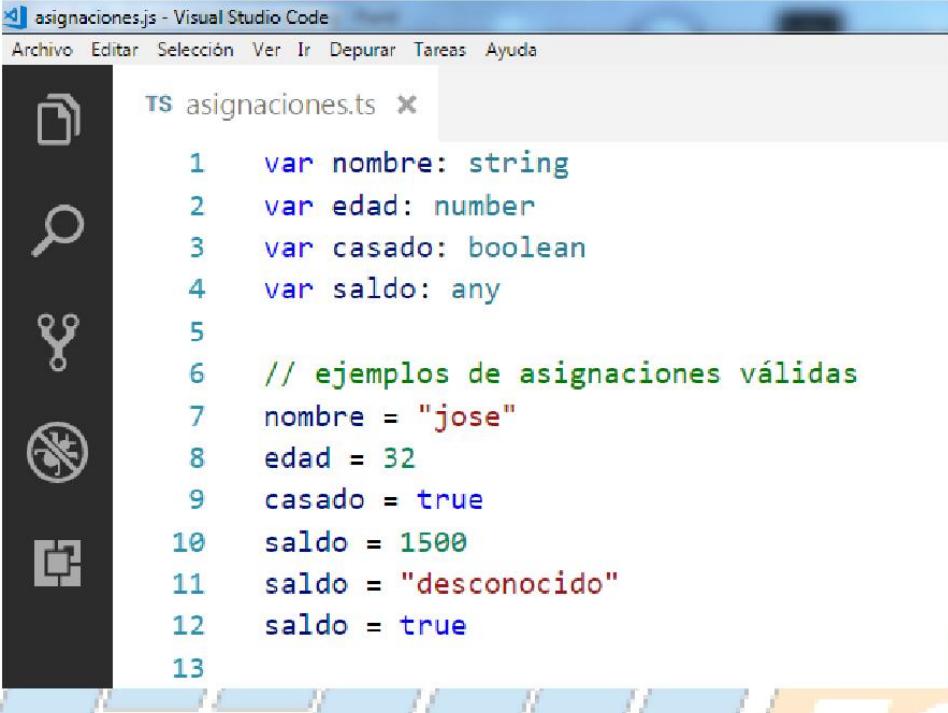
Donde "tipo" es uno de los tipos de datos definidos previamente. Ejemplos de declaración de variables son los siguientes:



```
declaraciones.ts - Visual Studio Code  
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda  
TS declaraciones.ts x 🔍  
1 var nombre: string  
2 var edad: number  
3 var casado: boolean  
4  
5 var saldo: any  
6 // se pueden declarar variables sin tipo  
7 // como en JavaScript  
8 var bono;  
9  
10 // solo prima es tipo number  
11 let sueldo,prima: number
```

2.3.- Asignación

Al declarar variables con un tipo de dato particular, al hacer una asignación, el compilador verifica el tipo de dato del valor asignado. Si no son compatibles, se produce un error de compilación. Por ejemplo:



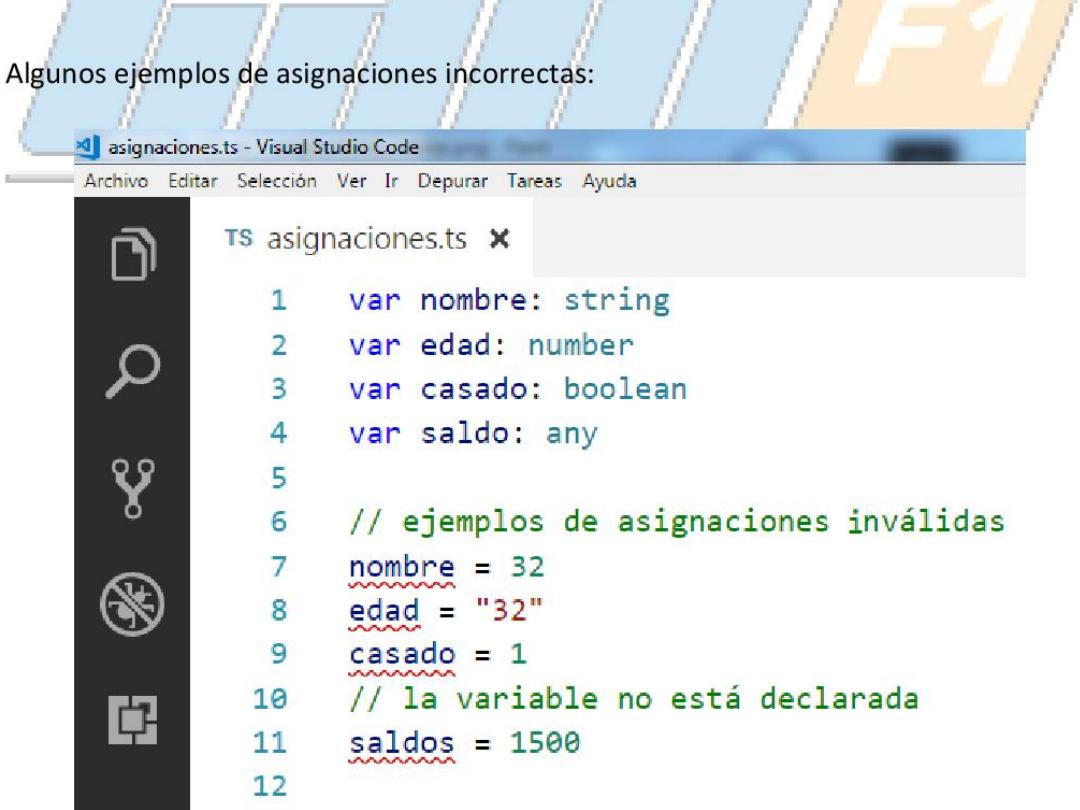
asignaciones.ts - Visual Studio Code

Archivo Editar Selección Ver Ir Depurar Tareas Ayuda

TS asignaciones.ts ×

```
1 var nombre: string
2 var edad: number
3 var casado: boolean
4 var saldo: any
5
6 // ejemplos de asignaciones válidas
7 nombre = "jose"
8 edad = 32
9 casado = true
10 saldo = 1500
11 saldo = "desconocido"
12 saldo = true
13
```

Algunos ejemplos de asignaciones incorrectas:



asignaciones.ts - Visual Studio Code

Archivo Editar Selección Ver Ir Depurar Tareas Ayuda

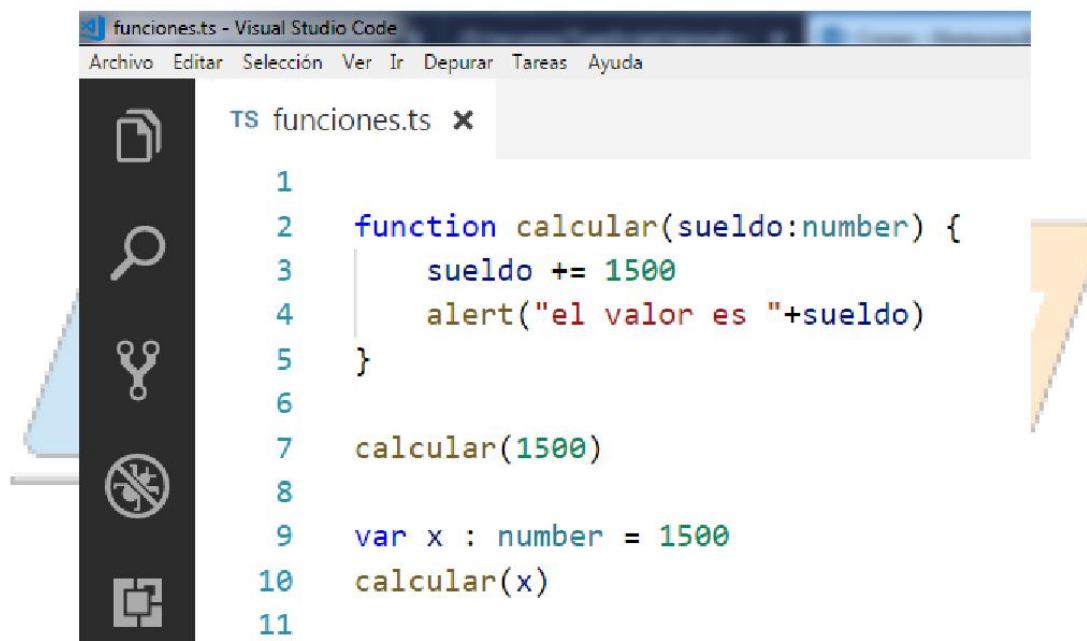
TS asignaciones.ts ×

```
1 var nombre: string
2 var edad: number
3 var casado: boolean
4 var saldo: any
5
6 // ejemplos de asignaciones inválidas
7 nombre = 32
8 edad = "32"
9 casado = 1
10 // la variable no está declarada
11 saldos = 1500
12
```

Capítulo 3. FUNCIONES

3.1.- Tipos de Datos de Parámetros

En TypeScript los parámetros de una función pueden tener un tipo de dato, lo que obliga que al usar la función se deben cumplir las reglas de coincidencia entre parámetros formales y actuales: misma cantidad, mismo tipo y mismo orden. Un ejemplo de una función con parámetros con tipos de datos:



```
funciones.ts - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
TS funciones.ts ×
1
2  function calcular(sueldo:number) {
3      sueldo += 1500
4      alert("el valor es "+sueldo)
5  }
6
7  calcular(1500)
8
9  var x : number = 1500
10 calcular(x)
11
```

The screenshot shows a Visual Studio Code window with a dark theme. On the left is a sidebar with icons for file operations, search, and other development tools. The main editor area has a title bar 'funciones.ts - Visual Studio Code' and a menu bar with Spanish options: Archivo, Editar, Selección, Ver, Ir, Depurar, Tareas, Ayuda. The code editor contains the following TypeScript code:

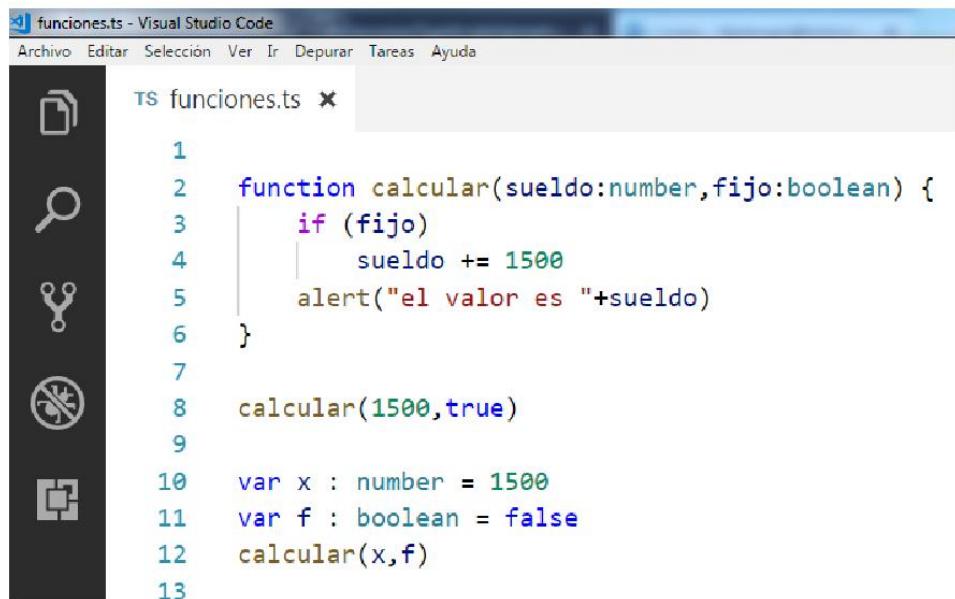
```
function calcular(sueldo:number) {
    sueldo += 1500
    alert("el valor es "+sueldo)
}

calcular(1500)

var x : number = 1500
calcular(x)
```

Line numbers 1 through 11 are visible on the left.

Cada parámetro de la función debe tener un tipo de dato



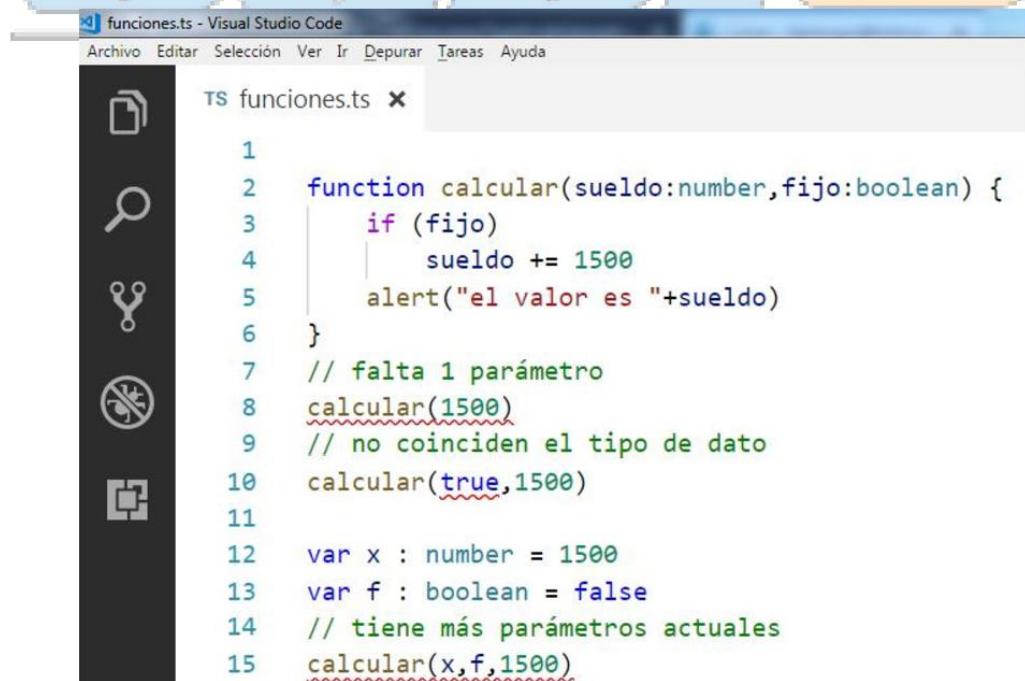
```

funciones.ts - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
TS funciones.ts ✘

1
2   function calcular(sueldo:number,fijo:boolean) {
3     if (fijo)
4       sueldo += 1500
5       alert("el valor es "+sueldo)
6   }
7
8   calcular(1500,true)
9
10  var x : number = 1500
11  var f : boolean = false
12  calcular(x,f)
13

```

El compilador verifica el tipo de dato y la cantidad de parámetros actuales (en el llamado) con respecto a los parámetros formales (la definición de la función). Los siguientes son ejemplos de errores al usar funciones:



```

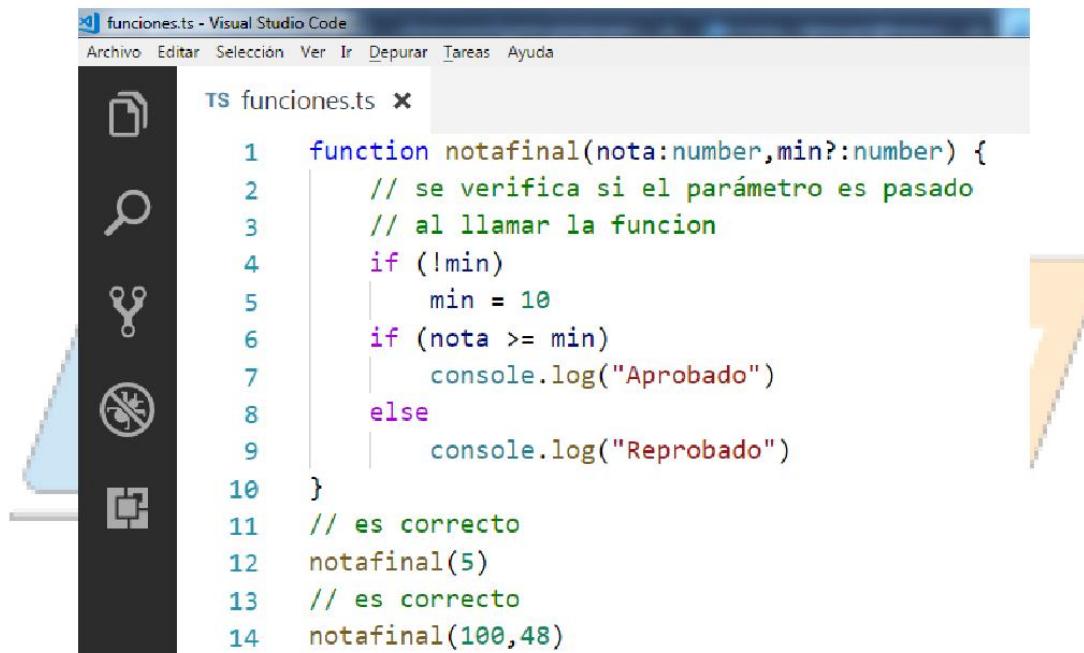
funciones.ts - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
TS funciones.ts ✘

1
2   function calcular(sueldo:number,fijo:boolean) {
3     if (fijo)
4       sueldo += 1500
5       alert("el valor es "+sueldo)
6   }
7   // falta 1 parámetro
8   calcular(1500)
9   // no coinciden el tipo de dato
10  calcular(true,1500)
11
12  var x : number = 1500
13  var f : boolean = false
14  // tiene más parámetros actuales
15  calcular(x,f,1500)

```

3.2.- Parámetros Opcionales

En TypeScript los parámetros de una función son todos obligatorios. Sin embargo, se pueden definir parámetros opcionales. Para lograrlo, debe colocarse el símbolo "?" justo después del nombre del parámetro. Cuando se llama la función y no envía el parámetro, toma el valor "undefined" :



```
funciones.ts - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
TS funciones.ts ✘
1  function notafinal(nota:number,min?:number) {
2      // se verifica si el parámetro es pasado
3      // al llamar la función
4      if (!min)
5          min = 10
6      if (nota >= min)
7          console.log("Aprobado")
8      else
9          console.log("Reprobado")
10 }
11 // es correcto
12 notafinal(5)
13 // es correcto
14 notafinal(100,48)
```

En el ejemplo anterior, el llamado realizado a la función en la línea 12 tiene un sólo parámetro, por lo tanto, el parámetro "min" en la función toma el valor "undefined". En la línea 14 se hace el llamado de la función pasando los 2 parámetros actuales, en cuyo caso, el parámetro "min" toma el valor 48.

Otra forma de usar parámetros opcionales es asignar un valor por defecto al parámetro. En caso de no pasar el parámetro en el llamado, se asume que tiene el valor asignado por defecto.

Los parámetrosopcionales deben estar al final del listado de parámetros formales.

```
funciones.ts - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
TS funciones.ts ×
1
2     function notafinal(nota:number,min=10) {
3
4         if (nota >= min)
5             console.log("Aprobado")
6         else
7             console.log("Reprobado")
8     }
9 // es correcto
10 notafinal(5)
11 // es correcto
12 notafinal(100,48)
```

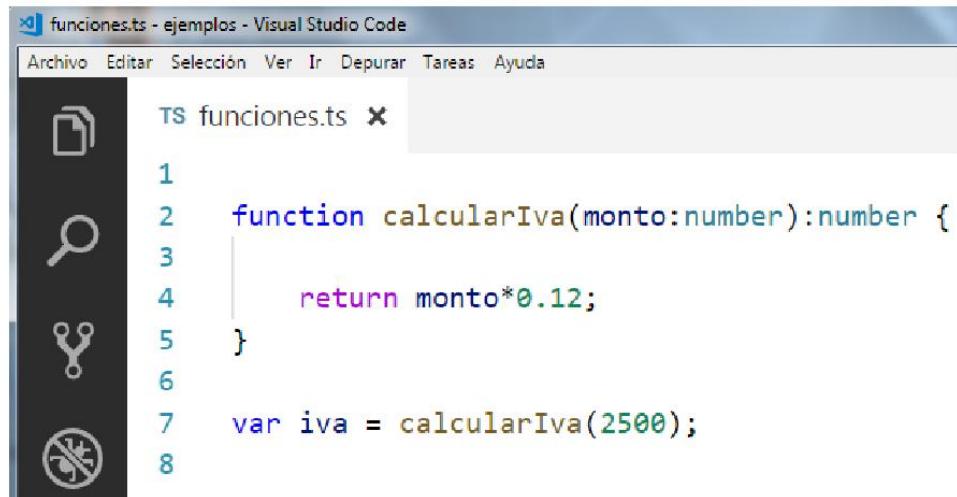
3.3.- Valor de Retorno

A una función se le puede especificar el tipo de dato de lo que retornará. La forma general de hacerlo es:

ACADEMIA DE SOFTWARE

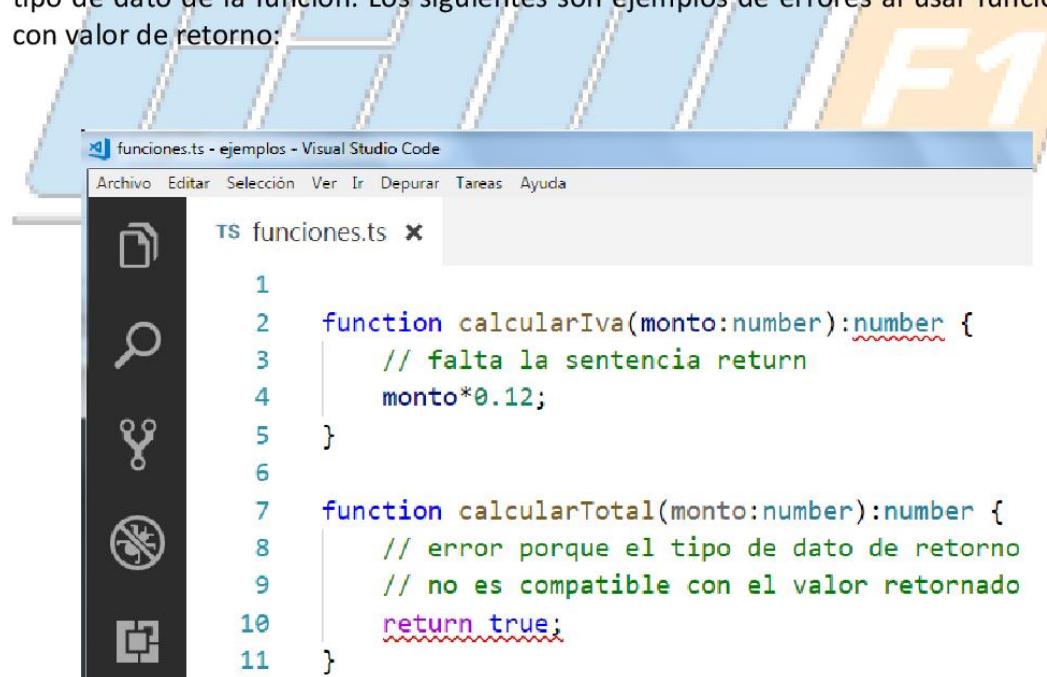
```
function nombre():tipodato {
    return valor;
}
```

Por ejemplo:



```
funciones.ts - ejemplos - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
TS funciones.ts ×
1
2     function calcularIva(monto:number):number {
3
4         return monto*0.12;
5     }
6
7     var iva = calcularIva(2500);
8
```

Si una función tiene un tipo de dato de retorno debe tener obligatoriamente la sentencia `return`. Además, el tipo de dato del valor de retorno debe coincidir con el tipo de dato de la función. Los siguientes son ejemplos de errores al usar funciones con valor de retorno:



```
funciones.ts - ejemplos - Visual Studio Code
Archivo Editar Selección Ver Ir Depurar Tareas Ayuda
TS funciones.ts ×
1
2     function calcularIva(monto:number):number {
3         // falta la sentencia return
4         monto*0.12;
5     }
6
7     function calcularTotal(monto:number):number {
8         // error porque el tipo de dato de retorno
9         // no es compatible con el valor retornado
10        return true;
11    }
```

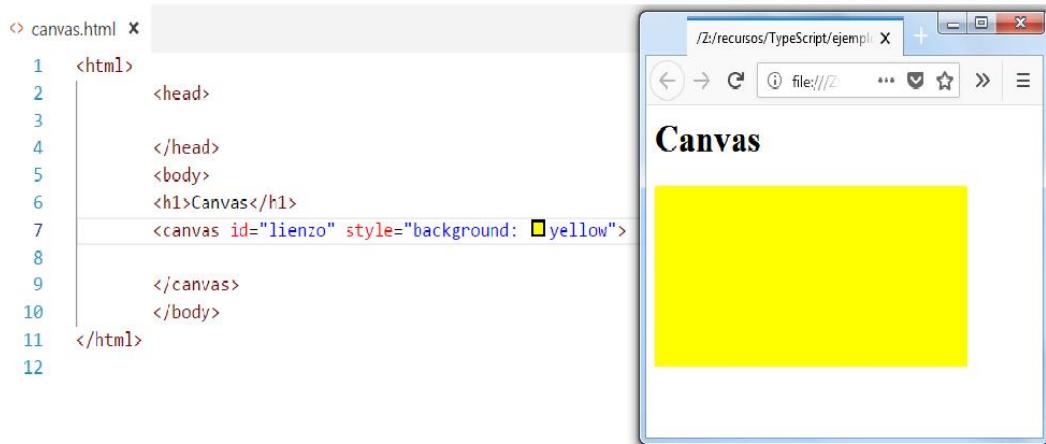
Capítulo 4. EL CANVAS. PARTE 1

4.1.- Concepto

El canvas es un elemento HTML, incorporado en HTML 5, que permite dibujar gráficos usando scripts. Puede ser usado para dibujar gráficos, realizar composición de fotos o simples animaciones en el documento HTML.



Para usar el elemento <canvas> se necesita saber y entender los aspectos básicos del HTML y JavaScript. El elemento <canvas> no está soportado en navegadores antiguos, pero está soportado en la mayoría de las versiones más recientes de los navegadores. El siguiente ejemplo muestra el uso de la etiqueta canvas:



El tamaño por defecto del canvas es 300px * 150px [ancho (width) * alto (height)], aunque se puede especificar valores distintos en las propiedades de la etiqueta. Para pintar en el canvas se utiliza JavaScript y es necesario que tenga un id único para poder buscarlo y manipularlo.

En el interior de la etiqueta canvas se pueden colocar otros elementos HTML, como imágenes o texto. Este contenido es alternativo, en caso de que el navegador no tenga compatibilidad con canvas, aparecerá este contenido, de lo contrario, no aparecerá.

4.2.- Javascript y el Canvas

El canvas por sí mismo no tiene métodos para pintar. Para empezar a dibujar primero se debe buscar el elemento

```

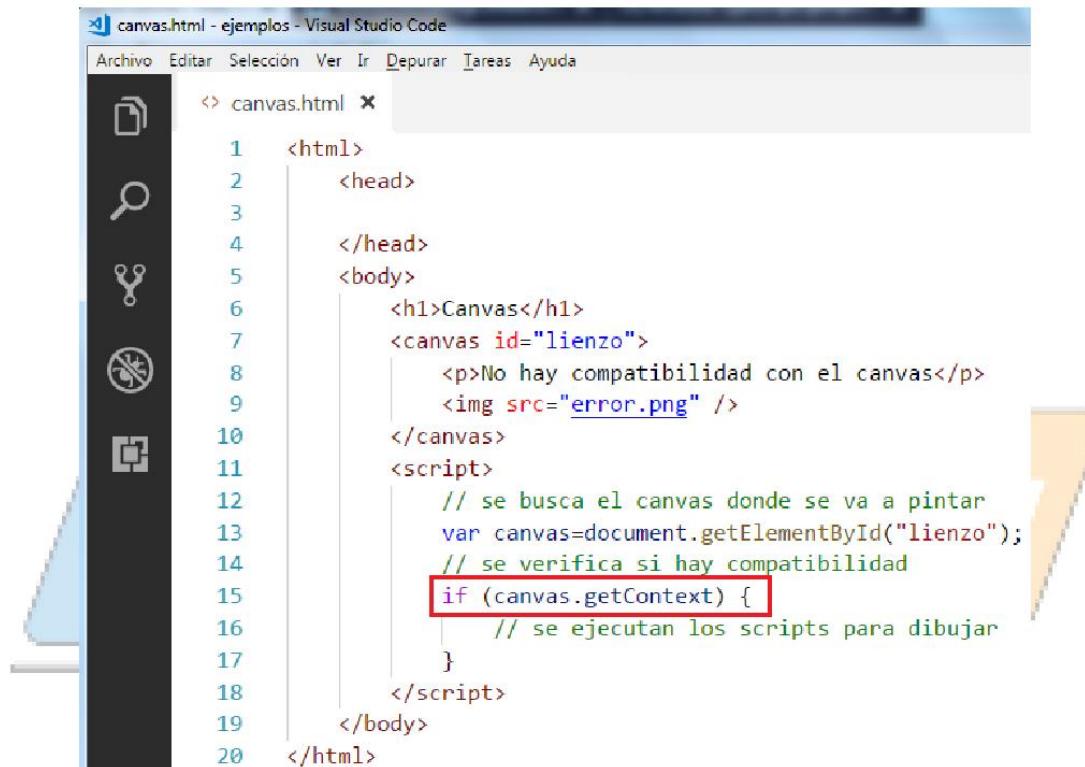
<h1>Canvas</h1>
<canvas id="lienzo">

</canvas>
<script>
    // se busca el canvas donde se va a pintar
    var canvas=document.getElementById("lienzo");

</script>

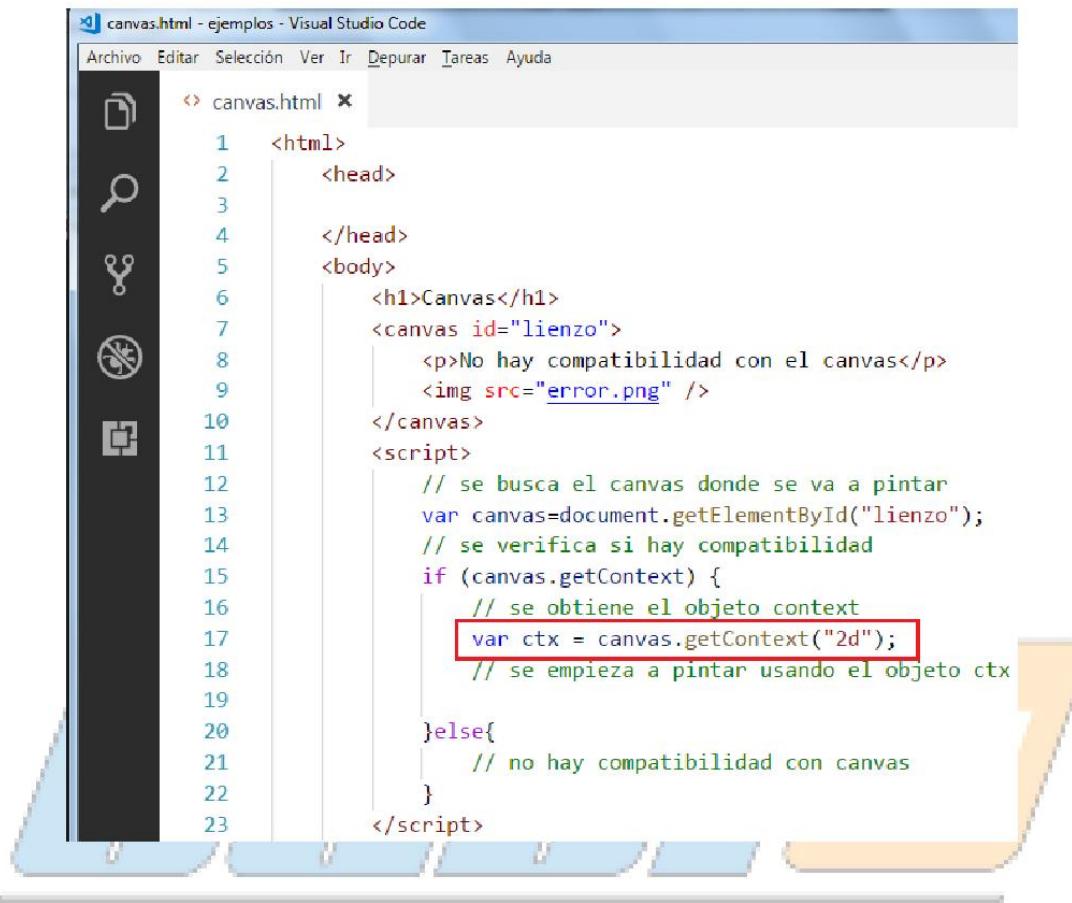
```

Algunos navegadores no tienen compatibilidad con el canvas, por lo tanto, lo primero que se debe hacer es verificar si hay compatibilidad, de lo contrario, no se podrá dibujar en éste. La forma de verificar si hay compatibilidad es la siguiente:



```
1 <html>
2   <head>
3
4   </head>
5   <body>
6     <h1>Canvas</h1>
7     <canvas id="lienzo">
8       <p>No hay compatibilidad con el canvas</p>
9       
10    </canvas>
11    <script>
12      // se busca el canvas donde se va a pintar
13      var canvas=document.getElementById("lienzo");
14      // se verifica si hay compatibilidad
15      if (canvas.getContext) {
16        // se ejecutan los scripts para dibujar
17      }
18    </script>
19  </body>
20 </html>
```

Si hay compatibilidad, el método `getContext()` retornará true, en caso contrario, retornará falso. Utilizando el método pero con el parámetro "2d" de la siguiente forma: `getContext("2d")`, se accede al objeto que permite pintar en el canvas, como se muestra en el siguiente ejemplo:



```
1 <html>
2   <head>
3
4   </head>
5   <body>
6     <h1>Canvas</h1>
7     <canvas id="lienzo">
8       <p>No hay compatibilidad con el canvas</p>
9       
10    </canvas>
11    <script>
12      // se busca el canvas donde se va a pintar
13      var canvas=document.getElementById("lienzo");
14      // se verifica si hay compatibilidad
15      if (canvas.getContext) {
16        // se obtiene el objeto context
17        var ctx = canvas.getContext("2d");
18        // se empieza a pintar usando el objeto ctx
19
20      }else{
21        // no hay compatibilidad con canvas
22      }
23    </script>
```

El objeto "context" permite acceder al contexto de dibujo (rederedcontext). Este objeto posee una gran lista de métodos que se usan para pintar, entre estos:

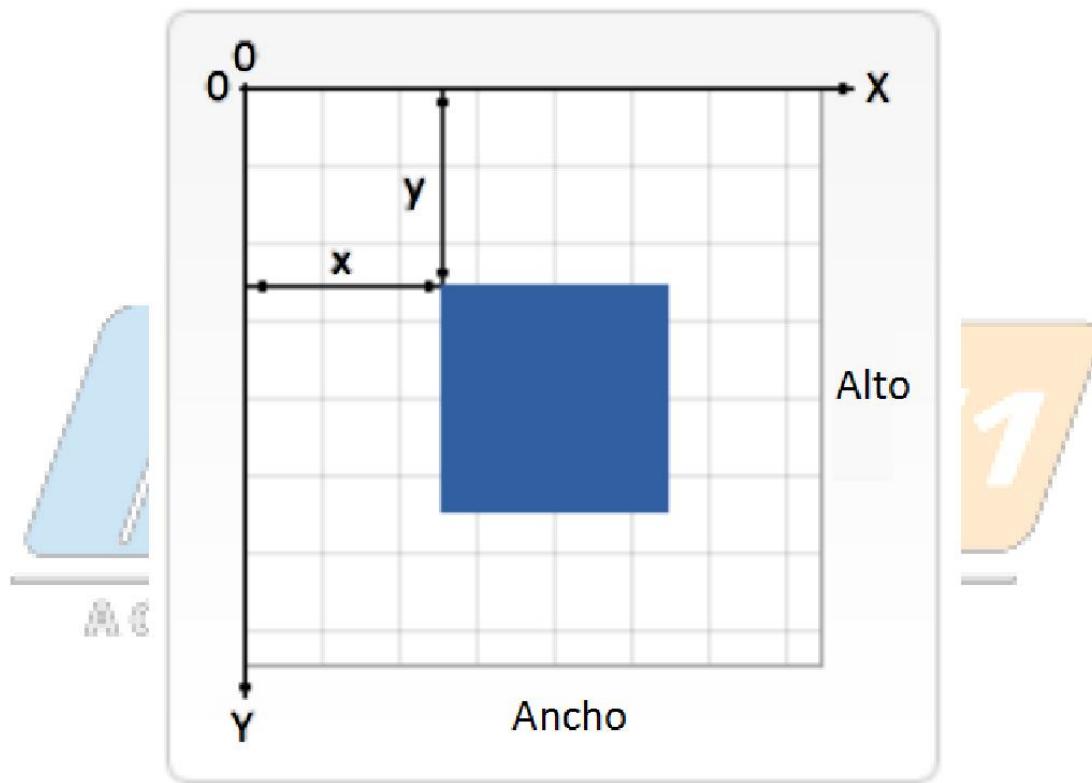
- para establecer colores y tipos de coloreado.
- para pintar línea.
- para pintar rectángulos.
- para pintar trazos.
- para transformaciones.
- para texto.
- para imágenes.
- para manipular pixeles.

4.3.- Dibujar en el Canvas

El canvas genera un área para pintar en el navegador. Por defecto, el canvas está limpio (vacío) para empezar a dibujar. Lo primero que se debe entender antes de

pintar es que el área del canvas se maneja similar un plano cartesiano, donde hay posiciones determinadas por un par de valores (X,Y).

Estos de valores de X y Y pueden ir desde 0 hasta un valor menor o igual al ancho o el alto del canvas respectivamente. Si se asignan valores menores a 0 o mayores al ancho o el alto, el dibujo puede que no se visualice en el canvas.

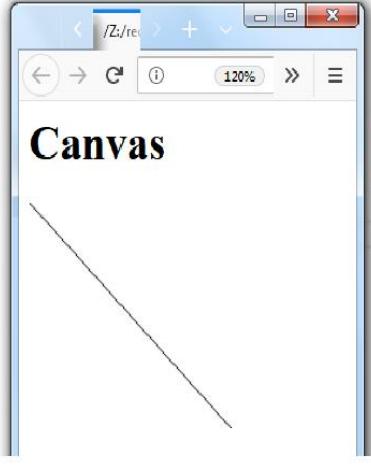


Como se visualiza en la imagen anterior, el punto (0,0) se encuentra en la esquina superior derecha del canvas. Todos los dibujos se deben realizar en base a esa referencia. El canvas permite dibujos de formas primitivas: rectángulos o trazos de rutas (paths).

Un Path (trazo), es un conjunto de puntos conectadas entre éstas con líneas, de diferentes colores y diferentes tipos de trazados. Para pintar un trazo, se deben realizar 6 pasos:

1. iniciar o abrir el path (beginPath).
2. mover el punto inicial a la ubicación donde se desee iniciar el trazo (moveTo).
3. pintar una línea hasta el punto deseado (lineTo) (se pueden hacer varias).
4. cerrar el path para trazar una línea hasta el punto de inicio del path (opcional) (closePath).
5. pintar el path (stroke).
6. llenar el path (fill) (opcional).

Por ejemplo, para pintar una simple línea desde un punto de origen hasta cierto punto destino se utiliza el método `moveTo(x,y)` y luego `lineTo(x,y)`, por ejemplo, para pintar una línea desde el punto (0,0) hasta el punto (200,200):



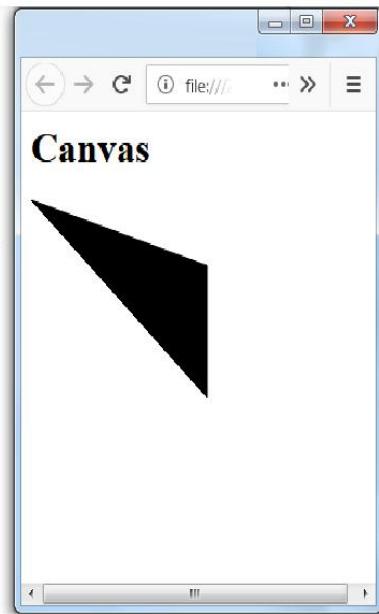
```

5   <body>
6     <h1>Canvas</h1>
7     <canvas id="lienzo">
8     </canvas>
9     <script>
10    // se busca el canvas donde se va a pintar
11    var canvas=document.getElementById("lienzo");
12    if (canvas.getContext) {
13      // se obtiene el objeto context
14      var ctx = canvas.getContext("2d");
15      ctx.beginPath();
16      ctx.moveTo(0, 0);
17      ctx.lineTo(200,200);
18      ctx.stroke();
19    }
20  </script>
21 </body>
22 </html>

```

Si se necesita crear un trazo cerrado y relleno se utiliza `closePath` y `fill`:

```
4 </head>
5 <body>
6   <h1>Canvas</h1>
7   <canvas id="lienzo">
8   </canvas>
9   <script>
10    // se busca el canvas donde se va a pintar
11    var canvas=document.getElementById("lienzo");
12    if (canvas.getContext) {
13      // se obtiene el objeto context
14      var ctx = canvas.getContext("2d");
15      ctx.beginPath();
16      ctx.moveTo(0, 0);
17      ctx.lineTo(150,150);
18      ctx.lineTo(150,50);
19      ctx.closePath();
20      ctx.fill();
21      ctx.stroke();
22    }
23 </script>
```



ACADEMIA DE SOFTWARE

Capítulo 5. EL CANVAS. PARTE 2

5.1.- Pintar Cuadros

Como se explicó anteriormente, en el canvas se pueden pintar "paths" (trazos de líneas) o cuadros. Para pintar un cuadrado se puede hacer un path de 4 líneas, aunque usar la función especialmente diseñada para pintar cuadros es más simple. Se puede pintar un cuadro vacío o un trazo relleno. Existen 3 métodos para pintar cuadros:

- fillRect: para pintar un cuadro relleno.
- rect: para pintar un cuadro sin relleno. Requiere el uso del método stroke.
- strokeRect: igual que el anterior.
- clearRect: borra una sección rectangular en el dibujo.

Todos los métodos reciben 4 parámetros:

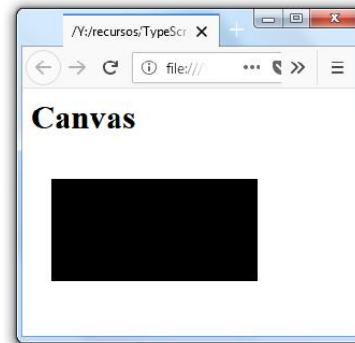
- x: posición X de la esquina superior izquierda del cuadro.
- y: posición Y de la esquina superior izquierda del cuadro.
- alto: alto del cuadro.
- ancho: ancho del cuadro.

El método fillRect pinta un cuadro relleno del color de llenado que se haya establecido previamente (por defecto el color de relleno es negro). Por ejemplo:

```

3   </head>
4   <body>
5     <h1>Canvas</h1>
6     <canvas id="lienzo">
7     </canvas>
8     <script>
9       // se busca el canvas donde se va a pintar
10      var canvas=document.getElementById("lienzo");
11      if (canvas.getContext) {
12        // se obtiene el objeto context
13        var ctx = canvas.getContext("2d");
14        ctx.fillRect(20,20,200,100)
15      }
16    </script>
17  </body>
18

```

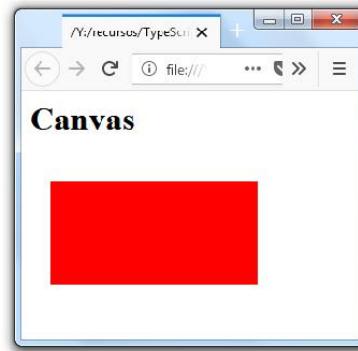


Se puede establecer el color de fondo previamente:

```

3
4      </head>
5      <body>
6          <h1>Canvas</h1>
7          <canvas id="lienzo">
8          </canvas>
9          <script>
10         // se busca el canvas donde se va a pintar
11         var canvas=document.getElementById("lienzo");
12         if (canvas.getContext) {
13             // se obtiene el objeto context
14             var ctx = canvas.getContext("2d");
15             ctx.fillStyle = 'red';
16             ctx.fillRect(20,20,200,100)
17         }
18     </script>

```



5.2.- Pintar Círculos

En el canvas no hay un método para pintar específicamente círculos, en su lugar, existe un método para dibujar arcos centrados en una posición X,Y, con un radio, un ángulo de inicio y un ángulo de fin. Existen 2 métodos para pintar arcos:

- arc: pinta un arco en cierto punto, con cierto radio, desde un ángulo de inicio hasta un ángulo de fin, en sentido de las agujas del reloj o en sentido contrario.
- arcTo: pinta un arco desde cierto punto hasta otro.

Los ángulos de inicio y fin para pintar el arco se utilizan en radianes, por lo tanto, si se desea un arco de cierta cantidad de grados (entre 0 y 360 grados), se debe usar una fórmula para convertir de grados a radianes:

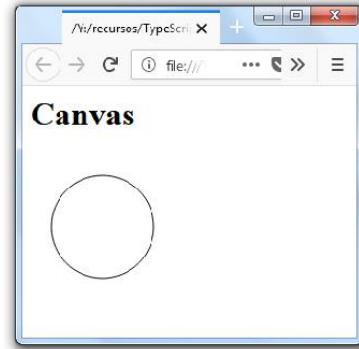
$$\text{radians} = (\text{Math.PI}/180)*\text{degrees}.$$

Para pintar un círculo se debe usar desde el ángulo 0 hasta el ángulo $\text{Math.PI}*2$. En el siguiente ejemplo se pinta un círculo de radio 50 desde el punto 70,70:

```

4   </head>
5   <body>
6     <h1>Canvas</h1>
7     <canvas id="lienzo">
8     </canvas>
9     <script>
10    // se busca el canvas donde se va a pintar
11    var canvas=document.getElementById("lienzo");
12    if (canvas.getContext) {
13      // se obtiene el objeto context
14      var ctx = canvas.getContext("2d");
15      ctx.arc(70, 70, 50, 0, Math.PI*2);
16      ctx.stroke();
17    }
18  </script>
19 </body>

```

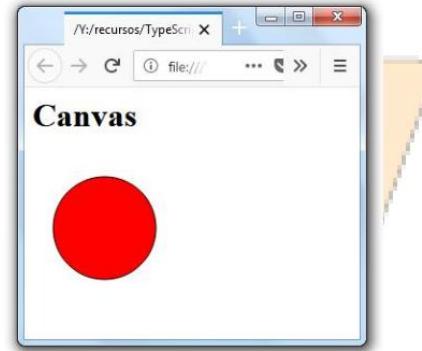


Si se necesita pintar el círculo, se debe usar el método `fill()`, por ejemplo:

```

4   </head>
5   <body>
6     <h1>Canvas</h1>
7     <canvas id="lienzo">
8     </canvas>
9     <script>
10    // se busca el canvas donde se va a pintar
11    var canvas=document.getElementById("lienzo");
12    if (canvas.getContext) {
13      // se obtiene el objeto context
14      var ctx = canvas.getContext("2d");
15      ctx.fillStyle = 'red';
16      ctx.arc(70, 70, 50, 0, Math.PI*2);
17      ctx.fill();
18      ctx.stroke();
19    }
20  </script>

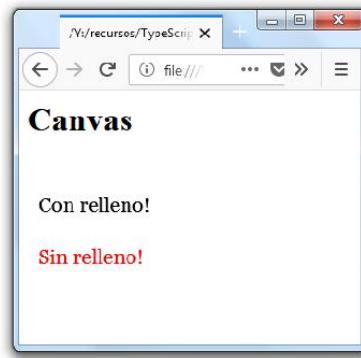
```



5.3.- Escribir Texto

Para escribir texto en el canvas se utiliza los métodos "strokeText" y "fillText", que reciben por parámetro la posición X,Y donde se desea ubicar el texto y el string con el texto que se desea mostrar. La diferencia entre stroke y fill es que el primero pinta un texto sin relleno y el segundo pinta el texto con relleno. Por ejemplo:

```
6   <h1>Canvas</h1>
7   <canvas id="lienzo">
8   </canvas>
9   <script>
10  // se busca el canvas donde se va a pintar
11  var canvas=document.getElementById("lienzo");
12  if (canvas.getContext) {
13      // se obtiene el objeto context
14      var ctx = canvas.getContext("2d");
15      ctx.font="20px Georgia";
16      ctx.fillText("Con relleno!",10,50);
17      ctx.fillStyle = 'red';
18      ctx.fillText("Sin relleno!",10,100);
19  }
20  </script>
21  </body>
```



Con el método "font" se establece el tamaño de la letra y el tipo de letra del texto que se desea generar.



Capítulo 6. ENUMS

6.1.- Concepto

Los enumerados permiten definir tipos de datos con valores predefinidos enumerables, que pueden facilitar la evaluación de ciertos posibles valores que puede tomar una variables. Ejemplos de enumerados son los números enteros, los días de la semana, los meses del año, las estaciones, etc.

Los enums son muy útiles al momento de limitar los posibles valores que puede tomar una variable, que de otro modo, por ejemplo, usando el tipo string, no se podría. Por ejemplo, si una variable almacena días de la semana (Lunes, Martes, Miércoles, etc.) y es inicializada en el valor "Martes", si se necesita determinar cuál es el día anterior o el siguiente, el enumerado limitaría el valor que puede tomar.

6.2.- Declaración de Enums

Para declarar un enumerado se utiliza la palabra reservada "enum" seguido de los valores que abarca el enumerado encerrado entre llaves. De forma general:

```
enum {  
    valor1,  
    valor2,  
    valorN  
}
```

Por ejemplo:

```
1  enum DiasSemana {  
2      Lunes,  
3      Martes,  
4      Miercoles,  
5      Jueves,  
6      Viernes  
7  }  
8  
9  enum TipoFigura {  
10     Cuadrado = 1,  
11     Triangulo = 2,  
12     Circulo = 3  
13 }
```

Cada valor del enumerado puede también tener valores alfanuméricos asignados, por ejemplo:

```
1  enum Estaciones {  
2      Invierno = "Invierno",  
3      Primavera = "Primavera",  
4      Verano = "Verano",  
5      Otoño = "Otoño"  
6 }
```

6.3.- Uso de Enums

Un enumerado es un tipo de dato, por lo tanto, se pueden declarar variables de ese tipo, se pueden pasar parámetros a funciones y se pueden retornar valores del tipo enum. Por ejemplo:

```
1  enum DiasSemana {  
2      Lunes,  
3      Martes,  
4      Miércoles,  
5      Jueves,  
6      Viernes  
7 }  
8  
9  let dia: DiasSemana;
```

Si una variable está declarada de un tipo Enum, sólo puede tomar valores definidos en el enumerado. Por ejemplo:

```
1 enum DiasSemana {  
2     Lunes,  
3     Martes,  
4     Miercoles,  
5     Jueves,  
6     Viernes  
7 }  
8  
9 let dia: DiasSemana;  
10  
11 // es válido  
12 dia = DiasSemana.Martes;  
13 // es un error, "lunes" no es del Enum DiasSemana  
14 dia = "lunes"
```

Los valores del enumerado se pueden acceder como posiciones de un arreglo:

```
1 enum DiasSemana {  
2     Lunes,  
3     Martes,  
4     Miercoles,  
5     Jueves,  
6     Viernes  
7 }  
8  
9 // Se imprimirá el valor "Lunes"  
10 console.log(DiasSemana[0]);
```

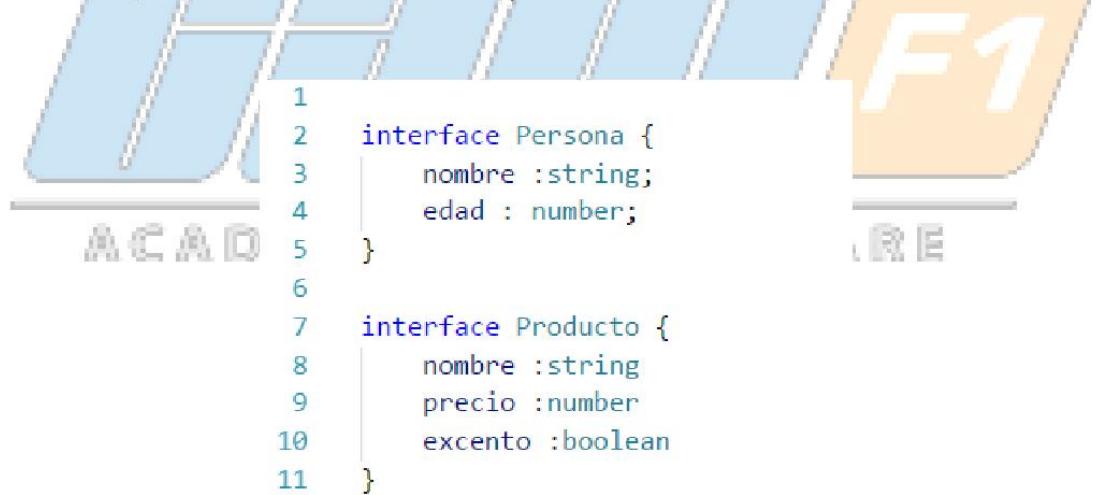
Capítulo 7. INTERFACES

7.1.- Concepto

Una interfaz en TypeScript permite definir tipos de datos estructurados. Es un nombre que representa un tipo de dato que puede abarcar variables de distintos tipos de datos. Es lo más similar al uso de objetos en JavaScript. Es similar también al concepto de estructuras o registros de otros lenguajes de programación como C o Pascal.

7.2.- Declaración de Interfaces

Las interfaces se definen usando la palabra reservada "interface" y un nombre seguido de un par de llaves { }. Entre las llaves se definen las variables internas a las interfaces. No hay límite en cuanto a la cantidad de variables que se pueden definir internamente en una interface. Por ejemplo, en la siguiente imagen se definen 2 interfaces, una con el nombre Persona y otra con el nombre Producto:

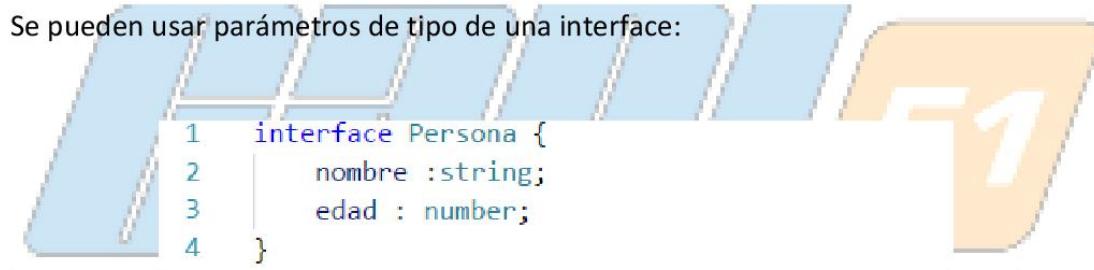


7.3.- Uso de Interfaces

Se pueden declarar variables usando como tipo de dato una interface definida previamente. También se puede usar en los tipos de datos de un parámetro o el tipo de dato de retorno de una función. Por ejemplo, si se define una interface con el nombre "Persona" y se declara una variable con este tipo de dato, la variable puede tener valores para cada variable interna de la interface:

```
1 interface Persona {  
2     nombre :string;  
3     edad : number;  
4 }  
5  
6 let q: Persona = { nombre:"Ana", edad:20}  
7 let p: Persona  
8  
9 p.nombre = "Jose Luis";  
10 p.edad = 20;  
11  
12 q = p; // es válido porque son del mismo tipo
```

Se pueden usar parámetros de tipo de una interface:



```
1 interface Persona {  
2     nombre :string;  
3     edad : number;  
4 }  
5  
6 function esMayor(x :Persona) :boolean {  
7     if (x.edad >= 18)  
8         return true;  
9     else  
10        return false;  
11 }
```

Al intentar hacer referencia a atributos que no existan en la interface o asignar valores a una variable declarada con un tipo de dato interface el compilador genera errores:

```
1  interface Persona {  
2      nombre :string;  
3      edad : number;  
4  }  
5  
6  let p: Persona  
7  
8  p.nombre = "Jose Luis"  
9  p.edad = 20  
10 p.peso = 70 // no hay un atributo peso  
11  
12 p = 100 // no son del mismo tipo
```

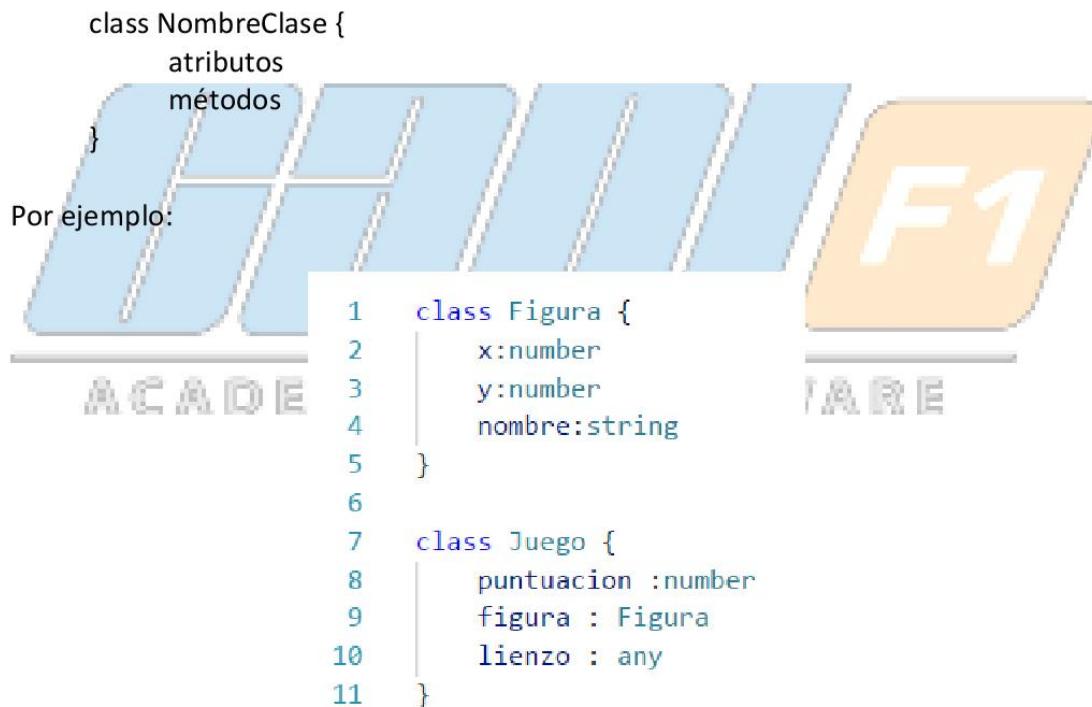


Capítulo 8. CLASES

8.1.- Definición

Una de las mayores diferencias de TypeScript con JavaScript es la formalidad para trabajar con el paradigma de la programación orientada a objetos. En las más recientes versiones de JavaScript se han incorporado nuevos elementos que han mejorado mucho, pero aún hay carencias en comparación con lenguajes más formales como C++, Java, C#, entre otros.

En TypeScript una clase se define usando la palabra reservada "class" seguido del nombre de la clase. De forma general:



8.2.- Instanciación

Para instanciar una clase se utiliza la palabra reservada "new" seguido del nombre de la clase. De forma general:

```
let objeto = new Clase();
```

Luego se puede acceder a los atributos de la clase usando el objeto. Por ejemplo:

```
1  class Figura {  
2      x:number  
3      y:number  
4      nombre:string  
5  }  
6  
7  let p :Figura  
8  p = new Figura()  
9  
10 p.x = 50  
11 p.y = 120  
12  
13 let f :Figura = new Figura()
```

8.3.- Métodos

Los métodos se declaran entre las llaves de la definición de la clase y deben tener las características de cualquier función de JavaScript, aunque se puede omitir la palabra reservada "function". Por ejemplo:

```
1  class Figura {  
2      x:number  
3      y:number  
4      nombre:string  
5  
6      mostrarInformacion(){  
7          console.log("La figura se llama: ")  
8      }  
9  }  
10  
11 let p :Figura  
12 p = new Figura()  
13 p.mostrarInformacion()
```

Para acceder a los atributos de una clase desde un método se debe utilizar obligatoriamente la palabra reservada "this" antes del nombre del atributo. Por ejemplo:

```
1  class Figura {  
2      x:number  
3      y:number  
4      nombre:string  
5  
6      mostrarInformacion(){  
7          console.log("La figura se llama: ")  
8          console.log(this.nombre)  
9      }  
10 }  
11  
12 let p :Figura  
13 p = new Figura()  
14 p.nombre = "Rombo"  
15 p.mostrarInformacion()
```

Capítulo 9. MODIFICADORES DE ACCESO

9.1.- Public

Como se ha visto en los ejemplos anteriores, se puede acceder libremente a los atributos y métodos definidos en las clases. Esto es porque en TypeScript los miembros de una clase son por defecto públicos, sin embargo, se pueden etiquetar como públicos explícitamente los atributos o métodos utilizando la etiqueta "public". El siguiente es un ejemplo:

```
1  class Figura {  
2      public x:number  
3      nombre:string  
4  }  
5  
6  
7  let f :Figura = new Figura()  
8  // es válido, "x" es explícitamente público  
9  f.x = 15;  
10 // es válido, "nombre" es público por defecto  
11 f.nombre = "círculo"
```

ACADEMIA DE SOFTWARE

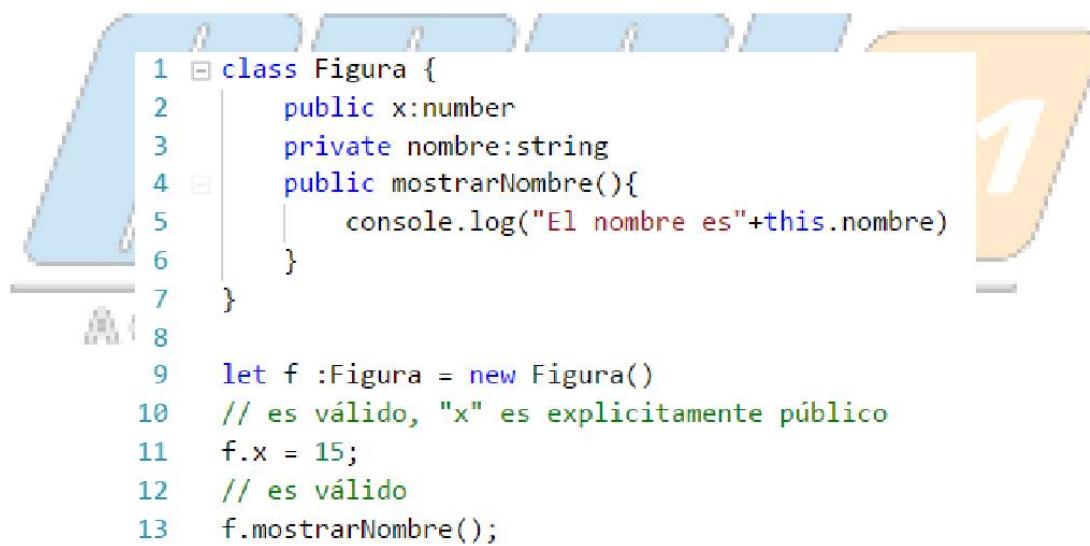
Aunque en el ejemplo se hace referencia a atributos, el comportamiento descrito también aplica a los métodos de la clase.

9.2.- Private

En caso de necesitar ocultar el acceso a atributos o métodos de una clase, se utiliza el modificador "private". Por ejemplo:

```
1  class Figura {  
2      public x:number  
3      private nombre:string  
4  }  
5  
6  
7  let f :Figura = new Figura()  
8  // es válido, "x" es explícitamente público  
9  f.x = 15;  
10 // es inválido, "nombre" es privado  
11 f.nombre = "círculo"
```

Si se necesita mostrar algún atributo privado, se utiliza un método público para tal fin. Más adelante se explicará cómo usar "getters" y "setter". Por ejemplo:



9.3.- Readonly

Un atributo puede ser declarado como constante o de sólo lectura. Esto se logra agregando el modificador "readonly" en la declaración del atributo. Los atributos de sólo lectura deben ser inicializados en su declaración o en el constructor de la clase (se verá más adelante). Por ejemplo:

```
1  class Figura {  
2      public readonly x:number = 50;  
3      public y: number = 20;  
4  
5      public mostrarPosicion(){  
6          console.log("X :" +this.x)  
7          console.log("Y :" +this.y)  
8      }  
9  }  
10  
11 let f :Figura = new Figura()  
12 // mostrará X:50, Y:20  
13 f.mostrarPosicion()  
14 // es inválido, "x" es de sólo lectura  
15 f.x = 15;  
16 // es válido  
17 f.y = 30
```



ACADEMIA DE SOFTWARE

Capítulo 10. GETTES Y SETTER

10.1.- Get

Cuando una clase tiene atributos privados debe tener mecanismos para acceder a estos. El mecanismo es crear los métodos "getters" y "setters". TypeScript posee palabras reservadas especiales para crear estos métodos. Para crear un método "getter" de un atributo se define un método usando la palabra "get" antes del nombre del mismo. Por ejemplo:

```
1  class Figura {  
2      private _x:number = 50;  
3      private _y: number = 20;  
4  
5      get x():number {  
6          return this._x  
7      }  
8      get y():number {  
9          return this._y  
10     }  
11 }
```



ACADEMIA DE SOFTWARE

Cuando se crean métodos con la palabra "get" antepuesta, se hace uso de éstos como si fueran atributos de la clase. Por ejemplo:

```
1  class Figura {  
2      private _x:number = 50;  
3      private _y: number = 20;  
4  
5      get x():number {  
6          return this._x  
7      }  
8      get y():number {  
9          return this._y  
10     }  
11 }  
12  
13 let f :Figura = new Figura()  
14 // se usan a los getter como atributos  
15 // se imprimirá: 50,20  
16 console.log(f.x+ " , "+f.y)  
17 // error porque el método "x" es un getter  
18 f.x = 20
```

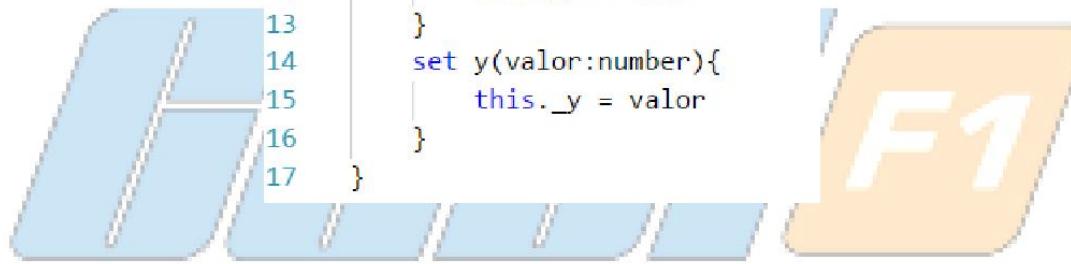
10.2.- Set

El método "setter" de un atributo permite modificar su valor desde el exterior de la clase. Así mismo como existe la palabra "get" para el getter, existe la palabra reservada "set" que permite crear el método setter de un atributo. Por ejemplo:

```
1  class Figura {  
2      private _x:number = 50;  
3      private _y: number = 20;  
4  
5      get x():number {  
6          return this._x  
7      }  
8      set x(valor:number){  
9          this._x = valor  
10     }  
11 }
```

En teoría, todos los atributos privados deberían tener sus respectivos métodos setter. Por ejemplo:

```
1  class Figura {  
2      private _x:number = 50;  
3      private _y: number = 20;  
4  
5      get x():number {  
6          return this._x  
7      }  
8      get y():number {  
9          return this._y  
10     }  
11     set x(valor:number){  
12         this._x = valor  
13     }  
14     set y(valor:number){  
15         this._y = valor  
16     }  
17 }
```



Cuando se crea un método setter, se puede ejecutar como si fuera un atributo de la clase. El uso del método setter sería el siguiente:

```
19 let f :Figura = new Figura()  
20 // se usan a los getter como atributos  
21 // se imprimirá: 50,20  
22 console.log(f.x+'' , "+f.y)  
23 // se ejecuta el setter del atributo X  
24 f.x = 20  
25 f.y = 30  
26 // se imprimirá: 20,30  
27 console.log(f.x+'' , "+f.y)
```

Capítulo 11. CONSTRUCTORES

11.1.- Definición

Un método constructor se ejecuta automáticamente al instanciar una clase. En TypeScript, un método se asume como el constructor de la clase cuando en su definición se le antepone la palabra reservada "constructor" antes del nombre del método. Por ejemplo:

```
1  class Figura {  
2      private _x:number = 50;  
3  
4      constructor(){  
5          this._x = Math.random();  
6      }  
7      get x():number {  
8          return this._x  
9      }  
10     set x(valor:number){  
11         this._x = valor  
12     }  
13 }
```



ACADEMIA DE SOFTWARE

En TypeScript, una clase sólo puede tener un método constructor. Un constructor puede tener parámetros, por ejemplo:

```
1  class Figura {  
2      private _x:number = 50;  
3      private _y:number = 80;  
4  
5      constructor(nuevoX){  
6          this._x = nuevoX;  
7      }  
8      get x():number {  
9          return this._x  
10     }  
11     set x(valor:number){  
12         this._x = valor  
13     }  
14 }
```

11.2.- Uso Del Constructor

Como se ha explicado anteriormente, una clase se instancia usando la palabra reservada "new" seguido del nombre de la clase que se desea instanciar. Es en ese momento que se ejecuta el constructor. Por ejemplo:

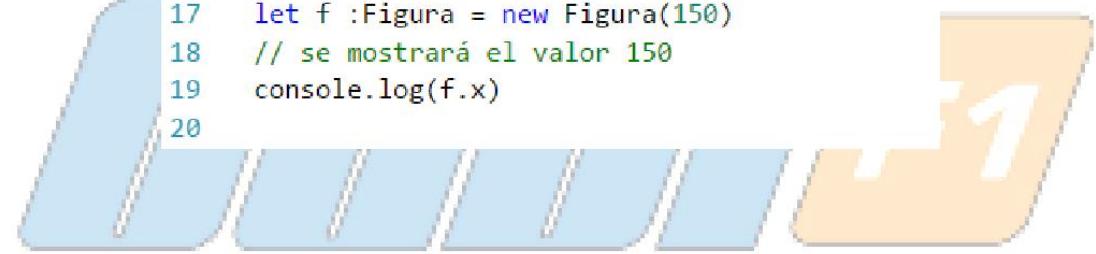
ACADEMIA DE SOFTWARE

```
1  class Figura {  
2      private _x:number = 50;  
3  
4      constructor(){  
5          // se pierde el valor inicial de _x  
6          // asignado en la declaración  
7          this._x = Math.random();  
8      }  
9      get x():number {  
10         return this._x  
11     }  
12     set x(valor:number){  
13         this._x = valor  
14     }  
15 }  
16  
17 let f :Figura = new Figura()  
18 // se mostrará el valor aleatorio que toma  
19 // el atributo _x al ejecutarse el constructor  
20 console.log(f.x)
```

Si el constructor tiene parámetros, estos se colocan entre los paréntesis que van después del nombre de la clase. Por ejemplo:

ACADEMIA DE SOFTWARE

```
1  class Figura {  
2      private _x:number = 50;  
3  
4      constructor(nuevoX:number){  
5          // se pierde el valor inicial de _x  
6          // asignado en la declaración  
7          this._x = nuevoX;  
8      }  
9      get x():number {  
10         return this._x  
11     }  
12     set x(valor:number){  
13         this._x = valor  
14     }  
15 }  
16  
17 let f :Figura = new Figura(150)  
18 // se mostrará el valor 150  
19 console.log(f.x)  
20
```



ACADEMIA DE SOFTWARE

Capítulo 12. HERENCIA

12.1.- Sub Clases

TypeScript permite definir una clase en base a otra clase ya existente. Para lograrlo, se utiliza la palabra reservada "extends" seguida del nombre de la clase que se utilizará como base. La clase base debe estar definida previamente. Por ejemplo:

```
1 class Figura {  
2     _x:number = 50;  
3     _y:number = 80;  
4     color:string  
5 }  
6 class Circulo extends Figura {  
7     _radio: number;  
8 }  
9 }
```

En los métodos de la clase derivada se pueden acceder a los atributos públicos de la clase base como si fueran propios. Por ejemplo:

ACADEMIA DE SOFTWARE

```

1  class Figura {
2      _x:number = 50;
3      _y:number = 80;
4      color:string
5      pintar(){
6          console.log("el color es "+this.color)
7      }
8  }
9  class Circulo extends Figura {
10     _radio: number;
11     mostrar(){
12         this.pintar()
13         console.log("Posicion: "+this._x+","+this._y)
14         console.log("Radio: "+this._radio)
15     }
16 }
```

Cuando se instancia un objeto de la clase heredera (también llamada sub clase), éste puede acceder a los atributos y métodos públicos tanto de la sub clase como de la clase base. Por ejemplo:

```

18 let f :Figura = new Figura()
19 let c :Circulo = new Circulo()
20
21 f.pintar();
22 c.mostrar();
23 c.pintar();
24 // es un error. La clase Figura no tiene el método mostrar
25 f.mostrar();
```

12.2.- Protected

Los atributos y métodos privados solo pueden ser accedidos por los métodos de la misma clase. Ni siquiera las subclases pueden acceder a estos. Para permitir acceder a atributos y métodos en una sub clase se utiliza el modificador "protected". Un atributo o método marcado como protected se comporta como público para las subclases pero se comporta como privado para el resto de las clases. Por ejemplo:

```

1  class Figura {
2      private _x:number = 50;
3      protected _y:number = 80;
4      color:string
5      protected pintar(){
6          console.log("el color es "+this.color)
7      }
8  }
9  class Circulo extends Figura {
10     _radio: number;
11     mostrar(){
12         this.pintar()
13         // no se puede acceder a _x porque es privado
14         console.log("Posicion: "+this._x+","+this._y)
15         console.log("Radio: "+this._radio)
16     }
17 }
```

Un objeto de una clase que tenga atributos `protected` no podrá ejecutar estos métodos directamente ni acceder a atributos porque se asumen como privados:

ACADEMIA DE SOFTWARE

```

19  let f :Figura = new Figura()
20  let c :Circulo = new Circulo()
21
22  // no se puede acceder porque se asume como privado
23  f.pintar();
24  // tampoco un objeto de una sub clase de Figura
25  // puede acceder al método protected
26  c.mostrar();
27  c.pintar();
```

12.3.- Super

Cuando una clase tiene un método constructor y es la clase base de otra, el constructor de la clase derivada debe obligatoriamente ejecutar el constructor de la

super clase colocando la palabra reservada "super" al comienzo del constructor de la sub clase. Por ejemplo:

```
1 class Figura {  
2     private _x:number = 50;  
3     protected _y:number = 80;  
4     color:string  
5     constructor(){  
6         this._y = Math.random();  
7     }  
8     protected pintar(){  
9         console.log("el color es "+this.color)  
10    }  
11}  
12 class Circulo extends Figura {  
13     _radio: number;  
14     constructor(){  
15         super();  
16         this._radio = Math.random();  
17     }  
18}
```

Super también es muy usado cuando una sub clase sobre escribe un método de la súper clase (se verá más adelante). Cuando el constructor de la súper clase tiene parámetros, también se deben pasar los parámetros utilizando "super". Por ejemplo:

```
1  class Figura {  
2      private _x:number = 50;  
3      protected _y:number = 80;  
4      color:string  
5      constructor(nuevoY){  
6          this._y = nuevoY;  
7      }  
8      protected pintar(){  
9          console.log("el color es "+this.color)  
10     }  
11 }  
12 class Circulo extends Figura {  
13     _radio: number;  
14     constructor(){  
15         super(15);  
16         this._radio = Math.random();  
17     }  
18 }
```

Se pueden pasar parámetros al constructor de la sub clase para luego pasarlo al constructor de la super clase. Por ejemplo:

ACADEMIA DE SOFTWARE

```
1  class Figura {  
2      private _x:number = 50;  
3      protected _y:number = 80;  
4      color:string  
5      constructor(nuevoY){  
6          this._y = nuevoY;  
7      }  
8      protected pintar(){  
9          console.log("el color es "+this.color)  
10     }  
11 }  
12 class Circulo extends Figura {  
13     _radio: number;  
14     constructor(nuevoY,nuevoRadio){  
15         super(nuevoY);  
16         this._radio = nuevoRadio;  
17     }  
18 }
```

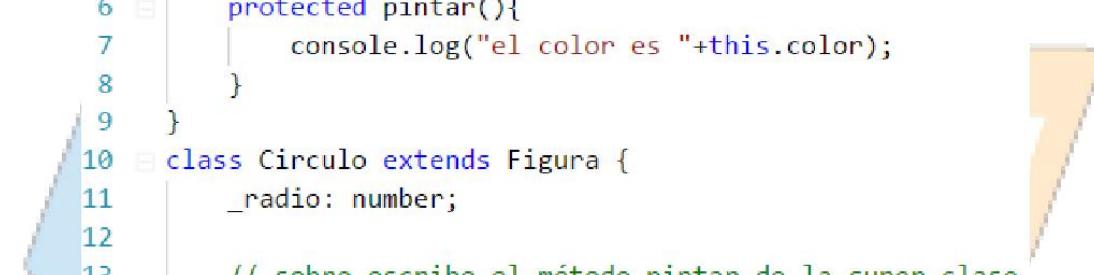


ACADEMIA DE SOFTWARE

Capítulo 13. CLASES ABSTRACTAS

13.1.- Sobre Escritura de Métodos

Un método en una sub clase puede llamarse igual que un método de la súper clase. En este caso, el método de la sub clase sobre escribe el método de la súper clase. Por ejemplo:



```
1 class Figura {
2     private _x:number = 50;
3     protected _y:number = 80;
4     color:string
5
6     protected pintar(){
7         console.log("el color es "+this.color);
8     }
9 }
10 class Circulo extends Figura {
11     _radio: number;
12
13     // sobre escribe el método pintar de la super clase
14     public pintar(){
15         console.log("pintar un circulo");
16     }
17 }
```

Por lo tanto, si un objeto de la sub clase intenta ejecutar dicho método, ejecutará el de la sub clase. Por ejemplo:

```
19 let f :Figura = new Figura()
20 let c :Circulo = new Circulo()
21
22 // muestra el mensaje: "El color es ..."
23 f.pintar();
24 // muestra el mensaje: "pintar un circulo"
25 c.pintar();
```

Para ejecutar el método de la súper clase, se utiliza la palabra reservada "super". Por ejemplo:

```
10  class Circulo extends Figura {  
11      _radio: number;  
12  
13      // sobre escribe el método pintar de la super clase  
14      public pintar(){  
15          super.pintar();  
16          console.log("pintar un circulo");  
17      }  
18  }  
19  
20  let f :Figura = new Figura()  
21  let c :Circulo = new Circulo()  
22  
23  // muestra el mensaje: "El color es ..."  
24  f.pintar();  
25  // muestra el mensaje: "el color es..."  
26  // y luego el mensaje: "pintar un circulo"  
27  c.pintar();
```

13.2.- Clases Abstractas

Una clase abstracta es una clase de la cual no se pueden instanciar objetos. Se utiliza solamente como base de otras clases. Para definir una clase como abstracta se utiliza la palabra reservada "abstract" antes de la definición de la clase. Por ejemplo:

```
1 abstract class Figura {  
2     private _x:number = 50;  
3     protected _y:number = 80;  
4     color:string  
5  
6     public pintar(){  
7         console.log("el color es "+this.color);  
8     }  
9 }  
10  
11 let f :Figura = new Figura()
```

Una clase abstracta siempre puede usarse como base para otra clase:

```
1 abstract class Figura {  
2     private _x:number = 50;  
3     protected _y:number = 80;  
4     color:string  
5  
6     public pintar(){  
7         console.log("el color es "+this.color);  
8     }  
9 }  
10 // es válido que una clase abstracta sea base de otra  
11 class Circulo extends Figura {  
12     _radio: number;  
13  
14     // sobre escribe el método pintar de la super clase  
15     public pintar(){  
16         super.pintar();  
17         console.log("pintar un circulo");  
18     }  
19 }
```

13.3.- Métodos Abstractos

Un método abstracto es un método que se define en una clase pero no se programa en ésta. El método debe ser programado (sobre escrito) en una sub clase. Cuando una clase tiene por lo menos un método abstracto, la clase debe ser definida como abstracta. Por ejemplo:

```
1  abstract class Figura {  
2      private _x:number = 50;  
3      protected _y:number = 80;  
4      color:string  
5  
6      public abstract pintar();  
7 }
```

Si una clase hereda de una clase que tiene un método abstracto, está obligada a programar el método abstracto o a definirse como clase abstracta también. Por ejemplo:

```
1  abstract class Figura {  
2      private _x:number = 50;  
3      protected _y:number = 80;  
4      color:string  
5  
6      public abstract pintar();  
7 }  
8 // Marca un error porque no implementa  
9 // el método abstracto  
10 class Circulo extends Figura {  
11     _radio: number;  
12 }
```

El siguiente ejemplo muestra como implementar el método abstracto de la súper clase:

```
1  abstract class Figura {  
2      private _x:number = 50;  
3      protected _y:number = 80;  
4      color:string  
5  
6      public abstract pintar();  
7  }  
8  
9  class Circulo extends Figura {  
10     _radio: number;  
11  
12     // sobre escribe el método pintar de la super clase  
13     public pintar(){  
14         console.log("pintar un circulo");  
15     }  
16 }
```



ACADEMIA DE SOFTWARE

Capítulo 14. MÉTODOS Y ATRIBUTOS ESTÁTICOS

14.1.- Métodos Estáticos

Un método estático es un método que se puede ejecutar sin necesidad de instanciar la clase a la cual pertenece. Se utiliza el nombre de la clase, de la forma: Clase.método (en vez de objeto.método). Ejemplo de métodos estáticos son los métodos de la clase "Math", por ejemplo:

```
1  
2  let x= Math.random();  
3  let y= Math.sqrt(x);  
4
```

Para definir un método estático se antepone a la definición del método la palabra reservada "static". Por ejemplo:



```
1  class Juego{  
2  
3      static jugar(){  
4          console.log("Este método es estatico")  
5      }  
6      inicializar(){  
7          console.log("Este método NO es estatico")  
8      }  
9  }  
10  
11  Juego.jugar();
```

Es un error intentar ejecutar un método estático usando un objeto, como también es un error intentar ejecutar un método no estático utilizando un objeto. Por ejemplo:

```
1  class Juego{  
2      static jugar(){  
3          console.log("Este método es estatico")  
4      }  
5      inicializar(){  
6          console.log("Este método NO es estatico")  
7      }  
8  }  
9 // es correcto  
10 Juego.jugar();  
11 // No es un método estático  
12 Juego.inicializar();  
13 //  
14 let x:Juego = new Juego();  
15 // es correcto  
16 x.inicializar();  
17 // Es un método estático  
18 x.jugar();  
19 //
```

14.2.- Atributos Estáticos

Un atributo estático es una variable declarada internamente en una clase, que se utiliza por medio de la clase y no por medio de una instancia de la clase. Por ejemplo:

```
1  class Juego{  
2      marcador :number;  
3      static jugador :string;  
4  
5      static jugar(){  
6          console.log("Este método es estatico")  
7      }  
8      inicializar(){  
9          console.log("Este método NO es estatico")  
10     }  
11 }  
12 Juego.jugador = "Jose L.";
```

No se pueden manipular atributos estáticos desde objetos de la clase, así como tampoco se puede acceder a atributos no estáticos por medio del nombre de la clase. Por ejemplo:

```
1  class Juego{  
2      marcador :number;  
3      static jugador :string;  
4  
5      static jugar(){  
6          console.log("Este método es estatico")  
7      }  
8      inicializar(){  
9          console.log("Este método NO es estatico")  
10     }  
11 }  
12 Juego.jugador = "Jose L.";  
13  
14 let x:Juego = new Juego();  
15 // no es válido porque "jugador" es estático  
16 x.jugador = "María";  
17 // es válido, porque no es estático  
18 x.marcador = 100  
19 // el atributo "marcador" no es estatico  
20 Juego.marcador = 500;
```

Hay algunas reglas que se deben respetar con respecto a la relación entre atributos estáticos y métodos estáticos. Entre estas:

- un método estático sólo puede acceder atributos estáticos.
- un método no estático puede acceder a atributos estáticos utilizando el nombre de la clase seguido del nombre del atributo (Clase.atributo).

```
1  class Juego{  
2      marcador :number;  
3      static jugador :string;  
4  
5      static jugar(){  
6          // error. el atributo no es estático  
7          this.marcador = 10;  
8          // es correcto, aunque puede ser confuso  
9          this.jugador = "Jesus"  
10         console.log("Este método es estatico")  
11     }  
12     inicializar(){  
13         // es correcto  
14         this.marcador = 50;  
15         // no se permite this para atributos estáticos  
16         // en contextos no estáticos  
17         this.jugador=""  
18         // es válido usando el nombre de la clase  
19         Juego.jugador = "José"  
20         console.log("Este método NO es estatico")  
21     }  
22 }
```

ACADEMIA DE SOFTWARE