54 - TypeScript: funciones y métodos

TypeScript aporta varias características que JavaScript no dispone hasta el momento cuando tenemos que plantear funciones y métodos.

Parámetros tipados y funciones que retornan un valor.

Podemos indicar a cada parámetro el tipo de dato que puede recibir y también el tipo de dato que retorna la función o método en caso que estemos en una clase:

```
function sumar(valor1:number, valor2:number): number {
  return valor1+valor2;
}

console.log(sumar(10, 5));
```

La función sumar recibe dos parámetros de tipo number y retorna un valor de tipo number. Luego si llamamos a esta función enviando un valor distinto a number el compilador nos avisará del error:

```
console.log(sumar('juan', 'carlos'));
```

Se genera un error: Argument of type 'string' is not assignable to parameter of type 'number'.

Inclusive editores de texto moderno como Visual Studio Code pueden antes de compilarse avisar del error.

El tipado estático favorece a identificar este tipo de errores antes de ejecutar la aplicación. Lo mismo cuando una función retorna un dato debemos indicar al final de la misma dicho tipo:

```
function sumar(valor1:number, valor2:number): number {
```

La función sumar retorna un valor de tipo number.

Luego si la función retorna un tipo distinto a number se genera un error:

```
function sumar(valor1:number, valor2:number): number {
  return 'Hola mundo';
}
```

Como estamos retornando un string se genera el error: Type 'string' is not assignable to type 'number'

Funciones anónimas.

Una función anónima no especifica un nombre. Son semejantes a JavaScript con la salvedad de la definición de tipos para los parámetros:

```
const funcSumar = function (valor1:number, valor2:number): number {
  return valor1 + valor2;
}

console.log(funcSumar(4, 9));
```

Parámetros opcionales.

En TypeScript debemos agregar el caracter '?' al nombre del parámetro para indicar que el mismo puede o no llegar un dato:

```
function sumar(valor1: number, valor2: number, valor3?: number): number {
  if (valor3)
    return valor1 + valor2 + valor3;
  else
    return valor1 + valor2;
}

console.log(sumar(5, 4));
console.log(sumar(5, 4, 3));
```

El tercer parámetro es opcional:

```
function sumar(valor1: number, valor2: number, valor3?: number): number {
```

Luego a la función 'sumar' la podemos llamar pasando 2 o 3 valores numéricos:

```
console.log(sumar(5,4));
console.log(sumar(5,4,3));
```

Si pasamos una cantidad de parámetros distinta a 2 o 3 se genera un error en tiempo de compilación: ' error TS2554: Expected 2-3 arguments, but got 4.'

Los parámetros opcionales deben ser los últimos parámetros definidos en la función. Puede tener tantos parámetros opcionales como se necesiten.

Parámetros por defecto.

Esta característica de TypeScript nos permite asignar un valor por defecto a un parámetro para los casos en que la llamada a la misma no se le envíe.

```
function sumar(valor1: number, valor2: number, valor3: number = 0): number {
  return valor1 + valor2 + valor3;
}

console.log(sumar(5, 4));
console.log(sumar(5, 4, 3));
```

El tercer parámetro almacena un cero si no se lo pasamos en la llamada:

```
console.log(sumar(5, 4));
```

Puede haber varios valores por defecto, pero deben ser los últimos. Es decir primero indicamos los parámetros que reciben datos en forma obligatoria cuando los llamamos y finalmente indicamos aquellos que tienen valores por defecto.

parámetros Rest.

Otra característica de TypeScript es la posibilidad de pasar una lista indefinida de valores y que los reciba un vector.

El concepto de parámetro Rest se logra antecediendo tres puntos al nombre del parámetro:

```
function sumar(...valores: number[]) {
  let suma = 0;
  for (let x = 0; x < valores.length; x++)
    suma += valores[x];
  return suma;
}

console.log(sumar(10, 2, 44, 3));
console.log(sumar(1, 2));
console.log(sumar());
```

El parámetro 'valores' se le anteceden los tres puntos seguidos e indicamos que se trata de un vector de tipo 'number'. Cuando llamamos a la función le pasamos una lista de valores enteros que luego la función los empaqueta

en el vector:

```
console.log(sumar(10, 2, 44, 3));
console.log(sumar(1, 2));
console.log(sumar());
```

La función con un parámetro Rest puede tener otros parámetros pero se deben declarar antes.

Los parámetros Rest no pueden tener valores por defecto.

operador Spread.

El operador Spread permite descomponer una estructura de datos en elementos individuales. Es la operación inversa de los parámetros Rest. La sintaxis se aplica anteponiendo al nombre de la variable tres puntos:

```
function sumar(valor1: number, valor2: number, valor3: number): number {
  return valor1 + valor2 + valor3;
}

const vec: [number, number, number] = [10, 20, 30];
const s = sumar(...vec);
console.log(s);
```

Otra cosa importante es hacer notar que el arreglo indicamos el tipo para cada elemento.

Funciones callbacks

Una función callback es una función que se pasa a otra función como parámetro y dentro de la misma es llamada.

```
function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number): number {
  return func(valor1, valor2);
}

console.log(operar(3, 7, (x: number,y: number): number => {
  return x+y;
}))

console.log(operar(3, 7, (x: number,y: number): number => {
  return x-y;
}))

console.log(operar(3, 7, (x: number,y: number): number => {
  return x*y;
}))
```

La función operar recibe tres parámetros, los dos primeros son de tipo 'number' y el tercero es de tipo función:

```
function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number): number {
```

La función que debe recibir debe tener como parámetros dos 'number' y retornar un 'number'.

Cuando llamamos a la función además de los dos enteros le debemos pasar una función que reciba dos 'number' y retorne un 'number':

```
console.log(operar(3, 7, (x: number,y: number): number => {
  return x+y;
}))
```

Como podemos observar llamamos a la función 'operar' tres veces y le pasamos funciones que procesan los dos enteros para obtener su suma, resta y multiplicación.

Podemos llamar a la función 'operar' con una sintaxis más concisa como en JavaScript:

```
console.log(operar(3, 7, (x: number,y: number): number => x+y))
console.log(operar(3, 7, (x: number,y: number): number => x-y))
console.log(operar(3, 7, (x: number,y: number): number => x*y))
```

Para hacer más claro nuestro código TypeScript mediante la palabra clave type permite crear nuevos tipos y luego reutilizarlos:

```
type Operacion = (x: number, y: number) => number;

function operar(valor1: number, valor2: number, func: Operacion): number {
  return func(valor1, valor2);
}

console.log(operar(3, 7, (x: number, y: number): number => x + y))
console.log(operar(3, 7, (x: number, y: number): number => x - y))
console.log(operar(3, 7, (x: number, y: number): number => x * y))
```

El tipo Operacion tiene la firma de una función con dos parámetros de tipo 'number' y el retorno de un 'number'. Luego cuando declaramos la función operar definimos el tercer parámetro llamado 'func' de tipo 'Operacion':

```
function operar(valor1: number, valor2: number, func: Operacion): number {
```

Parámetros de tipo unión.

Vimos en otro concepto que podemos definir variables que pueden almacenar más de un tipo de dato indicando los mismos el operador '|':

```
let edad: number | string;
edad=34;
console.log(edad);
edad='20 años';
console.log(edad);
```

Con parámetros podemos utilizar la misma sintaxis:

```
function sumar(valor1: number | string, valor2: number | string ): number | string  {
  if (typeof valor1 ==='number' && typeof valor2 ==='number')
    return valor1+valor2;
  else
    return valor1.toString() + valor2.toString();
}

console.log(sumar(4, 5));
console.log(sumar('Hola ', 2));
console.log(sumar('Hola ', 'Mundo'));
```

En este tipo de caso deberemos identificar que operación realizar según los tipos de datos de los parámetros. En el ejemplo si los dos parámetros se reciben tipos de datos 'number' procedemos a sumarlos como enteros:

```
  if (typeof valor1 ==='number' && typeof valor2 ==='number')
    return valor1+valor2;
```

En el caso contrario con que uno de los dos valores sea de tipo 'string' procedemos a concatenarlos, previamente los convertimos a string:

```
  else
    return valor1.toString() + valor2.toString();
```

Acotaciones

Hemos hecho siempre ejemplos con funciones, pero todos estos conceptos se aplican si planteamos métodos dentro de una clase:

```
class Operacion {
  sumar(...valores:number[]) {
    let suma=0;
    for(let x=0;x<valores.length;x++)
      suma+=valores[x];
    return suma;
  }
}

let op: Operacion;
op=new Operacion();
console.log(op.sumar(1,2,3));
```

Retornar