TypeScript
Funciones

CertiDevs

Índice de contenidos

1. Funciones

Las **funciones** son bloques de código que realizan una tarea específica y pueden ser llamadas varias veces.

En **TypeScript**, las **funciones** pueden tener tipos de datos especificados tanto para sus parámetros como para su valor de retorno.

Las **funciones** también pueden tener parámetros opcionales y valores predeterminados, parámetros rest, sobrecarga de funciones y sobrecarga de constructores.

2. Declaración de funciones

En TypeScript, las **funciones se declaran** utilizando:

1. la palabra clave `function`
2. seguida del nombre de la función
3. una lista de parámetros
4. el tipo de retorno (opcional).

```
function greet(firstName: string): string {
    return `Hello, ${firstName}!`;
}
```

3. Expresiones de función y funciones flecha

Además de las funciones declaradas, TypeScript también admite **expresiones de función** y **funciones flecha** (arrow functions).

Las funciones flecha tienen una sintaxis más corta y comparten el mismo ámbito léxico que su entorno circundante.

```
const greet = function (firstName: string): string {
  return `Hello, ${firstName}!`;
};

const greetArrow = (firstName: string): string => `Hello, ${firstName}!`;
```

4. Parámetros con valores predefinidos

En TypeScript, los **parámetros** de función pueden tener **valores predeterminados**. Los valores predeterminados se asignan utilizando el operador de igual (`=`).

```
function greet(firstName: string, greeting: string = 'Hello'): string {
  return `${greeting}, ${firstName}!`;
}

console.log(greet('John')); // "Hello, John!"
console.log(greet('John', 'Hi')); // "Hi, John!"
```

5. Parámetros opcionales

Los **parámetros opcionales** se indican con un signo de interrogación (?) después del nombre del parámetro.

En TypeScript, es posible hacer que algunos parámetros de una función sean opcionales, lo que significa que no es necesario proporcionar un valor para ellos al llamar a la función.

Para indicar que un parámetro es opcional, se utiliza el símbolo ? antes del tipo de dato del parámetro.

```
function greet(firstName: string, age?: number): string {
    if (age === undefined) {
        return `Hello, ${firstName}!`;
    } else {
        return `Hello, ${firstName}! You are ${age} years old.`;
    }
}
console.log(greet('John'));
console.log(greet('Jane', 25));
```

En este ejemplo, la función 'greet' toma dos parámetros: 'name' y 'age'. El parámetro 'age' es opcional, como se indica por el símbolo '?' antes de su tipo de dato. Cuando se llama a la función 'greet', es posible proporcionar solo el parámetro 'name' o ambos parámetros. La salida será:

```
Hello, John!
Hello, Jane! You are 25 years old.
```

5.1. Comprobaciones cuando un parámetro es opcional

Cuando un **parámetro es opcional**, es posible que no se proporcione un valor al llamar a la función. Por lo tanto, antes de usar un parámetro opcional, es importante verificar si se le ha asignado un valor o no.

Si no se pasa un argumento para un parámetro opcional en TypeScript, el valor predeterminado del parámetro es `undefined`.

Existen varias formas de **comprobar la existencia** de un parámetro opcional antes de usarlo para

evitar errores:

Comprobar si el parámetro es `undefined`:

```
function myFunction(param?: string): void {
    if (param !== undefined) {
        // Utilizar el parámetro aquí
    }
}
```

Utilizar el operador de **coalescencia nula** (`??`) para proporcionar un valor predeterminado si el parámetro es `null` o `undefined`:

```
function myFunction(param?: string): void {
    const validParam = param ?? 'default_value';
    // Utilizar validParam aquí
}
```

Utilizar el **operador condicional ternario** para verificar y proporcionar un valor predeterminado si el parámetro es `null` o `undefined`:

```
function myFunction(param?: string): void {
    const validParam = param != null ? param : 'default_value';
    // Utilizar validParam aquí
}
```

Utilizar un **bloque try-catch** si la operación que intentas realizar con el parámetro opcional puede generar un error:

```
function myFunction(param?: string): void {
    try {
        // Utilizar el parámetro aquí
    } catch (error) {
        // Manejar el error aquí
    }
}
```

Estas formas de comprobación permiten manejar correctamente parámetros opcionales y evitar errores al intentar utilizarlos en situaciones en las que no se les ha asignado un valor válido.

5.2. Restricciones al usar parámetros opcionales

Es importante tener en cuenta algunas restricciones al utilizar parámetros opcionales en TypeScript:

- Los parámetros opcionales deben seguir a los parámetros requeridos en la lista de parámetros de una función. No puedes tener un parámetro requerido después de un parámetro opcional.

```
// Esto es incorrecto
function incorrectExample(age?: number, name: string) {
// ...
}
// Esto es correcto
function correctExample(name: string, age?: number) {
// ...
}
```

- Si una función tiene múltiples parámetros opcionales, puedes omitir algunos de ellos al llamar a la función, pero debes proporcionar 'undefined' para los parámetros opcionales que desees omitir en medio de la lista.

```
function example(firstName: string, middleName?: string, lastName?: string) {
// ...
}
example('John', undefined, 'Doe'); // Omitir 'middleName'
```

Utilizar parámetros opcionales en funciones de TypeScript permite mayor flexibilidad en la forma en que se pueden llamar las funciones y simplifica el código al evitar la necesidad de proporcionar argumentos innecesarios o predeterminados. Sin embargo, asegúrate de seguir las restricciones mencionadas para mantener la consistencia y la legibilidad del código.

6. Parámetros rest

Los **parámetros rest** permiten aceptar un **número variable** de argumentos en una función y tratarlos como un array.

Para utilizar los **parámetros rest**, se utiliza el operador de propagación (···) antes del nombre del parámetro.

```
function sum(...numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4)); // 10
```

7. Sobrecarga de funciones

La **sobrecarga de funciones** permite tener **múltiples versiones** de una función con diferentes tipos de parámetros o números de parámetros.

En TypeScript, se logra definiendo las firmas de las funciones y luego implementando una función compatible con todas las firmas.

```
function add(a: number, b: number): number;

function add(a: string, b: string): string;

function add(a: number | string, b: number | string): number | string {
  if (typeof a === 'number' && typeof b === 'number') {
    return a + b;
  } else if (typeof a === 'string' && typeof b === 'string') {
    return a.concat(b);
  } else {
    throw new Error('Invalid argument types');
  }
}

console.log(add(1, 2)); // 3
console.log(add('Hello', 'World')); // "HelloWorld"
```

8. Sobrecarga de constructores

En TypeScript, las clases también pueden tener **múltiples constructores sobrecargados**.

Sin embargo, a diferencia de otros lenguajes de programación, no se permite declarar múltiples implementaciones del constructor.

En su lugar, se debe declarar **un único constructor con parámetros opcionales y valores predeterminados**, y luego se puede utilizar la lógica interna para manejar diferentes firmas de constructor.

```
class Rectangle {
  width: number;
  height: number;

  constructor();
  constructor(width: number, height: number);
  constructor(width?: number, height?: number) {
    this.width = width || 0;
    this.height = height || 0;
  }
}

const rect1 = new Rectangle();
const rect2 = new Rectangle(10, 20);
```

9. Funciones puras e impuras

Las funciones se pueden clasificar en dos categorías según si tienen efectos secundarios o no: funciones **puras** e **impuras**.

- **Funciones puras**: Una función es pura si, para los mismos argumentos de entrada, siempre devuelve el mismo resultado y no tiene efectos secundarios. Las funciones puras son más predecibles y más fáciles de probar y depurar.

```
function sum(a: number, b: number): number {
  return a + b;
}
```

- **Funciones impuras**: Una función es impura si tiene efectos secundarios, como modificar variables externas, realizar operaciones de entrada/salida o cambiar el estado del programa. Las funciones impuras pueden ser más difíciles de razonar y probar.

```
let counter = 0;

function increment(): number {
  return ++counter;
}
```

10. Funciones de orden superior y funciones de devolución de llamada

Las **funciones de orden superior** son funciones que aceptan otras funciones como argumentos o devuelven funciones como resultado. Las funciones de devolución de llamada son funciones que se pasan como argumento a otras funciones y se invocan en un momento posterior.

```
function filter<T>(
  items: T[],
  predicate: (item: T, index: number, items: T[]) => boolean
): T[] {
  const result: T[] = [];

  for (let i = 0; i < items.length; i++) {
    if (predicate(items[i], i, items)) {
      result.push(items[i]);
    }
  }

  return result;
}
```

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = filter(numbers, (num) => num % 2 === 0);
```

Las **funciones de orden superior** son aquellas que toman otras funciones como argumentos o devuelven funciones como resultado.

Las funciones que se pasan como argumentos a funciones de orden superior se denominan **funciones de devolución de llamada (callback)**.

En TypeScript, puedes aprovechar las funciones de orden superior y las funciones de devolución de llamada para crear código más modular y flexible.

10.1. Ejemplo de función de orden superior

Aquí hay un ejemplo de una función de orden superior que toma una función de devolución de llamada como argumento:

```
function processArray(array: number[], callback: (item: number) => void): void {
    for (const item of array) {
        callback(item);
    }
}

function printSquare(item: number): void {
    console.log(item * item);
}
const numbers = [1, 2, 3, 4, 5];
processArray(numbers, printSquare);

// Resultado:
/*
1
4
9
16
25
 */
```

En este ejemplo, la función 'processArray' es una función de orden superior que toma un arreglo y una función de devolución de llamada como argumentos.

La función 'printSquare' es una función de devolución de llamada que se pasa a 'processArray'.

10.2. Ejemplo de función de orden superior que devuelve una función

Aquí hay un ejemplo de una función de orden superior que devuelve una función:

```
function createMultiplier(multiplier: number): (num: number) => number {
    return (num: number) => {
        return num * multiplier;
    };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

En este ejemplo, la función 'createMultiplier' es una función de orden superior que devuelve una función que multiplica un número dado por un factor especificado.

11. Ejemplo 1: Declaración de funciones

En este ejercicio, practicaremos la declaración de funciones básicas en TypeScript.

```
function greet(): void {
    console.log('Hello, World!');
}
greet();
```

En este ejemplo, declaramos una función llamada 'greet' que muestra un mensaje en la consola. Luego, llamamos a la función 'greet'.

12. Ejemplo 2: Funciones y parámetros

En este ejercicio, practicaremos el uso de funciones con parámetros.

```
function add(a: number, b: number): number {
    return a + b;
}
const sum = add(5, 3);
console.log('Sum:', sum);
```

En este ejemplo, declaramos una función llamada 'add' que toma dos parámetros 'a' y 'b' y devuelve la suma de ambos. Luego, llamamos a la función 'add' y mostramos el resultado en la consola.

13. Ejemplo 3: Sobrecarga de funciones

En este ejercicio, practicaremos la sobrecarga de funciones en TypeScript.

En TypeScript, a diferencia de Java, no se puede tener diferentes implementaciones para una misma función sobrecargada. En su lugar, se pueden declarar varias funciones con el mismo nombre, pero con diferentes tipos de parámetros.

```
function print(value: string): void;
function print(value: number): void;
function print(value: boolean): void;

function print(value: any): void {
    if (typeof value === 'string') {
        console.log(`String: ${value}`);
    } else if (typeof value === 'number') {
        console.log(`Number: ${value}`);
    } else if (typeof value === 'boolean') {
        console.log(`Boolean: ${value}`);
    } else {
        console.log(`Unknown type: ${value}`);
    }
}

print('Hello');
print(42);
print(true);
```

En este ejemplo, declaramos una función 'print' con sobrecarga, lo que significa que puede tomar diferentes tipos de parámetros. La función 'print' muestra el tipo y el valor del parámetro en la consola. Luego, llamamos a la función 'print' con diferentes tipos de argumentos.

14. Ejemplo 4: Calcular el área de diferentes formas geométricas

Ejemplo de sobrecarga de funciones.

```
// Sobrecarga para calcular el área de un círculo
function calculateArea(radius: number): number;

// Sobrecarga para calcular el área de un rectángulo
function calculateArea(length: number, width: number): number;

// Sobrecarga para calcular el área de un triángulo
function calculateArea(base: number, height: number, triangle: boolean): number;

// Implementación de la función calculateArea
function calculateArea(a: number, b?: number, c?: boolean): number {
    if (b === undefined) {
        // Cálculo del área de un círculo
        return Math.PI * a * a;
```

```
    } else if (c === undefined) {
        // Cálculo del área de un rectángulo
        return a * b;
    } else {
        // Cálculo del área de un triángulo
        return 0.5 * a * b;
    }
}

const circleArea = calculateArea(5);
const rectangleArea = calculateArea(4, 6);
const triangleArea = calculateArea(3, 7, true);

console.log('Circle area:', circleArea);
console.log('Rectangle area:', rectangleArea);
console.log('Triangle area:', triangleArea);
```

15. Ejemplo 5: Funciones puras

Las funciones puras son aquellas que no tienen efectos secundarios y siempre devuelven el mismo resultado para los mismos argumentos. A continuación, se muestra un ejemplo de una función pura:

```
function square(x: number): number {
    return x * x;
}
const result = square(5);
console.log('Result:', result); // Result: 25
```

En este ejemplo, la función 'square' es pura porque no tiene efectos secundarios y siempre devuelve el mismo resultado para los mismos argumentos.

16. Ejemplo 6: Funciones impuras

Las funciones impuras son aquellas que tienen efectos secundarios, como modificar variables fuera de su alcance o interactuar con entradas y salidas.

A continuación, se muestra un ejemplo de una función impura:

```
let counter = 0;

function increment(): void {
counter++;
}
console.log('Counter before increment:', counter); // Counter before increment: 0
increment();
```

```
console.log('Counter after increment:', counter); // Counter after increment: 1
```

En este ejemplo, la función 'increment' es impura porque modifica una variable fuera de su alcance, lo que provoca un efecto secundario.

17. Ejemplo 7: Funciones de orden superior

Las funciones de orden superior son aquellas que toman otras funciones como argumentos o devuelven funciones como resultado.

A continuación, se muestra un ejemplo de una función de orden superior que toma una función de devolución de llamada como argumento:

```
function mapArray(array: number[], callback: (item: number) => number): number[] {
    const result: number[] = [];
    for (const item of array) {
        result.push(callback(item));
    }
    return result;
}

function multiplicarPor2(item: number): number {
    return item * 2;
}

const numbers = [1, 2, 3, 4, 5];
const doubledNumbers = mapArray(numbers, multiplicarPor2);
console.log('Doubled numbers:', doubledNumbers); // Doubled numbers: [ 2, 4, 6, 8, 10
]
```

En este ejemplo, la función 'mapArray' es una función de orden superior que toma un arreglo y una función de devolución de llamada como argumentos. La función 'double' es una función de devolución de llamada que se pasa a 'mapArray'. Luego, llamamos a 'mapArray' con la función 'double' y mostramos el resultado en la consola.

Estos ejercicios te ayudarán a familiarizarte con las funciones puras e impuras y las funciones de orden superior en TypeScript. Puedes seguir practicando con diferentes ejemplos y casos de uso para mejorar tus habilidades en TypeScript.