UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

Programación Genérica

¿Alguna vez han sentido que están escribiendo el mismo código una y otra vez, solo cambiando el tipo de dato?

Por ejemplo, una función para buscar un número en un array, luego otra casi idéntica para buscar un string, y otra para buscar un objeto...

```typescript
// Tenemos que escribir esto...
function buscarNumero(arr: number[], valor: number): number {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === valor) return i;
    }
    return -1;
}

// Y luego esto...
function buscarString(arr: string[], valor: string): number {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === valor) return i;
    }
    return -1;
}

// Y esto también...
function buscarBoolean(arr: boolean[], valor: boolean): number {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === valor) return i;
    }
    return -1;
}
```

Otra vez hablamos de **patrón**... Algo se está repitiendo y es obvio.

Y como en todos los conceptos anteriores, estamos buscando encapsular comportamientos de alguna forma, en este caso notamos que existe un recorrido que se esta repitiendo en cada una de las funciones, el detalle es que en cada una de ellas estamos trabajando con datos distintos, encapsular esto parece imposible... o no?

Existe una solución elegante para este tipo de problemas.

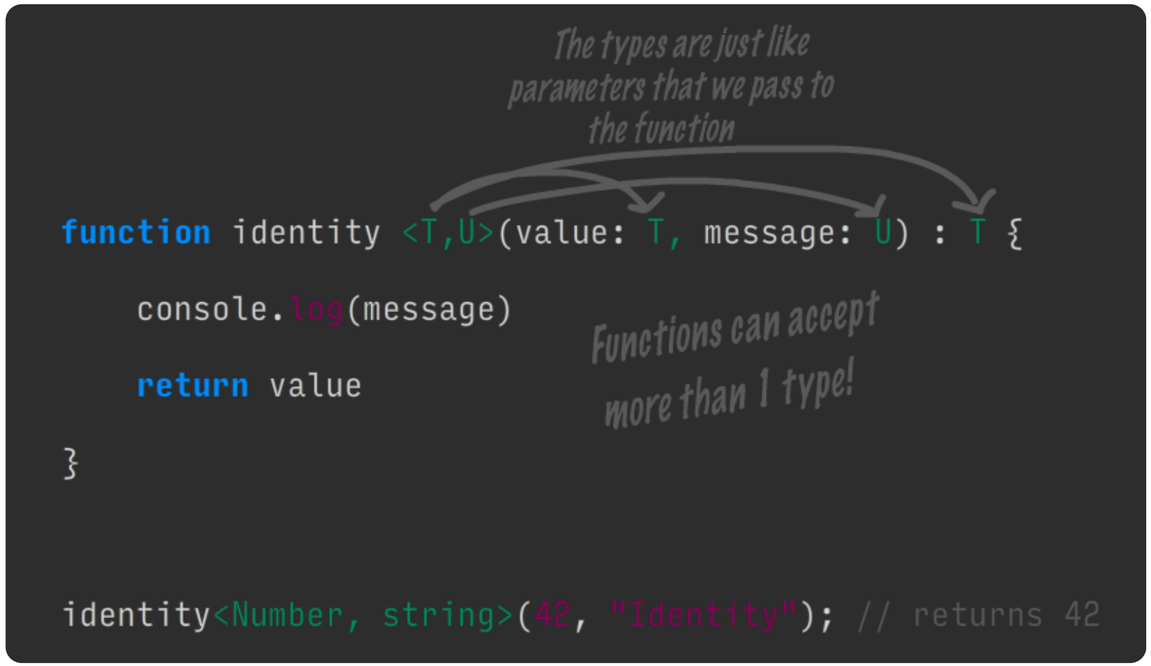
```typescript
// UNA sola función que funciona con CUALQUIER tipo
function buscar<T>(arr: T[], valor: T): number {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === valor) return i;
    }
    return -1;
}

// Ahora la usamos con lo que sea:
buscar<number>([1, 2, 3], 2);       // Funciona
buscar<string>(["a", "b"], "b");    // Funciona
buscar<boolean>([true, false], false); // Funciona
```

Esta **<T>** es como decir: 'No sé qué tipo vas a usar, pero sea lo que sea, lo voy a manejar consistentemente'.

La **T** es solo un nombre, podríamos llamarla **Tipo**, **X**, lo que sea, pero por convención usamos **T**.

Se puede ver como algo así:

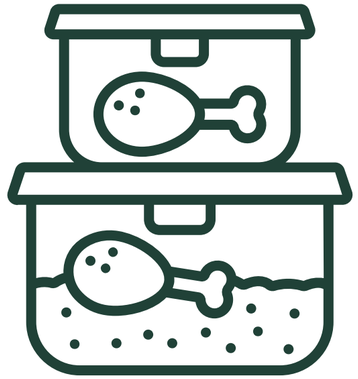
```
// Cuando:
buscar<number>(...)
// T = number y la funcion buscar se ve como:
function buscar(arr: number[], valor: number): number {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === valor) return i;
    }
    return -1;
}
```



```ts
function identity <T>(value: T) : T {
  return value
}


console.log(identity<Number>(42))
console.log(identity("Hello!"))
console.log(identity<Number>([1,2,3]))
```

Esto parece un **Tupperware**:

- No importa si le pones arroz, pasta, ensalada o frutas
- El contenedor funciona igual con todos
- No necesitas un contenedor especial para cada tipo de comida
- Pero una vez que lo llenas, el tipo queda definido (no mezclas arroz con helado)



Podemos empezar con ejemplos básicos:

Función Simple

```typescript
// Función que devuelve lo que recibe (identidad)
function identidad<T>(valor: T): T {
    return valor;
}

// Uso explícito del tipo
const num = identidad<number>(42);         // num es number
const texto = identidad<string>("Hola");  // texto es string

// TypeScript puede inferir el tipo
const auto = identidad(100);  // TypeScript sabe que es number
```

Array de Cualquier Tipo

```
// Función que devuelve el primer elemento
function primero<T>(arr: T[]): T | undefined {
    return arr.length > 0 ? arr[0] : undefined;
}

primero([1, 2, 3]);         // devuelve number
primero(["a", "b"]);        // devuelve string
primero([true, false]);     // devuelve boolean
```

Y aunque no lo crean, ya han usado tipos genéricos, solo que como TypeScript puede **inferir el tipo** gracias a los parámetros de entrada, no es necesario escribirlo de manera explicita.

```
function map<TElement, TResult>(
    items: TElement[],
    mappingFunction: (item: TElement) => TResult
): TResult[] {
    // ...
}
```

Ejercicio:

¿Cómo escribirías una función genérica que intercambie dos valores?

Solucion:

```ts
function intercambiar<T>(a: T, b: T): [T, T] {
    return [b, a];
}

const [x, y] = intercambiar(10, 20);          // [20, 10]
const [p, q] = intercambiar("Hola", "Mundo"); // ["Mundo", "Hola"]
```

Los genéricos no solo existen en funciones, también existen en tipos, clases, interfaces....

```typescript
// Nuestro Tupperware
class Box<T> {
    private value: T;

    constructor(value: T) {
        this.value = value;
    }

    getValue(): T {
        return this.value;
    }
}

let box = new Box<number>(42);
console.log(box.getValue()); // 42
```

```ts
interface GenericInterface<U> {
  value: U
  getIdentity: () => U
}

class IdentityClass<T> implements GenericInterface<T> {
  value: T

  constructor(value: T) {
    this.value = value
  }

  getIdentity () : T {
    return this.value
  }

}

const myNumberClass = new IdentityClass<Number>(1)
console.log(myNumberClass.getIdentity()) // 1

const myStringClass = new IdentityClass<string>("Hello!")
console.log(myStringClass.getIdentity()) // Hello!
```

Si no te resulta tan claro, intenta rastrear los valores de tipo (**type values**) subiendo por la cadena de llamadas a funciones.

Funciona de la siguiente manera:

1. Instanciamos una nueva instancia de IdentityClass, pasando Number y 1.
2. En la clase IdentityClass, T se convierte en Number.
3. IdentityClass implementa GenericInterface<T> y sabemos que T es Number, por lo que es como si estuviéramos implementando GenericInterface<T>.
4. En GenericInterface, U se convierte en Number.

Los nombres de variables son diferentes aquí para mostrar que el valor de tipo se propaga por la cadena y que el nombre de la variable no importa.

Incluso podemos ir mas alla.

¿Que pasa si queremos restringir nuestros genéricos?

TypeScript te permite restringir qué tipos pueden usarse mediante la palabra clave **extends**.

Por ejemplo, imagina que tienes un sistema de inventario donde solo quieres almacenar productos que tengan un **precio**.

```TypeScript
// Función genérica sin restricciones
function calcularTotal<T>(items: T[]): number {
    return items.reduce((total, item) => {
        return total + item.precio; //ERROR: 'precio' no existe en tipo 'T'
    }, 0);
}
```

Debemos agregar restricciones...

```typescript
// Definimos la interfaz que establece el contrato
interface Valuable {
    precio: number;
}

// Restringimos T para que DEBA tener 'precio'
function calcularTotal<T extends Valuable>(items: T[]): number {
    return items.reduce((total, item) => {
        return total + item.precio; // sabemos que 'precio' existe
    }, 0);
}

// Creamos tipos que cumplen el contrato
interface Producto {
    nombre: string;
    precio: number;
}

interface Servicio {
    descripcion: string;
    precio: number;
    duracion: number;
}

// Uso de la función
const productos: Producto[] = [
    { nombre: "Laptop", precio: 1000 },
    { nombre: "Mouse", precio: 25 }
];
```

```typescript
const servicios: Servicio[] = [
    { descripcion: "Consultoría", precio: 500, duracion: 2 },
    { descripcion: "Mantenimiento", precio: 200, duracion: 1 }
];

console.log(calcularTotal(productos));  // 1025
console.log(calcularTotal(servicios));  // 700

// Esto daría error de compilación:
const cosas = [
    { color: "rojo" }, // No tiene 'precio'
    { tamaño: "grande" }
];
// calcularTotal(cosas); // ERROR: tipo no compatible
```

"Puedo recibir un array de diferentes tipos de cosas, siempre y cuando me garantices que cada una de esas cosas tendrá una propiedad precio."

Por lo tanto tenemos:

1. Flexibilidad: La función sigue siendo genérica. Funciona con Producto, con Servicio, y funcionará con cualquier tipo futuro que creemos, siempre que cumpla el contrato.
2. Seguridad (Correctitud): El compilador sabe que cualquier cosa que entre a la función tendrá sí o sí un .precio, por lo que la operación total + item.precio es 100% segura.

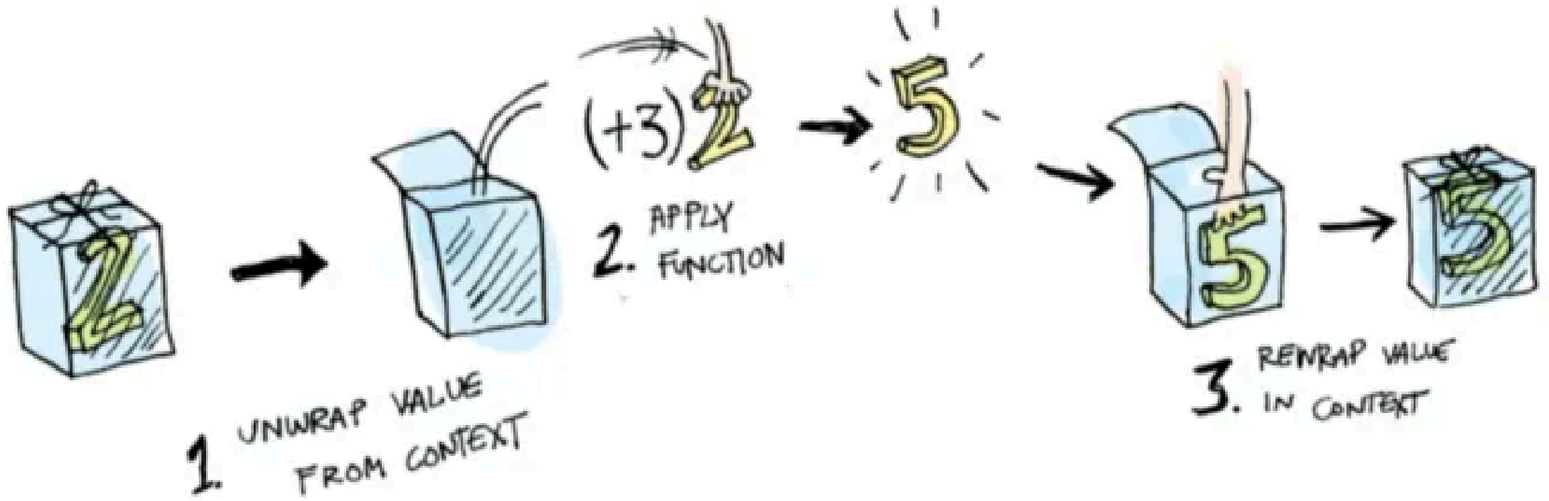
Hay usos mucho mas simples, pero también útiles.

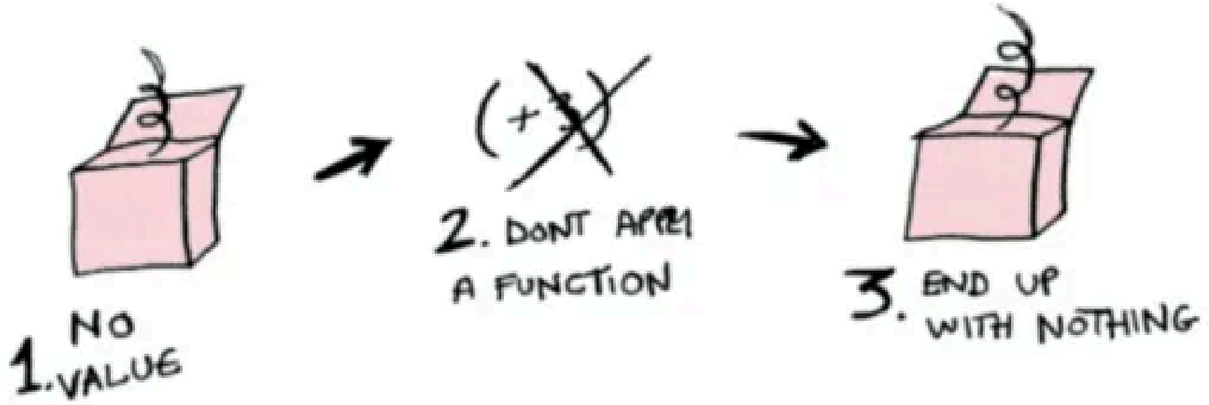
En este caso, la lógica de manejar la ausencia de un valor es independiente del tipo real del valor. Tenemos un tipo genérico **Optional** que puede almacenar cualquier otro tipo, ya que manejará cualquier cosa de la misma manera.

Pensando un poco, podemos notar que **Optional** está en una dimensión completamente diferente a **T** (el tipo genérico), ya que cualquier cambio que hagamos a **Optional** no afecta a **T**, y cualquier cambio hecho a **T** no afecta a **Optional**.

Este aislamiento es una característica extremadamente poderosa de la programación genérica.

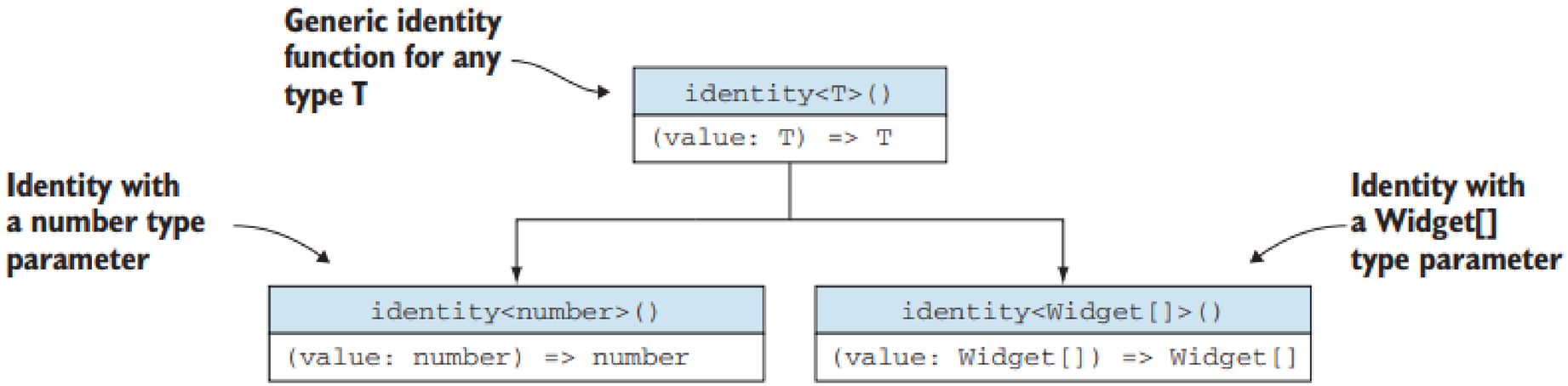
```
class Optional<T> {
    private value: T | undefined;
    private assigned: boolean;

    constructor(value?: T) {
        if (value) {
            this.value = value;
            this.assigned = true;
        } else {
            this.value = undefined;
            this.assigned = false;
        }
    }

    hasValue(): boolean {
        return this.assigned;
    }

    getValue(): T {
        if (!this.assigned) throw Error();
        return <T>this.value;
    }
}
```


(+3)
1. UNWRAP VALUE FROM CONTEXT
2. APPLY FUNCTION
3. REWRAP VALUE IN CONTEXT


1. NO VALUE
2. DONT APPLY A FUNCTION
3. END UP WITH NOTHING

Sabiendo esto... ¿Como definiríamos a un tipo genérico?

Un tipo genérico es una función genérica, clase, interfaz, etc., que está parametrizada sobre uno o más tipos. Los tipos genéricos nos permiten escribir código general que funciona con diferentes tipos, posibilitando un **alto nivel de reutilización de código**.



Los tipos genericos estan en muchisimos lados, y por eso su entendimiento es tan importante.
Pensemos en algo que hacemos todos los días.

¿Qué estructuras de datos almacenen una *secuencia* de cosas?

¿Qué tienen todas estas estructuras en común?

¿Qué es lo que podemos hacer con *todas* ellas para revisarlas elemento por elemento?

Los tipos genericos estan en muchisimos lados, y por eso su entendimiento es tan importante.
Pensemos en algo que hacemos todos los días.

¿Qué estructuras de datos almacenen una *secuencia* de cosas?

Array, listas, strings...

¿Qué tienen todas estas estructuras en común?

¿Qué es lo que podemos hacer con *todas* ellas para revisarlas elemento por elemento?

un bucle for, recorrerlas, iterar

```
let miArray: number[] = [1, 2, 3];
let miString: string = "HOLA";
let miSet: Set<boolean> = new Set([true, false]);

// Este MISMO bucle...
for (const item of miArray) { /* item es number */ }
for (const item of miString) { /* item es string */ }
for (const item of miSet) { /* item es boolean */ }
```

¿Alguna vez se han preguntado **cómo** es que el **for...of** "sabe" cómo recorrer un array (que es una lista), un string (que es una secuencia de caracteres) y un Set (que no tiene orden)?

La respuesta es que todas estas estructuras cumplen un **contrato**. Ese contrato se llama **Iterable.**

Iterable (El Contenedor): Es la estructura de datos. Es la "caja" (el array, el string). Su *único* trabajo es decir: "¡Oye, yo sé cómo crear un Iterador para mí mismo!".

Iterator (El Recorredor): No es la colección. Es un objeto separado, una especie de "vista paso a paso" sobre la colección. Su trabajo es tener un método **.next()** que, cada vez que lo llamas, te da el siguiente valor hasta que se acaba.

Piensen en un mazo de cartas (el **Iterable**). El **Iterador** es su mano, que saca una carta a la vez (**.next()**) hasta que el mazo se vacía.

El bucle **for...of** es simplemente un consumidor que le pide el **Iterador** al **Iterable** y luego llama **.next()**, **.next()**, **.next()**... por nosotros.

Cuando ustedes escriben

```
let miArray: number[] = [1, 2, 3]
```

¿Qué es **number[]**? Es solo azúcar sintáctica para **Array<number>.**

¡Han estado usando tipos genéricos todo este tiempo!

- **Array<T>** es un **Iterable Genérico**.
- Cuando **T** es **number**, su iterador nos da **number**.
- Cuando **T** es **string**, su iterador nos da **string**.

El bucle **for...of** es, en sí mismo, un **algoritmo genérico**. No le importa qué está iterando, solo le importa que la cosa **sea** un iterable.

Fin