¿Qué son los Genéricos?

Los genéricos son una característica fundamental de TypeScript que nos permite escribir código flexible y reutilizable que puede trabajar con múltiples tipos de datos. En lugar de estar limitados a un tipo específico, los genéricos nos permiten usar un "tipo placeholder" que se concreta en tiempo de compilación.

1. Funciones Genéricas Básicas

Veamos primero el problema que resuelven los genéricos:

```typescript
// Sin genéricos - Necesitamos una función para cada tipo
function firstElementNumber(arr: number[]): number {
    return arr[0];
}

function firstElementString(arr: string[]): string {
    return arr[0];
}

// Con genéricos - Una sola función que trabaja con cualquier tipo
function firstElement<T>(arr: T[]): T {
    return arr[0];
}
```

En el ejemplo anterior, `T` es un tipo genérico que se puede reemplazar con cualquier tipo cuando llamamos a la función:

```typescript
// Uso explícito del tipo
const primerNumero = firstElement<number>([1, 2, 3]);     // tipo: number
const primerTexto = firstElement<string>(["a", "b", "c"]); // tipo: string

// TypeScript puede inferir el tipo
const inferido = firstElement([true, false]);             // tipo: boolean
```

2. Interfaces Genéricas

Las interfaces genéricas nos permiten crear contratos que pueden trabajar con diferentes tipos:

```typescript
interface Container<T> {
    value: T;
    getValue(): T;
}

// Implementación para números
class NumberContainer implements Container<number> {
    constructor(public value: number) {}

    getValue(): number {
        return this.value;
    }
}
```

```typescript
// Uso
const numeroContenido = new NumberContainer(42);
console.log(numeroContenido.getValue()); // 42
```

3. Restricciones (Constraints)

Podemos limitar los tipos que pueden usar nuestros genéricos mediante constraints:

```typescript
// Definimos una interface que requiere una propiedad length
interface HasLength {
    length: number;
}

// T debe ser un tipo que tenga la propiedad length
function logLength<T extends HasLength>(item: T): number {
    return item.length;
}

// Ejemplos válidos
console.log(logLength("Hola"));           // 4 (string tiene length)
console.log(logLength([1, 2, 3]));        // 3 (arrays tienen length)
console.log(logLength({length: 10}));     // 10 (objeto con length)

// Esto NO compilaría:
// console.log(logLength(123)); // Error: number no tiene length
```

4. Múltiples Parámetros de Tipo

Podemos usar múltiples tipos genéricos en una misma función o clase:

```typescript
function merge<T extends object, U extends object>(obj1: T, obj2: U): T & U {
    return { ...obj1, ...obj2 };
}

// Ejemplo de uso
const persona = merge(
    { nombre: "Juan" },
    { edad: 30 }
);
// Resultado: { nombre: "Juan", edad: 30 }
```

5. Tipos Genéricos Utilitarios

TypeScript incluye varios tipos utilitarios genéricos muy útiles:

```typescript
// Partial: hace todas las propiedades opcionales
interface Todo {
    title: string;
    description: string;
    completed: boolean;
}

// Partial<Todo> es equivalente a:
```

```
// {
//     title?: string;
//     description?: string;
//     completed?: boolean;
// }
function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
    return { ...todo, ...fieldsToUpdate };
}

// Record: crea un objeto con claves de un tipo y valores de otro
type PageInfo = Record<string, number>;
const pages: PageInfo = {
    home: 1,
    about: 2,
    contact: 3
};

// Pick: selecciona ciertas propiedades de un tipo
type TodoPreview = Pick<Todo, "title" | "completed">;
// Equivalente a:
// {
//     title: string;
//     completed: boolean;
// }
```

6. Ejemplo Práctico: Implementación de una Stack Genérica

Veamos un ejemplo completo de una estructura de datos genérica:

```
class Stack<T> {
    private items: T[] = [];

    push(item: T): void {
        this.items.push(item);
    }

    pop(): T | undefined {
        return this.items.pop();
    }

    peek(): T | undefined {
        return this.items[this.items.length - 1];
    }

    isEmpty(): boolean {
        return this.items.length === 0;
    }

    size(): number {
        return this.items.length;
    }
}

// Ejemplo de uso con números
```

```typescript
const stackNumeros = new Stack<number>();
stackNumeros.push(1);
stackNumeros.push(2);
stackNumeros.push(3);
console.log(stackNumeros.pop()); // 3
console.log(stackNumeros.peek()); // 2
console.log(stackNumeros.size()); // 2

// Ejemplo de uso con strings
const stackTextos = new Stack<string>();
stackTextos.push("TypeScript");
stackTextos.push("es");
stackTextos.push("genial");
console.log(stackTextos.pop()); // "genial"
```

Ejercicios Prácticos

Ejercicio 1: Implementar una Cola (Queue) Genérica

```typescript
// Implementa una cola genérica con los métodos:
// - enqueue(item: T): void
// - dequeue(): T | undefined
// - peek(): T | undefined
// - isEmpty(): boolean
// - size(): number

class Queue<T> {
    // Tu implementación aquí
}
```

Ejercicio 2: Función de Filtrado Genérico

```typescript
// Implementa una función filter genérica que acepte un array
// y una función predicado para filtrar elementos

function filter<T>(arr: T[], predicate: (item: T) => boolean): T[] {
    // Tu implementación aquí
}
```

Ejercicio 3: Par Clave-Valor Genérico

```typescript
// Implementa una interfaz y clase KeyValuePair que pueda
// trabajar con cualquier tipo de clave y valor

interface IKeyValuePair<K, V> {
    // Tu implementación aquí
}

class KeyValuePair<K, V> implements IKeyValuePair<K, V> {
    // Tu implementación aquí
}
```

Soluciones a los Ejercicios

Solución Ejercicio 1: Queue

```
class Queue<T> {
    private items: T[] = [];

    enqueue(item: T): void {
        this.items.push(item);
    }

    dequeue(): T | undefined {
        return this.items.shift();
    }

    peek(): T | undefined {
        return this.items[0];
    }

    isEmpty(): boolean {
        return this.items.length === 0;
    }

    size(): number {
        return this.items.length;
    }
}
```

Solución Ejercicio 2: Filter

```
function filter<T>(arr: T[], predicate: (item: T) => boolean): T[] {
    const result: T[] = [];
    for (const item of arr) {
        if (predicate(item)) {
            result.push(item);
        }
    }
    return result;
}

// Ejemplo de uso:
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = filter(numbers, (n) => n % 2 === 0);
console.log(evenNumbers); // [2, 4]
```

Solución Ejercicio 3: KeyValuePair

```
interface IKeyValuePair<K, V> {
    key: K;
    value: V;
    getKey(): K;
    getValue(): V;
}
```

```
class KeyValuePair<K, V> implements IKeyValuePair<K, V> {
    constructor(public key: K, public value: V) {}

    getKey(): K {
        return this.key;
    }

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

// Ejemplo de uso:
const par = new KeyValuePair<string, number>("edad", 30);
console.log(par.getKey());   // "edad"
console.log(par.getValue()); // 30
```

Puntos Clave a Recordar

1. Los genéricos nos permiten escribir código reutilizable que funciona con múltiples tipos.
2. La sintaxis básica usa `<T>` donde T es un placeholder para cualquier tipo.
3. Podemos usar constraints (`extends`) para limitar los tipos permitidos.
4. Las interfaces y clases pueden ser genéricas.
5. TypeScript puede inferir tipos genéricos en muchos casos.
6. Los tipos utilitarios genéricos (Partial, Record, Pick) son herramientas poderosas.

Mejores Prácticas

1. Usa nombres descriptivos para los tipos genéricos:
    - `T` para tipos generales
    - `K` para tipos de llaves
    - `V` para tipos de valores
    - `E` para tipos de elementos
2. Utiliza constraints cuando necesites garantizar ciertas propiedades o métodos.
3. Aprovecha la inferencia de tipos cuando sea posible, pero sé explícito cuando mejore la legibilidad.
4. Considera usar tipos utilitarios antes de crear soluciones personalizadas.

```
1   // Definimos una interface genérica que describe un objeto con una propiedad id
2   interface IDetail<T> {
3     id: T;
4     }
5
6     // Definimos una clase genérica que utiliza el tipo genérico T
7     class Detail<T> implements IDetail<T> {
8     id: T;
9
10    constructor(id: T) {
11    this.id = id;
12    }
13    }
14
15    // Creamos una instancia de la clase con tipo number
16    const detail1 = new Detail<number>(1);
```

```
17
18    // La propiedad id ahora es de tipo number
19    detail1.id;
20
21    // Creamos una instancia de la clase con tipo string
22    const detail2 = new Detail<string>('a');
23
24    // La propiedad id ahora es de tipo string
25
26    detail2.id;
27
28    console.log(detail1.id);
29    console.log(detail2.id);
```

```
1   interface GenericIdentityFn<T> {
2     (arg: T): T;
3   }
4
5   function identity<T>(arg: T): T {
6     return arg;
7   }
8
9   let myIdentity: GenericIdentityFn<number> = identity;
10  let myIdentity2: GenericIdentityFn<String> = identity;
11
12  console.log(myIdentity(25));
13  console.log(myIdentity2('Hello'));
```

interface genérica

función genérica

instancias

De esta forma, en lugar de intentar limitar T a number y string, implementamos la interfcez genérica para asegurarnos que operar() retorne el tipo correcto.

El chequeo de tipos dentro del método también nos permite asegurar que la operación se realiza sólo cuando tiene sentido.

Ejemplo 2

```
1   // Clase genérica
2   interface OperacionInterface {
3     operar(): number | string;
4   }
5
6   class Operacion<T> implements OperacionInterface {
7
8     a: T;
9     b: T;
10
11    constructor(a: T, b: T) {
12      this.a = a;
13      this.b = b;
14    }
```

```
    operar() {
      if(typeof this.a === "number" && typeof this.b === "number") {
        return this.a + this.b;
      } else {
        return String(this.a) + String(this.b);
      }
    }

}

const suma = new Operacion<number>(2, 3);
const concat = new Operacion<string>("Hola ", "Mundo");

console.log(suma.operar()); // 5
console.log(concat.operar()); // Hola Mundo
```

```
// Clase genérica para representar la cuenta
class CuentaBancaria<T extends number> {
  saldo: any;

  constructor(saldoInicial: T) {
    this.saldo = saldoInicial;
  }

  getSaldo() {
    return this.saldo;
  }

  retirar(monto: number) {
    this.saldo -= monto;
  }


  depositar(monto: T) {
    this.saldo += monto;
  }
```

```
}

// Simulamos una cuenta de ahorros con saldo inicial de 5000
const cuentaAhorros = new CuentaBancaria<number>(5000);

// Consultamos el saldo
console.log(cuentaAhorros.getSaldo()); // 5000

// Retiramos 2500
cuentaAhorros.retirar(2500);

// Depositamos 1000
cuentaAhorros.depositar(1000);

// Consultamos nuevamente
console.log(cuentaAhorros.getSaldo()); // 3500
```

```
// Interface genérica para modelo de reserva
interface Reserva<T> {
  id: string;
  huésped: string;
  habitación: T;
}

// Clase para modelar una habitación
class Habitacion {
  numero: number;
  tipo: string;

  constructor(numero: number, tipo: string) {
    this.numero = numero;
    this.tipo = tipo;
  }
}
```

```
// Clase para crear reservas
class ReservaHotel implements Reserva<Habitacion> {
  id: string;
  huésped: string;
  habitación: Habitacion;

  constructor(id: string, huésped: string, habitación: Habitacion) {
    this.id = id;
    this.huésped = huésped;
    this.habitación = habitación;
  }
}
```

```
32    // Usando las clases
33
34    const habitacion101 = new Habitacion(101, 'standard');
35
36    const reserva = new ReservaHotel('R001', 'Juan Perez', habitacion101);
37
38    console.log(reserva.id); // R001
39    console.log(reserva.huésped); // Juan Perez
40    console.log(reserva.habitación.numero); // 101
```

Taller básico de aplicación:

1. Defina una clase Cita que tenga los atributos:

- paciente (nombre del paciente)
- doctor (nombre del doctor)
- fecha (fecha de la cita)
- hora (hora de la cita)
- motivo (motivo de la cita)

Cree una lista de citas inicializada con algunos ejemplos de citas.

Después cree las siguientes funciones (todo es simulado):

- agregar_cita(cita): agrega una nueva cita a la lista
- eliminar_cita(cita): elimina una cita de la lista
- buscar_cita(doctor): devuelve las citas de un doctor dado
- citas_hoy(): devuelve las citas de hoy
- citas_fecha(fecha): devuelve las citas de una fecha dada

Puede probar el funcionamiento con algunos ejemplos, agregando, eliminando y buscando citas y probando las funciones de búsqueda.

2. Defina una clase Producto con los atributos:

- nombre (nombre del producto)
- precio (precio del producto)

Defina una clase CajaRegistradora con los atributos:

- products (lista de productos)
- total (total de la compra)

Agregue métodos a la clase CajaRegistradora:

- cobrar(producto):
- Agrega el producto a la lista de productos
- Suma el precio del producto al total

imprimir_ticket():

- Imprime los productos con sus precios
- Imprime el total de la compra

Puede crear algunos productos de ejemplo y probar el funcionamiento de la caja registradora agregando productos con el método cobrar() y luego imprimiendo el ticket con el método imprimir_ticket().

3. Defina una clase Apartamento con los atributos:
   - número (número de apartamento)
   - habitaciones (número de habitaciones)
   - metros (metros cuadrados)
   - Propietarios (lista de propietarios)

   Defina una clase propietario con los atributos:

   - nombre
   - email

   Defina una clase Edificio con los atributos:

   - dirección
   - Apartamentos (lista de departamentos)

Agregue métodos a la clase Edificio:

- agregar_apartamento(apartamento): agrega un apartamento
- eliminar_apartamento(numero): elimina un apartamento
- buscar_apartamento(numero): busca un departamento
- agregar_propietario(numero, propietario): agrega un propietario a un departamento
- cobrar_renta(): cobra la renta a todos los inquilinos

Puede crear algunos apartamentos, propietarios de ejemplo y probar el funcionamiento agregando apartamentos y propietarios y probando los métodos para buscar, agregar y eliminar.

Esto permite modelar la administración básica de un edificio con apartamentos y propietarios. Se pueden agregar más funcionalidades como calcular cuota de administración, generar reportes, etc. Pero es un buen ejercicio inicial para modelar esta situación con programación orientada a objetos en TypeScript.