05/09/2025

Trabajo Práctico 9 – Programación Generica

Objetivos

- Entender la necesidad de la programación genérica para evitar la duplicación de código.
- Implementar un método genérico para operar sobre arrays de distintos tipos de objetos.
- Diseñar e implementar una clase genérica para crear una estructura de datos reutilizable y con seguridad de tipos.
- Valorar las ventajas de la seguridad de tipos (type safety) que ofrecen los genéricos en tiempo de compilación.

Ejercicio 1: Método Genérico - Una Caja de Herramientas para Arrays
Este ejercicio se centra en crear una utilidad que realice operaciones comunes sobre arrays, sin importar el tipo de dato que contengan.

Paso 1: El Problema (Sin Genéricos)
Imagina que necesitas un método para intercambiar las posiciones de dos elementos en un array. Sin genéricos, tendrías que crear una versión del método para cada tipo de dato: uno para int[], otro para String[], otro para double[], y así sucesivamente. Esto es repetitivo y poco mantenible. Pruebe de realizarlo y compruebelo.

Paso 2: La Solución (Con un Método Genérico)
Vamos a crear una clase de utilidad llamada ArrayUtils que contendrá un único método genérico para resolver este problema de una vez por todas.

```
class ArrayUtils {
    // T es un "parámetro de tipo" que será reemplazado por un tipo real
    public static <T> void intercambiar(T[] array, int indice1, int indice2) {
        // Lógica para validar los índices (opcional pero recomendado)
        if (indice1 < 0 || indice1 >= array.length || indice2 < 0 || indice2 >= array.length) {
            System.out.println("Índices fuera de rango.");
            return;
        }

        T temp = array[indice1];
        array[indice1] = array[indice2];
        array[indice2] = temp;
    }

    // EXTRA: Puedes pedirles que implementen otro método genérico, como imprimir un array.
    public static <T> void imprimirArray(T[] array) {
        System.out.print("[ ");
        for (T elemento : array) {
            System.out.print(elemento + " ");
        }
        System.out.println("]");
    }
}
```

Ahora, en el método main de TestGenericos, pon a prueba tu método genérico con diferentes tipos de arrays.
Java

```
public class TestGenericos {
    public static void main(String[] args) {
        // Prueba con un array de Integers
        Integer[] numeros = {10, 20, 30, 40, 50};
        System.out.println("Array de enteros original:");
        ArrayUtils.imprimirArray(numeros);

        ArrayUtils.intercambiar(numeros, 1, 3); // Intercambia 20 y 40

        System.out.println("Array de enteros después del intercambio:");
        ArrayUtils.imprimirArray(numeros);

        System.out.println("\n--------------------------------\n");

        // Prueba con un array de Strings
        String[] palabras = {"Hola", "Mundo", "desde", "Java"};
        System.out.println("Array de strings original:");
        ArrayUtils.imprimirArray(palabras);

        ArrayUtils.intercambiar(palabras, 0, 2); // Intercambia "Hola" y "desde"

        System.out.println("Array de strings después del intercambio:");
        ArrayUtils.imprimirArray(palabras);
    }
}
```

Para discutir
Permite la programación genérica la reutilización de código y ayuda a evitar el código duplicado?

Ejercicio 2: Clase Genérica - Creando un Contenedor Flexible
Este ejercicio se enfoca en construir una clase que pueda almacenar un par de objetos, que pueden ser de tipos diferentes, garantizando siempre la seguridad de tipos.

Paso 1: El Problema (Sin Genéricos)
Necesitamos una clase para guardar dos valores juntos, como el nombre de un producto (String) y su ID (Integer), o el nombre de un estudiante (String) y su promedio (Double). Una solución "vieja" sería usar Object, pero esto tiene un gran problema: se pierde la información del tipo y se necesitan castings (conversiones explícitas) que pueden fallar en tiempo de ejecución.

```
// Ejemplo del problema con Object (NO HACER ESTO)
Object idProducto = 123;
//...
String idString = (String) idProducto; // ¡ERROR! ClassCastException en tiempo de ejecución
```

Paso 2: La Solución (Con una Clase Genérica)
Crearemos una clase genérica Par<T, U> que contendrá dos elementos de tipos T y U.

```
public class Par<T, U> {
    private T primero;
    private U segundo;

    public Par(T primero, U segundo) {
        this.primero = primero;
        this.segundo = segundo;
    }

    public T getPrimero() {
        return primero;
    }

    public void setPrimero(T primero) {
        this.primero = primero;
    }

    public U getSegundo() {
        return segundo;
    }

    public void setSegundo(U segundo) {
        this.segundo = segundo;
    }

    @Override
    public String toString() {
        return "Par(" + primero + ", " + segundo + ")";
    }
}
```

Para probarla:

```
public static void main(String[] args) {
    // ... (código del ejercicio 1)

    System.out.println("\n--- Ejercicio Clase Genérica ---\n");

    // Creamos un par que asocia un String con un Integer
    Par<String, Integer> producto = new Par<>("Laptop Gamer", 101);
    System.out.println("Producto: " + producto);

    // Creamos un par que asocia un String con un Double
    Par<String, Double> estudiante = new Par<>("Ana Pérez", 9.5);
    System.out.println("Estudiante: " + estudiante);

    // Demostración de la SEGURIDAD DE TIPOS
    Integer idProducto = producto.getSegundo();
    System.out.println("ID recuperado (ya es Integer, no necesita cast): " + idProducto);

    // ¡Esto dará un ERROR DE COMPILACIÓN!
    // El compilador sabe que el segundo elemento debe ser Integer.
    // descomente la siguiente línea para ver el error.
    // producto.setSegundo("ciento-uno");
}
```