UCSM Escuela Profesional de Ingeniería de Sistemas

Septiembre - 2024

INFORME DE PRÁCTICAS

Lenguaje de Programación III

Práctica N°05: Generics en Java

Elaborado por:

Cusirramos Chiri, Santiago Jesús

GRUPO N° 0X

PRÁCTICAS DE COMPUTACIÓN EN RED III

Presentado por:

2023601381	Cusirramos Chiri, Santiago Jesús	100%

RECONOCIMIENTOS

El reconocimiento a los creadores de redes de computadoras es fundamental, ya que su visión y avances han permitido el desarrollo de tecnologías críticas como OSPF, esenciales para la gestión y el enrutamiento eficiente de grandes redes. Su trabajo ha revolucionado la forma en que se interconectan los sistemas, habilitando el monitoreo, la seguridad, y la escalabilidad de redes empresariales en un mundo cada vez más digital. Gracias a ellos, hoy podemos disfrutar de una comunicación fluida y confiable a nivel global.

PALABRAS CLAVES

xxxxx, xxxxxx, xxxxxx, xxxx, xxxxx.

ÍNDICE

1. RESÚMEN

entonces acá en el 2019 tenemos una herramienta que es el dictado y como verán ya está escribiendo a pesar que yo no quiero que escriba.

El lenguaje HTML nos permite crear la estructura de un documento web y presentarlo a los usuarios del mismo, el lenguaje está estructurado de tal forma que nos permite agrupar elementos bajo un solo contexto para ello también podemos hacer uso de formularios web del tipo HTML. y así escriben

2. INTRODUCCIÓN

En el desarrollo de software, uno de los desafíos comunes es la necesidad de crear métodos y clases que funciones con diferentes tipos de datos, seria como tener la posibilidad de tener un único método para todo, desde ordenar los elementos en arreglos de tipos variados, como Integer, String o cualquier otro tipo de dato.

La introducción a los Genéricos en Java alivia esta necesidad , permitiendo a los programadores definir métodos y clases que son flexibles y reutilizables. Con los genéricos, podemos escribir un método y diferentes clases mas flexibles y reutilizables. Con los genéricos, podemos escribir un método que ordene arreglos de diversos tipos (String, Int, Floats, etc), asegurando que los tipos de los elementos sea coherente con el tiempo de compilación. Esto no solo simplifica el código, sino que ayuda a la seguridad al prevenir cierto tipo de errores antes de que el programa se ejecute.

Debemos de utilizar los Genericos ya que si lo comparamos con los métodos sobre cargados (redefinimos el uso de un método para casos especiales) no estamos ahorrando una gran cantidad de código.

Contamos con el siguiente programa, el cual tienen como objetivo imprimir 3 diferentes tipos de arreglos (Integer, Double, Character), para los 3 casos estamos sobrecargando la función **imprimirArreglo** la cual en los 3 casos realiza la misma acción.

En nuestro 1er método debemos mandar un arreglo de INTEGER, posteriormente los imprime y acaba el método, no retorna nada debido a que el método definido es un void.

```
public class MetodosSobrecargados {
    public static void imprimirArreglo( Integer[] arregloEntrada ) {
        for ( Integer elemento : arregloEntrada )
            System.out.printf( "%s ", elemento );
        System.out.println();
    }
```

En nuestro 2do método debemos mandar un arreglo de DOUBLE, posteriormente los imprime y acaba el método, no retorna nada debido a que el método definido es un void.

```
    public static void imprimirArreglo( Double[] arregloEntrada ) {
        for ( Double elemento : arregloEntrada )
            System.out.printf( "%s ", elemento );
        System.out.println();
    }
```

En nuestro 3er método debemos mandar un arreglo de CHARACTER, posteriormente los imprime y acaba el método, no retorna nada debido a que el método definido es un void.

```
    public static void imprimirArreglo( Character[] arregloEntrada ){
        for ( Character elemento : arregloEntrada )
            System.out.printf( "%s ", elemento );
        System.out.println();
    }
```

Finalmente nuestro programa principal realiza la impresión de los 3 metodos, uno detrás del otro.

```
1.      public static void main( String args[] ) {
2.          Integer[] arregloInteger = { 1, 2, 3, 4, 5, 6 };
3.          Double[] arregloDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6,7.7};
4.          Character[] arregloCharacter = { 'H', 'O', 'L', 'A' };
5.          System.out.println( "El arreglo arregloInteger contiene:");
6.
7.          imprimirArreglo( arregloInteger );
8.          System.out.println( "\nEl arreglo arregloDouble contiene:");
9.          imprimirArreglo( arregloDouble );
10.         System.out.println( "\nEl arreglo arregloCharacter contiene:" );
11.         imprimirArreglo( arregloCharacter );
12.     }
13. }
14.
```

Obteniendo como resultado:

```
 MarcoTeorico.MetodosSobrecargados
El arreglo arregloInteger contiene:
1 2 3 4 5 6

El arreglo arregloDouble contiene:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

El arreglo arregloCharacter contiene:
H O L A

Process finished with exit code 0
```

Además de este resultado obtenemos la misma función, sobrecargada en 3 diferentes casos, con el fin de adecuarse a lo que el usuario está mandando.

Cuando el compilador de Java encuentra una llamada a un método, busca una declaración que coincida con el nombre y los tipos de los argumentos.

Si sustituimos estos tipos por un parámetro genérico, comúnmente representado como E, todos los métodos se pueden unificar en uno solo. Este método genérico **imprimirArreglo** puede aceptar cualquier arreglo de objetos, utilizando el especificador de formato %s para imprimir su representación de cadena, lo que invoca automáticamente el método toString() de cada objeto. De esta manera, el uso de métodos genéricos no solo simplifica el código, sino que mejora su flexibilidad y reutilización.

3. MARCO TEÓRICO

3.1 MÉTODOS GENÉRICOS

Si los métodos sobrecargados realizan las mismas operaciones para diferentes tipos de argumentos, se puede simplificar todo a un método genérico. Esto permite que el método sea invocado con difertens tipos de argumentos, el compilador será el que se encargue de manejar cada tipo de dato adecuadamente. En un método genéricos los datos deben ser de tipo referencias, no primitivos (Int, Double, Char).

El parámetro en los métodos genéricos puede decararse una sola vez en la sección de parámetros de tipo, pero puede aparecer varias veces en la lista de parámetros del método. Esto es útil, por ejemplo, al declarar un método que acepte dos arreglos del mismo tipo:

```
Public static <E> void imprimirArreglos(E[] arreglo1, E[] arreglo2)
```

Es importante destacar que los nombres de los parámetros de tipo no necesitan ser únicos entre distintos métodos genéricos. Ahora vamos a re-implementar la aplicación original utilizando un método genérico imprimirArreglo, lo que permite llamadas idénticas a las de la implementación anterior.

La declaración del método genérico comienza con una sección de parámetros de tipo, delimitada por “<” y “>”, que precede al tipo de retorno del método. Cada parámetro de tipo actúa como un identificar que representa un tipo genérico, permitiendo su uso en la declaración del tipo de retorno, los parámetros y las variables locales dentro del método.

```
public static < E > void imprimirArreglo( E[] arregloEntrada ) {
    for ( E elemento : arregloEntrada )
        System.out.printf( "%s ", elemento );
    System.out.println();
}
```

En este ejercicio, estamos definiendo nuestro método usando generics, esto con el fin de darnos mayor flexibilidad de uso. Lo que realiza esto es que nos ahorra 3 códigos (los métodos anteriores), resumiendolo a un solo método. La utilidad de este método es imprimir cualquier tipo de arreglo.

```
public static void main( String args[] ) {
    Integer[] arregloInteger = { 1, 2, 3, 4, 5, 6 };
    Double[] arregloDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
    Character[] arregloCharacter = { 'H', 'O', 'L', 'A' };

    System.out.println( "El arreglo arregloInteger contiene:" );
    imprimirArreglo( arregloInteger );

    System.out.println( "\nEl arreglo arregloDouble contiene:" );
    imprimirArreglo( arregloDouble );

    System.out.println( "\nEl arreglo arregloCharacter contiene:" );
    imprimirArreglo( arregloCharacter );
}
```

Finalmente contamos con nuestro main donde mandamos los diferentes arreglos, para posteriormente imprimirlo.

Este proceso se repite para todas las llamadas a imprimirArreglo, donde el compilador identifica arregloDouble como Double[] y arregloCharacter como Character[], y en cada caso, encuentra que el método genérico es la opción correcta. Esto demuestra la flexibilidad de los métodos genéricos, que permiten manejar diferentes tipos de arreglos sin necesidad de múltiples implementaciones.

3.2 TIPO DE VALOR DE RETORNO

Aqui realizamos un metodo general, estamos uando la interfaz **Comparable<T>**. Aunque no se pueden usar operadores relacionales como el < o >, el método utiliza compareTo para comparar los objetos de la misma clase, asegurando que los argumentos sean objetos que implementen Comparable.

```
public static < T extends Comparable< T > > T maximo( T x, T y, T z ) {
        T max = x;
        if ( y.compareTo( max ) > 0 )
            max = y;
        if ( z.compareTo( max ) > 0 )
            max = z;
        return max;
    }
```

Borrado de Tipos.

Al compilar, el compilador realiza un proceso conocido como borrado, donde sustituye los parámetros de tipo con su límite superior, que en este caso es Comparable. Esto significa que, aunque el código fuente utiliza tipos genéricos, el bytecode solo tiene una referencia a Comparable, lo que simplifica el código y lo hace más eficiente.

Ventajas de los Genéricos.

Los genéricos no solo reducen la necesidad de múltiples sobrecargas de métodos, sino que también previenen errores de tipo en tiempo de compilación, asegurando que las conversiones sean seguras y evitando excepciones como ClassCastException. Este enfoque también garantiza que el método maximo funcione correctamente sin importar la posición de los argumentos.

Sobre carga de métodos genéricos.

Un método genérico puede sobrecargarse, lo que permite a una clase tener múltiples métodos con el mismo nombre, pero diferentes parámetros. Por ejemplo, el método genérico usado imprimirArreglo tiene una versión que incluya parámetros adicionales para especificar qué parte del arreglo imprimir. Los **métodos genéricos** también pueden sobrecargarse con métodos no genéricos que tengan el mismo nombre y número de parámetros.

3.3 CLASES GENERICAS

El concepto de una estructura de datos, como una pila, puede comprenderse en forma independiente del tipo de elementos que manipula. Las clases genéricas proporcionan los medios para describir el concepto de una pila (o cualquier otra clase) en forma independiente de su tipo. Así, podemos crear instancias de objetos con tipos específicos de la clase genérica.

Estas clases se conocen como clases o tipos parametrizados, ya que aceptan uno o más parámetros. Recuerdar que los parámetros de tipo solo representan a los tipos de referencia, lo cual significa que la clase genérica **PILA** no puede instanciarse con tipos primitivos (int, double, float, char, bool). Sin embargo, podemos instancias una pila que almacene objetivos de las clases de envoltura de tipos de JAVA (Integer, Character, etc) y permitir que JAVA utilice la conversión autoboxing para convertir los valores primitivos en objetos. La conservación autoboxing ocurre cuando un valor de un tipo primitivo se mete en una Pila que contiene objetis de clase de envoltura como Integer. La conversión autounboxing ocurre cuando un objeto de la clase de envoltura se saca de la Pila y se asigna a una variable de tipo primitivo.

Para una mejor explicación podemos decir que la Pila es una estructura de datos que puede almacenar elementos de cualquier tipo, pero es necesario que sepamos de antemano que tipo de datos almacenara. Gracias a las clases genéricas, podemos diseñar una pila que pueda manejar cualquier tipo de datos y eso le da flexibilidad.

Este enfoque genérico permite la **reutilización del código**. En lugar de tener que reescribir la clase de pila para cada tipo de dato (Pila de enteros, Pila de caracteres, etc.), se escribe una sola vez y puede reutilizarse con diferentes tipos de datos.

En Java, los tipos primitivos como int, double, float, etc., no pueden usarse directamente como parámetros de tipo en una clase genérica. Esto se debe a que las clases genéricas solo pueden manejar tipos de referencia (es decir, objetos). Sin embargo, Java ofrece una solución con las clases envoltorio (Integer, Double, Character, etc.), que permiten tratar estos tipos primitivos como objetos.

Autoboxing: Es el proceso automático mediante el cual Java convierte un tipo primitivo en su correspondiente clase envoltorio. Por ejemplo, cuando se almacena un int en una Pila<Integer>, Java convierte automáticamente el int en un Integer (la clase envoltorio).

```
Pila<Integer> pila = new Pila<>();
pila.push(5);
```

Autounboxing: Ocurre cuando Java convierte automáticamente un objeto envoltorio en su tipo primitivo correspondiente al extraerlo de la pila. Por ejemplo, cuando sacamos un Integer de una pila y lo asignamos a un int.

```
int valor = pila.pop();
```

A continuación se presenta una declaración de la clase genérica Pila. La declaración de una clase genérica es similar a la de una clase no genérica, excepto que el nombre de la clase va seguido de una sección de parámetros de tipo. En este caso, el parámetro de tipo E el tipo del elemento que manipulará la Pila. Al igual que con los métodos genéricos, la sección de parámetros de tipo de una clase genérica puede tener uno o más parámetros separados por comas. El parámetro de tipo E se utiliza en la declaración de la clase Pila para representar el tipo del element

```
package MarcoTeorico;
public class Pila< E > {
    private final int tamanio;
    private int superior;
    private E[] elementos;
```

```
    public Pila(){
        this( 10 );
    }

    public Pila( int s ) {
        tamanio = s > 0 ? s : 10;
        superior = -1;
        elementos = ( E[] ) new Object[ tamanio ];
    }

    public void push( E valorAMeter ) {
        if ( superior == tamanio - 1 )
            throw new ExcepcionPilaLlena( String.format("La Pila esta llena, no se puede meter %s",
valorAMeter ) );
        elementos[ ++superior ] = valorAMeter;
    }

    public E pop(){
        if ( superior == -1 )
            throw new ExcepcionPilaVacia( "Pila vacia, no se puede sacar" );
        return elementos[ superior-- ];
    }
}

class ExcepcionPilaLlena extends RuntimeException {
    public ExcepcionPilaLlena(String mensaje) {
        super(mensaje);
    }
}

class ExcepcionPilaVacia extends RuntimeException {
    public ExcepcionPilaVacia(String mensaje) {
        super(mensaje);
    }
}
```

Ahora probamos el código:

```
public class PruebaPila {
    private double[] elementosDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
    private int[] elementosInteger = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
    private Pila<Double> pilaDouble;
    private Pila<Integer> pilaInteger;

    public void pruebaPilas() {
        pilaDouble = new Pila<Double>(5);
        pilaInteger = new Pila<Integer>(10);
        pruebaPushDouble();
        pruebaPopDouble();
        pruebaPushInteger();
        pruebaPopInteger();
    }

    public void pruebaPushDouble() {
        try {
            System.out.println("\nMetiendo elementos en pilaDouble");
            for (double elemento : elementosDouble) {
                System.out.printf(" %.1f ", elemento);
                pilaDouble.push(elemento);
            }
        } catch (ExcepcionPilaLlena excepcionPilaLlena) {
            System.err.println();
            excepcionPilaLlena.printStackTrace();
        }
```

```
    }

    public void pruebaPopDouble() {
        try {
            System.out.println("\nSacando elementos de pilaDouble");
            double valorASacar;
            while (true) {
                valorASacar = pilaDouble.pop();
                System.out.printf("%.1f ", valorASacar);
            }
        } catch (ExcepcionPilaVacia excepcionPilaVacia) {
            System.err.println();
            excepcionPilaVacia.printStackTrace();
        }
    }

    public void pruebaPushInteger() {
        try {
            System.out.println("\nMetiendo elementos a pilaInteger");
            for (int elemento : elementosInteger) {
                System.out.printf(" %d ", elemento);
                pilaInteger.push(elemento);
            }
        } catch (ExcepcionPilaLlena excepcionPilaLlena) {
            System.err.println();
            excepcionPilaLlena.printStackTrace();
        }
    }

    public void pruebaPopInteger() {
        try {
            System.out.println("\nSacando elementos de pilaInteger");
            int valorASacar;
            while (true) {
                valorASacar = pilaInteger.pop();
                System.out.printf("%d ", valorASacar);
            }
        } catch (ExcepcionPilaVacia excepcionPilaVacia) {
            System.err.println();
            excepcionPilaVacia.printStackTrace();
        }
    }

    public static void main(String args[]) {
        PruebaPila aplicacion = new PruebaPila();
        aplicacion.pruebaPilas();
    }
}
```

En esta aplicación de prueba que utiliza la clase genérica Pila con los tipos Double e Integer. El compilador utiliza estos tipos como argumentos de tipo para hacer la comprobación y conversión de tipos. La aplicación crea dos pilas: una para Double y otra para Integer. Luego, se ejecutan varios métodos para demostrar cómo funcionan estas pilas, incluyendo push y pop.

El método pruebaPushDouble intenta agregar varios valores double a la pila, y cuando se intenta agregar un sexto valor, que excede la capacidad, se lanza una excepción llamada ExcepcionPilaLlena. Esta excepción es atrapada y se imprime un rastreo de la pila para diagnosticar dónde ocurrió el error.

3.4 GENERICOS y HERENCIA

Los genéricos pueden utilizarse con la herencia de varias formas:

- Una clase genérica puede derivarse a una clase no genérica.
- Una clase genérica puede derivarse de otra clase genérica
- Una clase no genérica puede derivarse de una clase genérica
- Un método genérico en una subclase puede sobrescribir a un método genérico en una superclase, si ambos métodos tienen las mismas firmas.

3.5 NOTAS SOBRE EL METODO EQUALS

El método equals en Java es un método heredado de la clase base Object y su principal función es comparar dos objetos para verificar si son "iguales". La implementación por defecto de equals en la clase Object compara las referencias de los objetos, es decir, devuelve true si ambos objetos son la misma instancia en memoria. Sin embargo, muchas clases sobrescriben este método para comparar los contenidos o valores de los objetos, en lugar de las referencias.

```
Public boolean equeals(objet obj)
```

Este metodo toma un objeto como parámetro y devuelve un valor booleano:

- True: si los objetos son considerados “iguales”
- False: si son diferentes

```
package MarcoTeorico;

public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
         @Override     public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Persona persona = (Persona) obj;
                 return edad == persona.edad && nombre.equals(persona.nombre);
         }
}
```

3.6 NOTAS SOBRE ARRAYLIST

ArrayList es una colección en Java que permite almacenar elementos dinámicamente, estos podemos verlos como un arreglo que cambia.

```java
import java.util.ArrayList;

public class EjemploArray {
    public static void main(String[] args) {

        ArrayList<String> frutas = new ArrayList<>();

        frutas.add("Manzana");
        frutas.add("Banana");
        frutas.add("Naranja");

        System.out.println(frutas.get(1)); // Imprime "Banana"

        for (String fruta : frutas) {
            System.out.println(fruta);
        }
    }
}
```

4. ACTIVIDADES

4.1 Actividad 1

Programe el método genérico para imprimirArreglo visto en el marco téorico y sobrecárguelo para que reciba dos argumentos adicionales: subindiceInferior y subindiceSuperior. Este método debe imprimir solo la parte del arreglo que esté dentro de los índices indicados. Valide que ambos índices estén dentro del rango permitido. Si alguno de los dos está fuera de rango, o si subindiceSuperior es menor o igual a subindiceInferior, el método sobrecargado debe lanzar una excepción InvalidSubscriptException. De lo contrario, el método debe devolver la cantidad de elementos impresos. Modifique el método main para ejecutar ambas versiones de imprimirArreglo con los arreglos

Este es nuestro método inicial, el cual es capaz de imprimir todos los elementos de nuestro arreglo, sin importar el tipo de datos que sea ya que es un genero y nos da esa flexibilidad.

```java
public static < E > void imprimirArreglo( E[] arregloEntrada ) {
    for ( E elemento : arregloEntrada )
        System.out.printf( "%s ", elemento );
    System.out.println();
}
```

Esta otra funcion por otro lado, realiza una funcion similar al anterior método, la diferencia radica en que a este se le solicita un index inferior y superior, para poder imprimir en ese margen los elementos solicitiados. Aquí adicionalmente se hace uso de una excepción para verificar si los valores del index son validos.

```
public static <E> int imprimirArreglo(E[] arregloEntrada, int indexLower, int indexUpper) throws
InvalidSubscriptException {
    if (indexLower < 0 || indexUpper >= arregloEntrada.length || indexUpper < indexLower) {
        throw new InvalidSubscriptException("Índices fuera de rango y/o inválidos.");
    }

    int totalElements = 0;
    for (int i = indexLower; i <= indexUpper; i++) {
        System.out.printf("%s ", arregloEntrada[i]+ "\n");
        totalElements++;
    }

    return totalElements;
}
```

En este caso, realizamos la creación de nuestra excepción, utilizando el método de la clase para permitirnos mayor control, en este mismo se puede incluir la excepción para posteriormente solo llamarla.

```
class InvalidSubscriptException extends Exception {
    public InvalidSubscriptException(String message) {
        super(message);
    }
}
```

Nuestro Main, el cual es similar al que se nos entrego en la practica, la diferencia esta en el proceso para resaltar los diferentes llamados (de ambos métodos sobrecargados).

```
public static void main( String args[] ) throws InvalidSubscriptException {
        Integer[] arregloInteger = { 1, 2, 3, 4, 5, 6 };
        Double[] arregloDouble = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        Character[] arregloCharacter = { 'H', 'O', 'L', 'A' };

        System.out.println( "El arreglo arregloInteger contiene:" );
        int amount = imprimirArreglo( arregloInteger, 0 , 3 );
        System.out.println( "El arreglo arregloDouble contiene:" + amount);

        System.out.println( "\nEl arreglo arregloDouble contiene:" );
        imprimirArreglo( arregloDouble );

        System.out.println( "\nEl arreglo arregloCharacter contiene:" );
        imprimirArreglo( arregloCharacter );
    }
}
```

4.2 Actividad 2

Programe la Pila genérica vista en el marco teórico.

```
public class Pila<E> {
    private final int tamanio;
    private int superior;
    private E[] elementos;

    public Pila() {
        this(10);
    }

    public Pila(int s) {
        tamanio = s > 0 ? s : 10;
        superior = -1;
        elementos = (E[]) new Object[tamanio];
```

```
    }

    public void push(E valorAMeter) {
        if (superior == tamanio - 1) {
            throw new ExcepcionPilaLlena(String.format("La Pila esta llena, no se puede meter %s",
valorAMeter));
        }
        elementos[++superior] = valorAMeter;
    }

    public E pop() {
        if (superior == -1) {
            throw new ExcepcionPilaVacia("Pila vacia, no se puede sacar");
        }
        return elementos[superior--];
    }
}
```

Modifica la clase Pila para implementar el método contains(E elemento): Este método debe recibir un elemento y devolver true si el elemento está en la pila, o false si no lo está. La búsqueda debe realizarse desde el tope de la pila hacia el fondo, sin modificar el estado actual de la pila.

Para este caso, para poder implementar contains, en un principio pensé en utilizar literalmente el método **contains** pero este no se puede usar directamente, esto ya que en un arreglo no se contiene dicho método, así que en lugar de usar eso, preferí por utilizar un bucle for, donde el 1er elemento que sea igual al solicita, automáticamente nos dará el true o la confirmación de que existe, en caso no sea así, nos dará un false.

```
public boolean contains(E element) {
    for (int i = 0; i <= superior; i++) {
        if (elementos[i].equals(element)) {
            return true;
        }
    }
    return false;
}
```

Aquí nuestro main el cual ayudo con las pruebas.

```
public class Main {
    public static void main(String[] args) {
        Pila<Integer> pila = new Pila<>(5);

        pila.push(10);
        pila.push(20);
        pila.push(30);

        System.out.println("¿La pila contiene el número 20? " + pila.contains(20));
        System.out.println("¿La pila contiene el número 40? " + pila.contains(40));

        System.out.println("Elemento sacado: " + pila.pop());
        System.out.println("¿La pila contiene el número 30? " + pila.contains(30));

    }
}
```

4.3 Actividad 3

Crea una clase IgualGenerico donde escriba una versión genérica simple del método esIgualA que compare sus dos argumentos con el método equals y devuelva true si son iguales, y false en caso contrario. Use este método genérico en un programa que llame a esIgualA con los tipos integrados, null, Object, Integer y String. ¿Qué resultado obtiene al tratar de ejecutar este programa?

Aqui realizamos el programa, lo ponemos en el caso en el que ambos valores puedan ser nulos de entrada o que sean diferentes (por ello los dos primeros IFs), este método lo terminamos con el uso de equals, este se encarga de comparar si los objetos o valores son iguales o diferentes. Para este ejercicio lo probamos con diferentes valores.

```
public static <T> boolean esIgualA(T a, T b) {
    if (a == null && b == null) {
        return true;
    }

    if (a == null || b == null) {
        return false;
    }

    return a.equals(b);
}
```

Aquí el Main, créditos a chat GPT por generar los casos de uso.

```
public static void main(String[] args) {
        // Comparación de tipos integrados
        System.out.println("Comparación de enteros: " + IgualGenerics.esIgualA(10, 10)); // true
        System.out.println("Comparación de enteros distintos: " + IgualGenerics.esIgualA(10, 20)); //
false

        // Comparación con null
        System.out.println("Comparación de null con null: " + IgualGenerics.esIgualA(null, null)); //
true
        System.out.println("Comparación de null con un entero: " + IgualGenerics.esIgualA(null, 10));
// false
        System.out.println("Comparación de un entero con null: " + IgualGenerics.esIgualA(10, null));
// false

        // Comparación de Object
        Object obj1 = new Object();
        Object obj2 = new Object();
        System.out.println("Comparación de objetos distintos: " + IgualGenerics.esIgualA(obj1,
obj2)); // false
        System.out.println("Comparación de un objeto consigo mismo: " + IgualGenerics.esIgualA(obj1,
obj1)); // true

        // Comparación de Integer
        Integer num1 = 100;
        Integer num2 = 100;
        Integer num3 = 200;
        System.out.println("Comparación de Integer iguales: " + IgualGenerics.esIgualA(num1, num2));
// true
        System.out.println("Comparación de Integer distintos: " + IgualGenerics.esIgualA(num1,
num3)); // false

        // Comparación de String
        String str1 = "Hola";
        String str2 = "Hola";
```

```
        String str3 = "Bye";
        System.out.println("Comparación de String iguales: " + IgualGenerics.esIgualA(str1, str2));
// true
        System.out.println("Comparación de String distintos: " + IgualGenerics.esIgualA(str1, str3));
// false
    }
```

```
 Actividad.IgualGenerics
Comparación de enteros: true
Comparación de enteros distintos: false
Comparación de null con null: true
Comparación de null con un entero: false
Comparación de un entero con null: false
Comparación de objetos distintos: false
Comparación de un objeto consigo mismo: true
Comparación de Integer iguales: true
Comparación de Integer distintos: false
Comparación de String iguales: true
Comparación de String distintos: false

Process finished with exit code 0
```

4.4 Actividad 4

Implementa el método genérico esIgual(Pila otraPila) en la clase Pila: Este método debe comparar dos pilas para verificar si contienen los mismos elementos en el mismo orden. Si ambas pilas son iguales en tamaño y contenido, debe devolver true, de lo contrario, false. El método no debe modificar el estado.

Para esta actividad lo que podemos realizar es comparar las pilas, en caso sean diferentes pues reotrnara el falso, en caso pase ese filtro. Continuaremos con un for el cual verificara si los elementos son iguales y si están en el mismo orden, en caso algo sea diferente pues retornara el falso.

Una vez pasado ambos casos para verificar si son iguales o diferentes, finalmente podremos retornar true, afirmando que las PILAS no son diferentes y cuentan con el mismo orden de elementos.

```
public boolean esIgual(Pila<E> otraPila) {
    if (this.superior != otraPila.superior) {
        return false;
    }
         for (int i = 0; i <= this.superior; i++) {
        if (!this.elementos[i].equals(otraPila.elementos[i])) {
            return false;
        }
    }

    return true;  }
```

Contamos con nuestro main que nos ayuda a validar los diferentes casos.

```
public static void main(String[] args) {
    Pila<Integer> pila1 = new Pila<>(5);
    pila1.push(10);
    pila1.push(20);
    pila1.push(30);

    Pila<Integer> pila2 = new Pila<>(5);
    pila2.push(10);
    pila2.push(20);
    pila2.push(30);

    Pila<Integer> pila3 = new Pila<>(3);
    pila3.push(20);
    pila3.push(10);
    pila3.push(30);

    System.out.println("¿Las pilas son iguales? " + pila1.esIgual(pila2));
    System.out.println("¿Las pilas son iguales? " + pila1.esIgual(pila3));
    System.out.println("¿Las pilas son iguales? " + pila2.esIgual(pila3));
}
```

5. EJERCICIOS

5.1 Ejercicio

Escriba una clase genérica llamada Par, que tenga dos parámetros de tipo: F y S, cada uno de los cuales representa el tipo del primer y segundo elementos del par, respectivamente. Agregue métodos getPrimero, getSegundo, setPrimero y setSegundo para los elementos primero y segundo del par. [Sugerencia: el encabezado de la clase debe ser public class Par< F, S >]. Agregue el método toString que devuelva la representación del par de la forma “(Primero: x, Segundo: y)”.

Realzamos nuestra clase, tomamos en consideración la recomendación del encabezado para tener mayor flexibilidad. Cuando queramos imprimir a nuestro objeto, se llamara automáticamente a nuestro toString, permitiendo una impresión de nuestros números o valores ingresados.

```
class Par<F,S> {
    private F firstN;
    private S secondN;

    public Par(F firstN, S secondN) {
        this.firstN = firstN;
        this.secondN = secondN;
    }

    public F getFirstN() {
        return firstN;
    }

    public S getSecondN() {
        return secondN;
    }

    public void setFirstN(F firstN) {
        this.firstN = firstN;
    }

    public void setSecondN(S secondN) {
        this.secondN = secondN;
```

```
    }

    @Override     public String toString() {
        return String.format("(Primero: %s, Segundo: %s)", firstN, secondN);
    }
```

Posteriormente usamos nuestro MAIN para poder observar casos de uso practico.

```
public static void main(String[] args) {
    Par<Integer, String> par1 = new Par<>(1, "Uno");
    System.out.println(par1);

    Par<Double, String> par2 = new Par<>(3.5, "decimal");
    System.out.println(par2);

    Par<String, Double> par3 = new Par<>("Pi", 3.14159);
    System.out.println(par3);

}
```

```
 Ejercicio.Par
(Primero: 1, Segundo: Uno)
(Primero: 3.5, Segundo: decimal)
(Primero: Pi, Segundo: 3.14159)
```

5.2 Ejercicio

Agrega un método esIgual en la clase Par que compare dos pares. Este método debe devolver true si ambos pares son iguales (mismos valores en el mismo orden) y false en caso contrario. Prueba este método en la clase PruebaPar.

Para este ejercicio, consideramos 2 casos, el 1ro donde el otro objeto que íbamos a ingresar es nulo, ósea que al ser vacio no debería poder ser semejante a un objeto el cual invoca el método en 1ra instancia. Posteriormente para identificar si el orden de los valores es correcto usamos una comparación simple, donde si todo es correcto retornara un true, caso que el orden sea diferente,, nos dará un false.

```
public boolean esIgual(Par<F, S> otroPar) {
    if (otroPar == null) {
        return false;
    }

    return this.firstN.equals(otroPar.getFirstN()) && this.secondN.equals(otroPar.getSecondN());
}
```

El main para observar los casos de uso.

```
    public static void main(String[] args) {
        Par<Double, String> par1 = new Par<>(1.0, "Uno");
        System.out.println(par1);
```

```
        Par<Double, String> par2 = new Par<>(1.0, "Uno");
        System.out.println(par2);

        Par<Double, String> par3 = new Par<>(3.14159, "Pi");
        System.out.println(par3);
                System.out.println("¿par1 es igual a par2? " + par1.esIgual(par2));
        System.out.println("¿par1 es igual a par3? " + par1.esIgual(par3));
        System.out.println("¿par2 es igual a par3? " + par2.esIgual(par3));
```

```
(Primero: 3.14159, Segundo: Pi)
¿par1 es igual a par2? true
¿par1 es igual a par3? false
¿par2 es igual a par3? false

Process finished with exit code 0
```

5.3 Ejercicio

Crea un método genérico estático imprimirPar en la clase Main que acepte un Par como argumento y lo imprima. Llama a este método con pares de String,Integer; Double, Boolean; PERSONA, Integer.

Disculpe Ing si me estoy equivocando, pero por lo que entiendo necesito generar una funcion que me permita imprimir los objetos, sin embargo hay un tipo de variable que no concuerda el cual es PERSONA, no se si yo este bien o mal pero optare cambiarlo por un Double.

Lo que realizamos en esta funcion es solicitar al usuario que ingrese un objeto X, esto con el fin de que podamos imprimir ese objeto, ahora este método llamara al toString para facilitar la impresión del objeto.

```
public static <F, S> void imprimirPar(Par<F, S> par) {
    System.out.println(par.toString());
}
```

Para el main lo que se ha realizado son los diferentes casos de uso

```
public static void main(String[] args) {
    Par<String, Integer> par1 = new Par<>("Edad", 30);
    Par<Double, Boolean> par2 = new Par<>(3.14, true);
    Par<Double, Integer> par3 = new Par<>(2.718, 42);

    imprimirPar(par1);
    imprimirPar(par2);
    imprimirPar(par3);
```

```
(Primero: Edad, Segundo: 30)
(Primero: 3.14, Segundo: true)
(Primero: 2.718, Segundo: 42)
```

5.4 Ejercicio

Crear una clase genérica Contenedor que almacene múltiples pares de elementos. Implemente una clase genérica llamada Contenedor. Esta clase debe contener un ArrayList para almacenar múltiples objetos de tipo Par. Debe incluir los siguientes métodos

λ agregarPar(F primero, S segundo): Permite añadir un nuevo par al contenedor.

```
//Agregar por atributos
    public void agregarPar(F primero, S segundo) {
        Par<F, S> nuevoPar = new Par<>(primero, segundo);
        contenedorPares.add(nuevoPar);
    }

    //Agregar por objetos directos
    public void agregarPar(Par<F,S> par) {
        contenedorPares.add(par);
    }
```

Estos metodos agregan objetos a mi contenedorPares, pero cada manera en que los agregan son idferentes, uno es un método con el objeto directo y el otro con los atributos de las clase.

λ obtenerPar(int indice): Devuelve el par en la posición especificada.

```
// obtener pares por indice
    public Par<F, S> obtenerPar(int indice) {
        if (indice >= 0 && indice < contenedorPares.size()) {
            return contenedorPares.get(indice);
        }
        throw new IndexOutOfBoundsException("Índice fuera de rango");
    }
```

Este método por otro lado, lo que realiza es la obtención de los objetos por medio del índice, retornando únicamente el valor del índice.

λ obtenerTodosLosPares(): Devuelve la lista completa de pares.

```
//obtener todos los pares del arreglo
    public List<Par<F, S>> obtenerTodosLosPares() {
        return contenedorPares;
    }
```

Retorna todos los valores

λ mostrarPares(): Imprime todos los pares almacenados en el contenedor

```
// mostrar todos los pares
    public void mostrarContenedor() {
        for (Par<F,S> par : contenedorPares ) {
            System.out.printf("%s " + par +"\n");
        }
    }
```

Imprime todos los valores

6. CONCLUSIONES DE LA PRÁCTICA:

- En lugar de tener múltiples métodos que realizan la misma tarea pero con diferentes tipos de datos, podemos usar un método genérico que permite manejar distintos argumentos de manera más eficiente y con menos esfuerzo. Esto simplifica el código y lo hace más fácil de mantener.
- Los métodos genéricos nos ahorran la necesidad de escribir varias versiones del mismo método para diferentes tipos de datos, haciéndolos mucho más flexibles y reutilizables. Así, evitamos tener que duplicar esfuerzos cuando se necesita realizar tareas similares con distintos tipos de información.
- Los métodos genéricos también pueden comparar distintos tipos de objetos siempre que implementen la interfaz `Comparable`. Esto hace que el código sea más consistente y versátil, ya que nos asegura que podemos comparar elementos de manera efectiva sin importar su tipo específico.
- Cuando el compilador encuentra un tipo genérico, lo reemplaza con su límite superior, como `Comparable`, lo que hace que el código generado sea más eficiente y directo. A nivel técnico, este proceso simplifica las cosas detrás de escena sin que tengamos que preocuparnos demasiado.

7. CUESTIONARIO

7.1 Explique qué significa restringir un tipo genérico con extends en la definición de una clase o método genérico. Proporcione un ejemplo de cómo se puede utilizar esta característica.

Cuando utilizamos extends en la definición de un genérico, lo que hacemos es limitar los tipos que se pueden usar con ese parámetro genérico a aquellos que heredan de una clase especifica. Esto lo realizamos con el afán de poder tener mayor control sobre lo que queremos realizar, por ejemplo, necesitamos un método genérico que solo trabaje con números, si no restringimos el tipo del objeto, podríamos terminar permitiendo cualquier acceso a cualquiero tipo de datp, lo que podría generar errores al momento de ejecutar el programa.

El tipo genérico debe ser un subtipo de la clase o interfaz que se especifica con extends. (Ricardo & Ricardo, 2024)

```
public class NumberPrint {
    public <T extends Number> void printNumber(T number) {
        System.out.println("The number is: " + number);
    }
}
```

Aquí restringimos el uso a solo valores números, denegan el acceso a otros tipos de datos como String o Boolean, utilizamos el extends para asegurar que este método solo reciba subtipos números “Number”

7.2 Discuta algunas limitaciones de los generics en Java. Por ejemplo, ¿se pueden usar tipos primitivos como parámetros de tipo? Justifique su respuesta.

Algo a tener muy en cuenta es que los genéricos no pueden usar datos primitivos como parámetros como int, char, etc. Esto ya que los genéricos solo funcionan con tipos de referencias u objetos (como yo lo entiendo). Esto nos dice que debemos de usar tipos de datos mas envolvente o con características de instancias a una clase, entre estos tenemos a los Integer, String, Double, etc.

Los genéricos en Java están basados en la conversión (type erasure), y los tipos primitivos no pueden ser representados como Object, que es la base de los genéricos.

7.3 Describa un caso de uso real donde el uso de generics podría ser beneficioso en una aplicación. Explique cómo los generics mejorarían la claridad y la seguridad del código.

Usamos genéricos en Java para asegurar el tipo de datos en tiempo de compilación, reducir errores y aumentar la legibilidad y reutilización del código. En los casos de vida real, el obtener datos generales como en un BD o en algún sistema de estructura de datos nos daría la facilidad de manejar dichos datos, convirtiéndolos en aquello que necesito, ya que al ser genéricos podríamos manejarlo como más nos convenga.

Por ejemplo, una clase genérica List<T> puede almacenar cualquier tipo de objeto.

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
```

Pero esto no solo se quedaría allí, también podríamos realizar registros, informes con esa información.

7.4 ¿Qué problemas pueden surgir al utilizar generics con una jerarquía de clases en Java? Discuta cómo la herencia afecta a los generics.

Uno de los problemas que pude identificar jugando con los generics es que con una jerarquía de clases no se pueden referenciar a otras clases heredadas. Según StarkOverFlow esto se le conoce como **covarianza**.

Los que entendí fue que los genéricos no soportan la covarianza de forma implícita y que al momento de manejar este tipo de cosas, es mejor usar los wildcards o métodos sobrecargados para manejar la jerarquía de clases. En C++ habia presentado un problema similar, pero que mal no recuerde si me dejaba acceder con mayor facilidad a la jararquia de clases (no estoy seguro del todo).

7.5 ¿Qué es un wildcard?

El caracter Wildcard(?) aporta una solución y nos vale para decirle al lenguaje Java que cuando usemos un tipo genérico se puede aplicar cualquier tipo al parámetro lista. (Caules, 2021)

Así se le agrega flexibilidad ya que usamos cualquier tipo de dato como parámetro genérico, podemos decir que es un tipo genérico que representa cualquier tipo desconocido de dato, el cual se usa principalmente en situaciones en las que no podemos manejar o desconocemos el tipo de dato.

7.6 ¿Cuál es la diferencia entre usar un tipo de parámetro genérico (por ejemplo, T) y un wildcard (?)? Proporcione ejemplos de cuándo sería adecuado usar cada uno.

Un tipo de parámetro genérico como T define un tipo específico para la instancia del objeto, mientras que un wildcard (?) permite una mayor flexibilidad al trabajar con diferentes tipos.

Para el genérico, si sabemos con que tipo de datos se va a trabajar es buena idea manejarlo a corde lo que necesitamos o a corde a los valores que tenemos, como lo vimos en las actividades y/o ejercicios, se le puede dar un caso de uso dependiendo de lo que se solicite.

```
public <T> void Metodo(T item) {
    // Código
}
```

Pero por otra parte contamos con el wildcard (?), el cual nos da una mayor flexibilidad al momento de utilizar los generics.

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

En este código observamos que desconocemos el tipo de dato, así que al usar el wildcard nos da una flexibilidad para poder manejar todo lo correspondiente a ello.

7.7 ¿Qué significa > y cuándo se usa?

El símbolo > se usa en la definición de genéricos en Java para indicar que un tipo es "subclase de" o "implementa una interfaz". Se utiliza para restringir el tipo genérico y proporcionar un límite superior.

```
public <T extends Comparable<T>> T getMax(T a, T b) {
    return a.compareTo(b) > 0 ? a : b;
}
```

En este ejemplo, T extends Comparable<T> significa que el tipo T debe implementar la interfaz Comparable, lo que permite usar el método compareTo() en los objetos de tipo T.

8. BIBLIOGRAFÍA

D. F. Silva, R. M. (2021). "A Study on the Performance of Java Virtual Machine Garbage Collectors. *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 1*(35), 23-31.

Hortsmann, C. (2009). *Big Java.* San José: Pearson.

Hortsmann, C. S. (2019). *Big Java: Early Objects.* Hoboken: John Wiley & Sons.

Oracle. (1994). *The Java™ Tutorials*. (Oracle) Recuperado el 28 de 07 de 2024, de https://docs.oracle.com/javase/tutorial/index.html

Ricardo, & Ricardo. (2024, March 29). *Genéricos en JAVA*. Programando En Java. https://programandoenjava.com/genericos-en-java/

Caules, C. Á. (2021, August 2). *Java Generics (II) uso de WildCard*. Arquitectura Java. https://www.arquitecturajava.com/java-generics-uso-de-wildcard/

Picodotdev. (2019, April 5). *El concepto de wildcard capture en Java*. Blog Bitix. https://picodotdev.github.io/blog-bitix/2019/04/el-concepto-de-wildcard-capture-en-java/

9. Anexo: Adjuntar los PROGRAMAS

En Hiperviculo → Guia5.zip