Interfaces

Interfaces parte 2

Contenido

Interfaces - parte 1

- ¿Qué es una interface?
- Usos y generalidades.
- Declaración.
- Implementación.
- Herencia de interfaces.
- Ventajas.

Interfaces - parte 2

- Interfaces funcionales.
- Interfaces funcionales predefinidas.
- Comparator<T> y Comparable<T>.
- Iterator<T> e Iterable<T>.

Interfaces funcionales

Una interfaz funcional es aquella que tiene *un solo método abstracto*.
Estas interfaces son clave para el uso de expresiones *lambda* y *referencias de método*.

```java
@FunctionalInterface
public interface Operacion {

    int ejecutar(int a, int b);
}
```

```java
Operacion suma = (a, b) -> a + b;

System.out.println(suma.ejecutar(5, 3));
```

Interfaces funcionales predefinidas

El ordenamiento de colecciones en Java se puede lograr fácilmente utilizando interfaces funcionales, como las que proporciona el framework **java.util**.

Dos interfaces clave para el ordenamiento son:

- **Comparator<T>**: Se utiliza para definir el orden de los elementos en una colección.
- **Comparable<T>**: Implementada por las clases cuyos objetos pueden ser ordenados de manera natural.

Interfaces funcionales predefinidas - Comparator<T>

El **Comparator<T>** posee el método abstracto ***compare(T obj1, T obj2)*** y se utiliza para definir un ***orden externo*** sobre una colección.

```
ArrayList<Integer> numeros = new ArrayList<>();
numeros.add(5);
numeros.add(88);
numeros.add(-15);
numeros.add(0);

numeros.sort((n1, n2) -> Integer.compare(n1, n2));
// [-15, 0, 5, 88]
```

```
List<Persona> personas = Arrays.asList(
    new Persona("Ana", 56),
    new Persona("Roberto", 32),
    new Persona("Zulema", 15)
    );

personas.sort((p1, p2) -> Integer.compare(p1.edad, p2.edad));

System.out.println(personas);
// [Nombre: Zulema - edad: 15, Nombre: Roberto - edad: 32, Nombre: Ana - edad: 56]
```

Interfaces funcionales predefinidas - Comparator<T>

El método **Comparator.comparing** puede usarse para crear comparadores a partir de funciones de tipo ***getter***, cuándo la lógica de comparación es simple.

```
personas.sort(Comparator.comparing(p -> p.getNombre()));
System.out.println(personas);
// [Nombre: Ana - edad: 56, Nombre: Roberto - edad: 32, Nombre: Zulema - edad: 15]

personas.sort(Comparator.comparing(p -> p.getEdad()));
System.out.println(personas);
//[Nombre: Zulema - edad: 15, Nombre: Roberto - edad: 32, Nombre: Ana - edad: 56]

personas.sort(Comparator.comparing((Persona p) -> p.getEdad()).reversed());
System.out.println(personas);
// [Nombre: Ana - edad: 56, Nombre: Roberto - edad: 32, Nombre: Zulema - edad: 15]

personas.sort(Comparator.comparing((Persona p) -> p.getNombre()).reversed());
System.out.println(personas);
// [Nombre: Zulema - edad: 15, Nombre: Roberto - edad: 32, Nombre: Ana - edad: 56]
```

Interfaces funcionales predefinidas - Comparator<T>

El método **Comparator.thenComparing()** puede usarse para ordenar por múltiples criterios.

```
personas.sort(Comparator.comparing((Persona p) -> p.getNombre())
                        .thenComparingInt((Persona p) -> p.getEdad()));

System.out.println(personas);
// [Nombre: Ana - edad: 30, Nombre: Ana - edad: 56, Nombre: Roberto - edad: 32]
```

Interfaces funcionales predefinidas - Comparable<T>

```java
public class Persona implements Comparable<Persona> {

    private String nombre;
    private int edad;

    public String getNombre() {
        return this.nombre;
    }

    public int getEdad() {
        return this.edad;
    }

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    @Override
    public String toString(){
        return "Nombre: " + this.nombre + " - edad: " + this.edad;
    }

    @Override
    public int compareTo(Persona p) {
        return Integer.compare(this.edad, p.edad);
    }
}
```

La interface **Comparable<T>** permite que una clase defina un **orden natural**.

Se debe sobrescribir el método ***compareTo(T obj)***.

```java
Collections.sort(personas);

System.out.println(personas);
// [Nombre: Zulema - edad: 15, Nombre: Roberto - edad: 32, Nombre: Ana - edad: 56]

Collections.reverse(personas);

System.out.println(personas);
// [Nombre: Ana - edad: 56, Nombre: Roberto - edad: 32, Nombre: Zulema - edad: 15]
```

Ejercitación

Interfaces funcionales predefinidas

Las interfaces relacionadas con la iteración de elementos en colecciones son **Iterator** e **Iterable**.

Cabe destacar algunas formas más modernas como las funciones de **streaming** y **forEach**.

Interfaces funcionales predefinidas - Iterator<T>

Las interface **Iterator<T>** permite recorrer elementos de una colección de a uno por vez. Es una interface fundamental en Java, especialmente útil cuando se necesita más control sobre el recorrido, como la posibilidad de eliminar elementos durante la iteración.

Métodos principales:

- hasNext(): retorna *true* si hay más elementos por iterar.
- next() : retorna el siguiente elemento en la iteración.
- remove(): elimina el último elemento devuelto por *next()*. Opcional.

Interfaces funcionales predefinidas - Iterator<T>


```java
List<String> nombres = new ArrayList<>(Arrays.asList("Ana", "Juan", "Pedro", "Lucía"));

Iterator<String> iterador = nombres.iterator();

while (iterador.hasNext()) {

    String nombre = iterador.next();

    if (nombre.equals("Juan")) {
        iterador.remove();  // Elimina "Juan" de la lista
    }
}

System.out.println(nombres);
// [Ana, Pedro, Lucía]
```

Interfaces funcionales predefinidas - Iterator<T>

```java
public class Cosa {

    public String cadena;

    public Cosa(String cadena) {
        this.cadena = cadena;
    }

    @Override
    public String toString() {
        return "Cosa{" + "cadena=" + this.cadena + '}';
    }
}
```

```java
public class Contenedor implements Iterator<Cosa>{

    private ArrayList<Cosa> cosas;
    private int posicion;

    public Contenedor() {
        this.cosas = new ArrayList<>();
        this.posicion = 0;
    }

    public void agregarCosas(Cosa cosa) {

        this.cosas.add(cosa);
    }

    @Override
    public boolean hasNext() {

        return this.posicion < this.cosas.size();
    }

    @Override
    public Cosa next() {

        Cosa unaCosa = this.cosas.get(this.posicion);
        this.posicion++;
        return unaCosa;
    }
}
```

```java
Contenedor contenedor = new Contenedor();

contenedor.agregarCosas(new Cosa("Cosa 1"));
contenedor.agregarCosas(new Cosa("Cosa 2"));
contenedor.agregarCosas(new Cosa("Cosa 3"));

while (contenedor.hasNext()) {
    System.out.println(contenedor.next());
}
// Cosa{cadena=Cosa 1}
// Cosa{cadena=Cosa 2}
// Cosa{cadena=Cosa 3}
```

Interfaces funcionales predefinidas - Iterable<T>

Las interface **Iterable<T>** es más genérica y representa cualquier estructura de datos que se pueda iterar. Las colecciones en Java, como las listas (List), los conjuntos (Set) y los mapas (Map), implementan la interfaz ***Iterable***.

Método principal:

- iterator(): retorna un iterador.

Interfaces funcionales predefinidas - Iterable<T>

A partir de Java 8, la interfaz *Iterable* incluye el método **forEach**, que permite aplicar una acción a cada elemento de la colección. Esto se basa en la funcionalidad de las interfaces funcionales, y forEach acepta una instancia de *Consumer<T>*, lo que facilita el uso de **expresiones lambda**.

```java
List<String> nombres = Arrays.asList("Ana", "Juan", "Pedro", "Lucía");

// for obtiene un iterador
for (String nombre : nombres) {
    System.out.println(nombre);
}

// forEach con lambda
nombres.forEach(nombre -> System.out.println(nombre));
```

Interfaces funcionales predefinidas - Iterable<T>

```java
public class Cosa {

    public String cadena;

    public Cosa(String cadena) {
        this.cadena = cadena;
    }

    @Override
    public String toString() {
        return "Cosa{" + "cadena=" + this.cadena + '}';
    }
}
```

```java
public class Contenedor implements Iterable<Cosa>, Iterator<Cosa>{

    private ArrayList<Cosa> cosas;
    private int posicion;

    public Contenedor() {
        this.cosas = new ArrayList<>();
        this.posicion = 0;
    }

    public void agregarCosas(Cosa cosa) {

        this.cosas.add(cosa);
    }

    @Override
    public boolean hasNext() {

        return this.posicion < this.cosas.size();
    }

    @Override
    public Cosa next() {

        Cosa unaCosa = this.cosas.get(this.posicion);
        this.posicion++;
        return unaCosa;
    }

    @Override
    public Iterator<Cosa> iterator() {

        this.posicion = 0;
        return this;
    }
}
```

```java
Contenedor contenedor = new Contenedor();

contenedor.agregarCosas(new Cosa("Cosa 1"));
contenedor.agregarCosas(new Cosa("Cosa 2"));
contenedor.agregarCosas(new Cosa("Cosa 3"));

for (Cosa cosa : contenedor) {
    System.out.println(cosa);
}
// Cosa{cadena=Cosa 1}
// Cosa{cadena=Cosa 2}
// Cosa{cadena=Cosa 3}
```

Ejercitación