Clases ordenables. Orden natural y órdenes alternativos

El **orden natural** define el *mecanismo por defecto* para ordenar objetos.

- Hace que los objetos, en sí mismos, sean comparables y ordenables.
- La interfaz `Comparable<T>` permite definir el **orden natural**.
- El orden natural se define en la propia clase ordenable (comparable).

```
public class Persona implements Comparable<Persona> {
    public int compareTo(Persona otra) { /* ... */ }          // @Override
}
```

Los **órdenes alternativos** definen *mecanismos alternativos* para ordenar objetos.

- Utiliza objetos auxiliares para comparar otros objetos.
- La interfaz `Comparator<T>` permite definir un **orden alternativo**.
- Cada uno de los órdenes alternativos debe implementarse en una clase aparte diferente (*clase "satélite"*), que proporcione esa funcionalidad.

```
public class OrdenAlt1 implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2) { /* ... */ }    // @Override
}
public class OrdenAlt2 implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2) { /* ... */ }    // @Override
}
```

Sólo es posible definir un **único** orden natural para una determinada clase, aunque se pueden definir **varios** órdenes alternativos.

Orden natural. La interfaz java.lang.Comparable<T>

La interfaz `Comparable<T>` permite definir el **orden natural** para una clase.

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Cuando una clase `T` proporciona el **Orden Natural**, entonces la clase `T` debe implementar la interfaz `Comparable<T>` y definir el método `compareTo`.

- El método `o1.compareTo(o2)` debe devolver:
  - **negativo** si `o1` **es menor que** `o2`.
  - **cero** si `o1` **es igual a** `o2`
  - **positivo** si `o1` **es mayor que** `o2`.
  - Si `o1.equals(o2)` es `true`, entonces `o1.compareTo(o2)` debe devolver **cero**.
  - Atención a las comparaciones de String *sin diferenciar mayúsculas de minúsculas* (*IgnoreCase*).
  - Atención al orden (ascendente o descendente) en las comparaciones de los componentes.
  - Comparación de tipos primitivos utilizando método `compare` de *clases envoltorios*.

```
public class Persona implements Comparable<Persona> {
    // Compara la edad ascendente, la nota descendente, el nombre ascendente-IgnoreCase
    public int compareTo(Persona other) {    // El tipo del parámetro es Persona
        int resultado = Integer.compare(this.edad, other.edad); // Ascendente
        if (resultado == 0) {
            resultado = Double.compare(other.nota, this.nota); // Descendente
            if (resultado == 0) {
                resultado = this.nombre.compareToIgnoreCase(other.nombre); // Ascendente
            }
        }
        return resultado;
    }
```

La interfaz `java.lang.Comparable<T>` en la API de Java

La clase `String` implementa la interfaz `Comparable<T>`, y proporciona los *métodos de instancia* `compareTo(String)` (orden *natural* **lexicográfico**) y `compareToIgnoreCase(String)`.

Las clases *envoltorios* (`Character`, `Boolean`, `Integer`, `Double`, etc) implementan la interfaz `Comparable<T>`, y proporcionan el *método de instancia* `compareTo(T)`.

Además, las clases *envoltorios* también proporcionan el *método de clase* `compare()`, que permiten comparar los tipos primitivos:

- `int Character.compare(char a, char b);`
- `int Boolean.compare(char a, char b);`
- `int Integer.compare(int a, int b);`
- `int Double.compare(double a, double b);`

Las enumeraciones (`enum`) también implementan (proporcionada automáticamente por el lenguaje Java) la interfaz `Comparable<T>`, y proporcionan el *método de instancia* `compareTo(T)`.

Ejemplo 1: clase Persona

```
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public String nombre() { return nombre; }
    public int edad() { return edad; }
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona)o;
            ok = (this.edad == other.edad)&&(this.nombre.equals(other.nombre));
        }
        return ok;
    }
    public int hashCode() {
        return java.util.Objects.hash(this.edad, this.nombre);
        //  return Integer.hashCode(this.edad) + this.nombre.hashCode();
    }
    public int compareTo(Persona other) {
        // Comparación por edad ascendente, y a igualdad de edad, por nombre ascendente
        int resultado = Integer.compare(this.edad, other.edad);
        if (resultado == 0) {
            resultado = this.nombre.compareTo(other.nombre);
        }
        return resultado;
    }
}
```

```
public static void main(String[] args) {
  Persona p1 = new Persona("Juan", 35);
  Persona p2 = new Persona("Pedro", 22);
  System.out.println(p1.compareTo(p2));
}
```

Ejemplo 2: clase Persona (*IgnoreCase*)

```
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public String nombre() { return nombre; }
    public int edad() { return edad; }
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona)o;
            ok = (this.edad == other.edad)&&(this.nombre.equalsIgnoreCase(other.nombre));
        }
        return ok;
    }
    public int hashCode() {
        return java.util.Objects.hash(this.edad, this.nombre.toLowerCase());
        //  return Integer.hashCode(this.edad) + this.nombre.toLowerCase().hashCode();
    }
    public int compareTo(Persona other) {
        // Comparación por edad ascendente, y a igualdad de edad, por nombre ascendente-IgnoreCase
        int resultado = Integer.compare(this.edad, other.edad);
        if (resultado == 0) {
            resultado = this.nombre.compareToIgnoreCase(other.nombre);
        }
        return resultado;
    }
}
```

```
public static void main(String[] args) {
  Persona p1 = new Persona("Juan", 35);
  Persona p2 = new Persona("Pedro", 22);
  System.out.println(p1.compareTo(p2));
}
```

Ejemplo 3: clase Persona (descendente)

```
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public String nombre() { return nombre; }
    public int edad() { return edad; }
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona)o;
            ok = (this.edad == other.edad)&&(this.nombre.equals(other.nombre));
        }
        return ok;
    }
    public int hashCode() {
        return java.util.Objects.hash(this.edad, this.nombre);
        //  return Integer.hashCode(this.edad) + this.nombre.hashCode();
    }
    public int compareTo(Persona other) {
        // Comparación por edad descendente, y a igualdad de edad, por nombre descendente
        // El orden en el que se comparan los componentes es importante
        int resultado = Integer.compare(other.edad, this.edad);
        if (resultado == 0) {
            resultado = other.nombre.compareTo(this.nombre);
        }
        return resultado;
    }
}
```

```
public static void main(String[] args) {
  Persona p1 = new Persona("Juan", 35);
  Persona p2 = new Persona("Pedro", 22);
  System.out.println(p1.compareTo(p2));
}
```

Orden alternativo. La interfaz `java.util.Comparator<T>`

La interfaz `Comparator<T>` permite definir un **orden alternativo** para una clase.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    default Comparator<T> reversed() {...}
    default Comparator<T> thenComparing(Comparator<T>) {...}
    static <T extends Comparable<? super T>> Comparator<T> naturalOrder() {...}
    static <T extends Comparable<? super T>> Comparator<T> reverseOrder() {...}
}
```

Cuando una **clase satelite** proporciona un **Orden Alternativo** sobre otra clase `T`, entonces la clase satélite debe implementar la interfaz `Comparator<T>`, y definir el método **`compare`**.

- El método `sat.compare(o1, o2)` debe devolver:
  - **negativo** si `o1` **es menor que** `o2`.
  - **cero** si `o1` **es igual a** `o2`.
  - **positivo** si `o1` **es mayor que** `o2`.

Es deseable que el método `compare(o1, o2)` **sea consistente** con `o1.equals(o2)`. En caso de que no lo sea, tanto el método `add()` sobre un conjunto ordenado, como el método `put()` sobre una correspondencia ordenada, utilizan el método `compare()` en vez de `equals()` para comprobar la igualdad de elementos o claves.

Ejemplo de uso de java.util.Comparator<T>

```
import java.util.*;
public class OrdenPersona implements Comparator<Persona> {
    // Comparación por nombres, y a igualdad de nombres, por edad
    @Override
    public int compare(Persona p1, Persona p2) {
        int resultado = p1.nombre().compareTo(p2.nombre());
        if (resultado == 0) {
            resultado = Integer.compare(p1.edad(), p2.edad());
        }
        return resultado;
    }
}
```

```
import java.util.*;
public class MainPersona3 {
    public static void main(String[] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);
        Comparator<Persona> op = new OrdenPersona();
        System.out.println(op.compare(p1, p2));
    }
}
```

Composición de órdenes alternativos

```java
public class OrdenNombre implements Comparator<Persona> {
    @Override
    public int compare(Persona p1, Persona p2) {  // Comparación por nombres
        return p1.nombre().compareTo(p2.nombre());
    }
}
public class OrdenEdad implements Comparator<Persona> {
    @Override
    public int compare(Persona p1, Persona p2) {  // Comparación por edad
        return Integer.compare(p1.edad(), p2.edad());
    }
}
import java.util.*;
public class MainPersona4 {
    public static void main(String[] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);

        Comparator<Persona> op1 = new OrdenEdad().thenComparing(new OrdenNombre());
        System.out.println(op1.compare(p1, p2));

        Comparator<Persona> op2 = new OrdenNombre().reversed().thenComparing(new OrdenEdad());
        System.out.println(op2.compare(p1, p2));

        Comparator<Persona> op3 = Comparator.naturalOrder(); // Inferencia de Tipos
        System.out.println(op3.compare(p1, p2));

        Comparator<Persona> op4 = Comparator.<Persona>naturalOrder();
        System.out.println(op4.compare(p1, p2));
    }
}
```

Uso de Comparable<Persona> y Comparator<Persona>

- Podemos definir una Asamblea como un grupo de personas ordenadas.

```
public class Asamblea { // grupo de personas (ordenadas)
    SortedSet<Persona> personas;
    public Asamblea() {
        // Se crea una asamblea que utilizará
        // el orden natural de Persona para ordenar
        personas = TreeSet<>();
    }
    public Asamblea(Comparator<Persona> comp) {
        // Se crea una asamblea que utilizará
        // el orden alternativo de Persona para ordenar
        personas = TreeSet<>(comp);
    }
}
```

- Podemos utilizarla donde sea necesaria.

```
Asamblea a = new Asamblea();
Asamblea a = new Asamblea(new OrdenPersona());
Asamblea a = new Asamblea(new OrdenEdad().thenComparing(new OrdenNombre()));
// ...
```