iSIL Instituto San Ignacio
Aprende haciendo

Curso	Desarrollo de Software I
Tema	Java Collection Framework – Parte II
Semana	Nro. 02
Docente	Ing. Eric Gustavo Coronel Castillo

Objetivos

- Aplicar la interfaz Comparable y Comparator para ordenar colecciones de datos y hacer búsquedas.

Interfaz Comparable

Las clases que implementan esta interfaz cuentan con un **"orden natural"**. Este orden es total, es decir, siempre han de poder ordenarse dos objetos cualesquiera de la clase que implementa esta interfaz. La interfaz Comparable declara el método compareTo() de la siguiente forma:

```
public int compareTo(Object obj)
```

Este método compara su argumento implícito (this) con el objeto que se le pasa como parámetro (**obj**), retorna un entero negativo, cero o positivo según que el argumento implícito sea anterior, igual o posterior al objeto **obj**.

Las estructuras cuyos objetos tengan implementado el interfaz Comparable podrán ejecutar ciertos métodos basados en el orden tales como sort o binarySearch.

Si queremos programar el método compareTo() debemos hacerlo con cuidado, ha de ser coherente con el método equals() y ha de cumplir la propiedad transitiva.

Interfaz Comparator

Si una clase ya tiene una ordenación natural y se desea realizar una ordenación diferente, por ejemplo descendente, dependiente de otros campos o simplemente requerimos varias formas de ordenar una clase, haremos que una clase distinta de la que va a ser ordenada implemente este interfaz.

Su principal método se declara en la forma:

```
public int compare(Object o1, Object o2)
```

El método *compare()* devuelve un entero negativo, cero o positivo según su primer argumento sea anterior, igual o posterior al segundo (Así asegura un oren ascendente).

Es muy importante que *compare()* sea compatible con el método *equals()* de los objetos que hay que mantener ordenados.

Los objetos que implementa esta interfaz pueden ser utilizados en las siguientes situaciones (especificando un orden distinto al natural):

- Como argumento a un constructor *TreeSet* o *TreeMap*, con la idea de que las mantengan ordenadas de acuerdo con dicho *Comparator*.
- Collections.sort(List, Comparator), Arrays.sort(Object[], Comparator)
- Collections.binarySearch(List, Object, Comparator), Arrays.binarySearch(Object [] v, Object key, Comparator c) Object también debe ser implementado.

¿Cuándo usar cada uno?

En una colección de objetos, éstos pueden ser ordenados por diferentes criterios. Dependiendo de la clase que realiza la comparación, se implementará la interfaz *Comparator* o *Comparable*.

Usaremos *Comparable* para definir el orden natural de una clase **C**, entendiendo por orden natural aquel que se utilizará normalmente o simplemente por convenio. Así, diremos que los objetos de clase **C** son comparables.

Por otro lado, implementaremos nuevas clases (**C1** … **Cn**) que extiendan el interfaz *Comparator* por cada ordenación nueva que necesitemos distinta a la natural para la clase **C**. Así tendremos una "librería de comparadores" (**C1** … **Cn**) para la clase **C**.

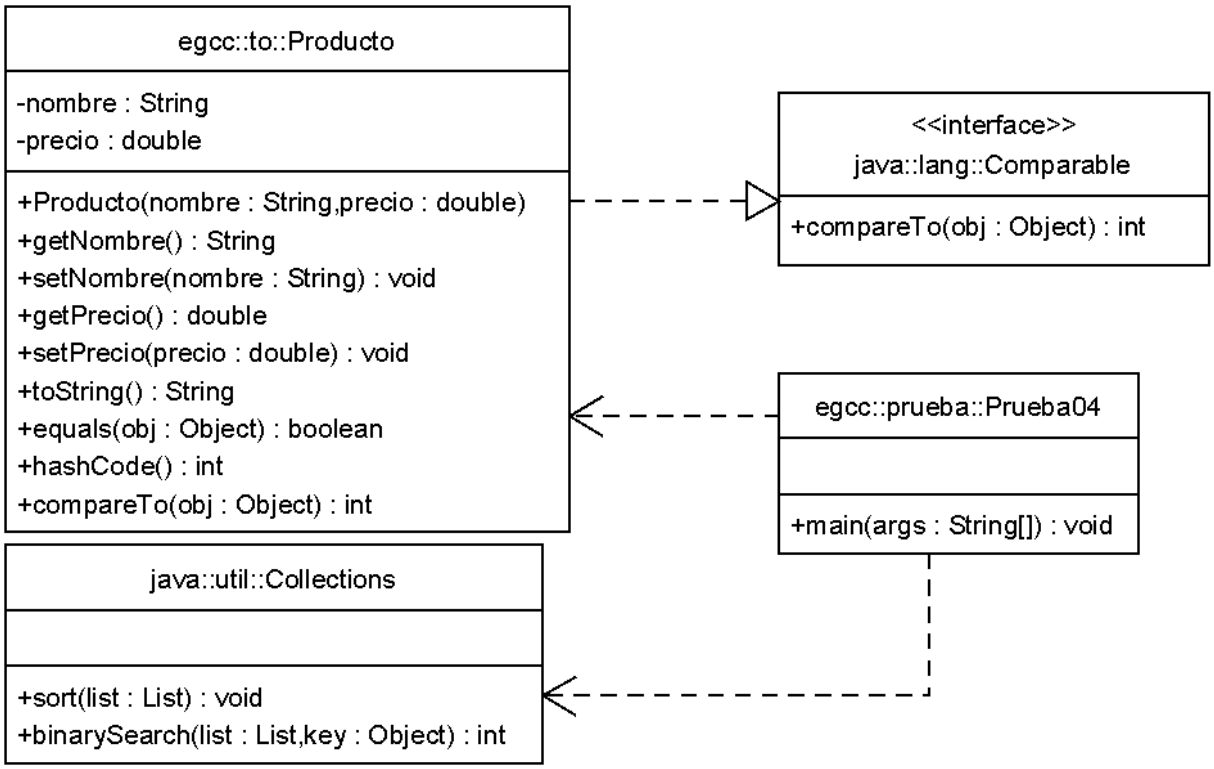
Ejemplo Ilustrativo

La clase *Collections* (que no es la interfaz *Collection*) nos permite ordenar y buscar elementos en listas.

Para ordenar los elementos de la lista se utiliza el método *sort()* y para buscar un elemento se utiliza el método *binarySearch()*.

Los objetos de la lista deben tener métodos *equals()*, *hashCode()* y *compareTo()* adecuados.

La siguiente figura muestra el diagrama de clases de la implementación de ordenamiento y búsqueda utilizando la clase *Collentions*.



Producto.java

```
package egcc.to;

public class Producto implements Comparable {
    // Campos
    private String nombre = null;
    private double precio;
    // Constructor
    public Producto(String nombre, double precio) {
        this.setNombre(nombre);
        this.setPrecio(precio);
    } // Producto
    // Métodos
    public String getNombre() {
        return nombre;
```

```
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}
public double getPrecio() {
    return precio;
}
public void setPrecio(double precio) {
    this.precio = precio;
}
@Override
public String toString() {
    return this.getNombre() + " - " + this.getPrecio();
}
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (obj instanceof Producto) {
        Producto producto = (Producto) obj;
        return this.getNombre().equals(producto.getNombre());
    } else {
        return false;
    }
}
@Override
public int hashCode() {
    return this.getNombre().hashCode();
}
public int compareTo(Object obj) {
    // Indica en base a que atributos se compara el objeto
    // Devuelve +1 si this es > que objeto
    // Devuelve -1 si this es < que objeto
    // Devuelve  0 si son iguales
    Producto producto = (Producto) obj;
    String nombreObj = producto.getNombre().toLowerCase();
```

```
        String nombreThis = this.getNombre().toLowerCase();
        return (nombreThis.compareTo(nombreObj));
    }
} // Producto
```

Prueba04.java

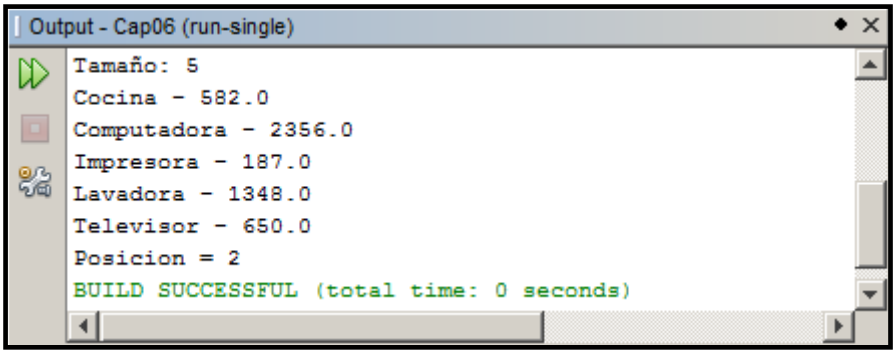
```
package egcc.prueba;

import egcc.to.Producto;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class Prueba04 {
    public static void main(String[] args) {
        // Creación de la lista
        List lista = new ArrayList<Producto>();
        // Creación de los productos
        Producto prod01 = new Producto("Televisor", 650.0);
        Producto prod02 = new Producto("Impresora", 187.0);
        Producto prod03 = new Producto("Computadora", 2356.0);
        Producto prod04 = new Producto("Lavadora", 1348.0);
        Producto prod05 = new Producto("Cocina", 582.0);
        // Agregar productos a la lista
        lista.add(prod01);
        lista.add(prod02);
        lista.add(prod03);
        lista.add(prod04);
        lista.add(prod05);
        // Ordenar lista
        Collections.sort(lista);
        // Mostrar tamaño de la lista
        System.out.println("Tamaño: " + lista.size());
        // Mostrar lista
        Iterator it = lista.iterator();
        while (it.hasNext()) {
```

```
            System.out.println(it.next().toString());
        }
        // Buscar en la lista
        Producto dato = new Producto("impresora", 0);
        int posicion = Collections.binarySearch(lista, dato);
        System.out.println("Posicion = " + posicion);
    } // main
} // Prueba04
```

A continuación tenemos el resultado de la ejecución de este ejemplo:



Para ordenar la lista se está utilizando el método *sort()* de la clase *Collections*, y para la búsqueda de un elemento se está utilizando el método *binarySearch()* de la misma clase.

Al método *binarySearch()* se le pasa como parámetro la lista y un objeto de tipo *Producto* con el nombre del artículo que se quiere buscar, en este caso no interesa el valor del campo precio, y esto por la forma como está implementado el método *compareTo()* en la clase *Producto*, donde la comparación solo se realiza en base al nombre del producto y descarta el valor del precio.

Referencia

Desarrollando Soluciones con Java y MySQL Server
Eric Gustavo Coronel Castillo