Colecciones de objetos

Java tiene desde la versión 1.2 todo un juego de clases e interfaces para guardar **colecciones** de **objetos**. En él, todas las entidades conceptuales están representadas por interfaces, y las clases se usan para proveer implementaciones de esas interfaces.

La interfaz nos dice qué podemos hacer con un objeto. Un objeto que cumple determinada interfaz es "algo con lo que puedo hacer X". La interfaz no es la descripción entera del objeto, solamente un mínimo de funcionalidad con la que debe cumplir.

Como corresponde a un lenguaje tan orientado a objetos, estas clases e interfaces están estructuradas en una jerarquía. A medida que se va descendiendo a niveles más específicos aumentan los requerimientos y lo que se le pide a ese objeto que sepa hacer.

Collection

La interfaz más importante es Collection. Una **Collection** es todo aquello que **se puede recorrer (o "iterar") y de lo que se puede saber el tamaño**. Muchas otras clases extenderán Collection imponiendo más restricciones y dando más funcionalidades.

No puedo construir una Collection. No se puede hacer "new" de una Collection, sino que todas las clases que realmente manejan colecciones "son" Collection, y admiten sus operaciones.

Las operaciones básicas de una collection entonces son:

- **add(T)**

  Añade un elemento.

- **iterator()**

Obtiene un "iterador" que permite recorrer la colección visitando cada elemento una vez.

- **size()**

  Obtiene la cantidad de elementos que esta colección almacena.

- **contains(t)**

  Pregunta si el elemento *t* ya está dentro de la colección.

Una capacidad de **un objeto Collection es la de poder ser recorrido**. Como a este nivel no está definido un orden, la única manera es proveyendo un iterador, **mediante el método iterator().** Un iterador es un objeto "paseador" que **nos permite ir obteniendo todos los objetos al ir invocando progresivamente su método next().** También, si la colección es modificable, podemos remover un objeto durante el recorrido mediante el método remove() del iterador. El siguiente ejemplo recorre una colección de Integer borrando todos los ceros:

```
void borrarCeros(Collection<Integer> ceros)
{
        Iterator<Integer> it = ceros.iterator();
        while(it.hasNext())
        {
                int i = it.next();
                if(i == 0)
                        it.remove();
        }
}
```

En este ejemplo hago uso de la conversión automática entre Integer e int.

Cómo ordenar los elementos de una colección

La clase Collection también tiene un método **sort** para ordenar los elementos de una lista según el orden natural y para ordenar los elementos de la lista según el criterio de un *Comparator*.

A tener en cuenta que tanto los métodos *sort* de *Arrays* como de *Collections* no devuelven una nueva instancia de *array* o colección ordenada sino que modifican la instancia de *array* o colección que se proporciona para ordenar.

Para poder ordenar objetos Collection con el método sort por un determinado atributo, necesitamos la interfaz **Comparable**.

El método **sort()** no sabe cómo ordenar un objeto, o sea, no sabe qué parámetro o criterio debe utilizar para comparar por ejemplo, alumnos. Sin embargo, podemos informarle esto implementando la interfaz **Comparable** en la clase Alumno:

Partiendo de la clase Alumno:

```
public class Alumno {
   private final String nombre;
   private int puntos;
   public Alumno(String nombre, int puntos) {
      this.nombre = nombre;
      this.puntos = puntos;
   }
   //métodos
}
```

Entonces vamos a crear a nuestros alumnos:

```
Alumno alumno1 = new Alumno("Alex Felipe", 13450);
Alumno alumno2 = new Alumno("Maurício Aniche", 19930);
Alumno alumno3 = new Alumno("Guilherme Silveira", 23143);
Alumno alumno4 = new Alumno("Rodrigo Turini", 13500);
```

Ahora necesitamos una **List** para almacenar estos alumnos:

```
List<Alumno> alumnos = new ArrayList<Alumno>();
alumnos.add(alumno1);
```

```
alumnos.add(alumno2);
alumnos.add(alumno3);
alumnos.add(alumno4);
```

Probemos nuestra lista:

```
System.out.println(alumnos);
```

Resultado:

```
[Alex Felipe - 13450, Maurício Aniche - 19930, Guilherme Silveira - 23143, Rodrigo Turini - 13500]
```

Tengo mi lista con los alumnos. Ahora vamos a ordenarla con el método estático sort() de la clase **Collections()**.

```
Collections.sort(alumnos);
```

El código no compila. El problema es que el método **sort()** no sabe cómo ordenar un alumno, o sea, no sabe qué parámetro o criterio debe utilizar para comparar alumnos. Sin embargo, podemos informarle de esto implementando la interfaz **Comparable** en la clase Alumno:

```
public class Alumno implements Comparable<Alumno> {
   //atributos y métodos
   @Override
   public int compareTo(Alumno otroAlumno) {
      //implementación
   }
}
```

Cuando implementamos la interfaz Comparable, necesitamos pasar como parámetro un objeto con el que queremos comparar, en este caso, queremos comparar objetos de tipo Alumno.
Por último, necesitamos implementar el método **compareTo()**. La regla de ordenación utilizando el método compareTo() compara dos objetos (alumnos) y funciona de la siguiente manera (mayor a menor puntos):

- Para colocar al alumno más a la izquierda de la lista, retornamos -1.

- Para colocar al alumno más a la derecha de la lista, retornamos 1.
- Cuando retornamos 0, significa que los alumnos comparados son iguales y no alteran sus posiciones.

En nuestro ejemplo:

```
@Override
public int compareTo(Alumno otroAlumno) {
    if (this.puntos > otroAlumno.getPuntos()) {
      return -1;
    }
    if (this.puntos < otroAlumno.getPuntos()) {
      return 1;
    }
    return 0;
}
```

Indicamos que si los puntos del alumno son mayores, muévalo más a la izquierda (-1), si son menores, muévalo a la derecha (1) y si los alumnos son iguales, no haga nada (0).

Después de implementar Comparable, ¡nuestro código compila! Vamos a probarlo:

```
Collections.sort(alumnos);
System.out.println(alumnos);
```

Resultado:

```
[Guilherme Silveira - 23143, Maurício Aniche - 19930, Rodrigo Turini - 13500, Alex Felipe - 13450]
```

¡Observa que ahora nuestros alumnos están ordenados!

A diferencia de ordenar números, para ordenar listas de objetos específicos, como en el caso del objeto alumno, es necesario especificar cómo se debe ordenar el objeto implementando la interfaz Comparable y su método compareTo().

Si queremos cambiar el orden de ordenación podemos utilizar el método estático reverseOrder() de la clase **Collections()**.

```
Collections.reverseOrder(alumnos);
```

En este caso el ArrayList de alumnos quedaría ordenado de forma inversa.

NOTA: Recordando el método compareTo de String

```
public int compareTo(String str)
```

El método devuelve un tipo de datos int que se basa en la comparación lexicográfica entre dos cadenas.

- devuelve < 0, si la cadena que llama al método es lexicográficamente menor que la cadena pasada como parámetro
- devuelve 0, si las dos cadenas son lexicográficamente equivalentes
- devuelve > 0, si la cadena que llama al método es lexicográficamente mayor que la cadena pasada como parámetro

Ejemplo:

```
String ejemplo=”Ana es mi amiga”;
if (ejemplo.compareTo(“Pedro también es mi amigo”) == 0) {
    ……
}
```

En este caso la comparación nos devolvería -1