Ordenación de colecciones de objetos.

Revisión 1.1

Introducción.

Debido a que en los últimos tiempos esta parece ser una de las preguntas más frecuentes que aparecen, me he decidio a escribir este pequeño ejemplo sobre como ordenar elementos. Este ejemplo no trata de ser una *guía de buena programación*, si no que trata simplemente de enseñar la manera de ordenar fácilmente colecciones de elementos.

Como toda persona que trabaja con Java debería saber, el API de Java es muy rico, y contiene ya mucho del código genérico que nuestra aplicación pueda necesitar. Solo hay que saber buscarlo.

Este es el caso de la ordenación de elementos, que viene ya implementada, con un algoritmo de los más eficientes, el *mergesort*. Además, date cuenta de que, como dice el título, este artículo trata el ordenamiento de **objetos**, no elementos primitivos.

Un poco de teoría: el interface `java.lang.Comparable`.

Gran parte de la *magia* en la ordenación de objetos que ofrece Java2 es culpa del buen diseño del API *Collections*, que ofrece el conjunto de funcionalidad básica para tratamientos de conjuntos de elementos. Este API es capaz de ordenar todos los objetos de clases que implementen el interface `java.lang.Comparable`, que contiene un solo método:

Method Summary	
`int`	**`compareTo`**`(Object o)` Compares this object with the specified object for order.

El entero que devuelve este método será el equivalente a `objeto1 - objeto2`, es decir:

```
  objecto1.comparteTo(objeto2);

      negativo si objeto1 < objeto2
      cero     si objeto1 = objeto2
      positivo si objeto1 > objeto2
```

Lo que decida si un objeto es mayor o menor que otro es lo que tenemos que hacer nosotros. Por supuesto gran parte de las clases básicas del API de Java, como `String, Integer, File, Date` y demás, ya la implementan, con el orden esperado (alfabético, ordinal, cronológico, etc), así que solo tendremos que preocuparnos de este interface con nuestras clases propias más complejas, como por ejemplo, Usuario:

```
class Usuario{

  private String nombre;
  private int edad;

  Usuario(String nombre, int edad) {
```

```
    this.nombre = nombre;
    this.edad = edad;
  }

  public String getNombre() {
    return nombre;
  }

  public int getEdad() {
    return edad;
  }

  public String toString() {
    return nombre + " (" + edad + ")";
  }
}
```

Nosotros tendremos que decidir que hace que un `Usuario` vaya antes que otro, pongamos por ejemplo el orden alfabético.

```
class Usuario1 implements Comparable {

  private String nombre;
  private int edad;

  Usuario1(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  public String getNombre() {
    return nombre;
  }

  public int getEdad() {
    return edad;
  }

  public String toString() {
    return nombre + " (" + edad + ")";
  }

  public int compareTo(Object o) {
    Usuario1 otroUsuario = (Usuario1) o;
    //podemos hacer esto porque String implementa Comparable
    return nombre.compareTo(otroUsuario.getNombre());
  }
}
```

Ordenando: la clase `java.util.Collections`.

Como he dicho antes, muchas de las cosas más básicas (y muchas otras no tan básicas) que podamos necesitar en nuestros programas Java ya estan implementadas. En el caso de las colecciones de elementos se encuentran en el paquete `java.util`. Allí podreís encontrar todas las clases para mantener grupos de elementos (listas, pilas, árboles, etc), y como no, clases de utilidad. Eso es lo que ahora necesitamos, una clase de utilidad: `java.util.Collections`, no confundir con el interface `java.util.Collection`, base de (casi) todas las colecciones de objetos.

Esta clase contiene muchos métodos, para hallar el elemento más pequeño o el más grande de una colección, para *barajearlos* (cambiar su orden de forma aleatoria), para ponerlos en orden inverso al actual, etc. Solo teneis que consultar el API.

Por supuesto esta clase tiene un método para ordenar colecciones de elementos, y es eso lo que nos ocupa ahora.

Method Summary	
static void	**sort**(List list) Sorts the specified list into ascending order, according to the *natural ordering* of its elements.

Como veis este método puede ordenar cualquier colecion que implemente el interface `java.util.List`, y ese *orden natural* se refiere al orden que indican el método `comparteTo` que hemos visto antes.

Un ejemplo para ordenar distintos `java.lang.String` sería de la siguietne forma:

```
ArrayList lista = new ArrayList();
lista.add("uno");
lista.add("dos");
lista.add("tres");
lista.add("cuatro");
printList(lista);
System.out.println("\n Ahora ordenados...");
Collections.sort(lista);
//metodo que imprime la lista
printList(lista);
```

Y para ordenar nuestros queridos `Usuario` haríamos algo como esto

```
ArrayList lista = new ArrayList();
lista.add(new Usuario1("uno", 11));
lista.add(new Usuario1("dos", 2));
lista.add(new Usuario1("tres", 3));
lista.add(new Usuario1("cuatro", 44));
printList(lista);
System.out.println("\n Ahora ordenados...");
Collections.sort(lista);
//metodo que imprime la lista
printList(lista);
```

Cambiando el orden: el interface `java.util.Comparator`.

Hasta aquí hemos visto lo fácil que era todo, pero aún nos queda *la parte complicada*, ¿qué hacemos si queremos ordenar nuestros `Usuario` por edad en lugar de nombre?. Eso no nos lo permite hacer el interface `java.lang.Comparable`, así que consultamos la documentacion de nuestra clase de utilidades `java.util.Collections` y descubrimos el siguiente método:

Method Summary	
static void	**sort**(List list, Comparator c) Sorts the specified list according to the order induced by the specified

	comparator.

Así que nos pica la curiosidad y vamos a ver que es eso de `java.util.Comparator`, y descubrimos que es otro interface que se usa para comparar objetos, y que tiene los siguientes métodos:

Method Summary	
int	**compare**(Object o1, Object o2) Compares its two arguments for order.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this Comparator.

El método `equals(Object obj)` no nos interesa, es el que como en la clase `java.lang.Object` determina si dos objetos son el mismo o no, el que nos interesa es el otro, `compare(Object o1, Object o1)`.

El funcionamiento es el mismo que el visto anteriormente en el interface comparable, solo que en esta ocasión se pasan los dos objetos a comparar como argumentos. El resultado será:

```
negativo si o1 < o2
cero     si o1 = o2

positivo si o1 > o2
```

o al menos eso es lo que se espera, porque nosotros podemos decidir. El método `sort` de la clase `java.util.Collections` ordenadrá en función de este resultado, pero somos nosotros quién decide ese orden.

Podríamos hacer que nuestras clases implementasen `java.util.Comparator`, pero entonces no habríamos avanzado mucho desde el ejemplo anterior, es decir, estaríamos limitados a un orden, por lo que lo normal será escribir distintas clases que implementen este interface para después usar la que necesitemos en cada momento. Para ordenar nuestra clase `Usuario` escribiremos dos clases, una para ordenarlos por nombre, y otra por edad.

```
import java.util.Comparator;

class NombreComparator implements Comparator {

  public int compare(Object o1, Object o2) {
    Usuario u1 = (Usuario) o1;
    Usuario u2 = (Usuario) o2;
    return u1.getNombre().compareTo(u2.getNombre());
  }

  public boolean equals(Object o) {
    return this == o;
  }
}
import java.util.Comparator;

class EdadComparator implements Comparator {

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

```
    Usuario u1 = (Usuario) o1;
    Usuario u2 = (Usuario) o2;
    return u1.getEdad() - u2.getEdad();
  }

  public boolean equals(Object o) {
    return this == o;
  }
}
```

Ahora solo nos queda usarlos, nada más fácil

```
    ArrayList lista = new ArrayList();
    lista.add(new Usuario("uno", 11));
    lista.add(new Usuario("dos", 2));
    lista.add(new Usuario("tres", 3));
    lista.add(new Usuario("cuatro", 44));
    printList(lista);
    System.out.println("\n Ahora ordenados por nombre...");
    Collections.sort(lista, new NombreComparator());
    //metodo que imprime la lista
    printList(lista);
    System.out.println("\n y ahora ordenados por edad...");
    Collections.sort(lista, new EdadComparator());
    //metodo que imprime la lista
    printList(lista);
```

¿Y que hago con los arrays?.

Adivina adivinanza: `java.util.Arrays`

Código de los ejemplos.

Descargate el código con los ejemplos de los artículos de la siguiente dirección: http://www.javahispano.com/download/ejemplos/ordenar.zip

Anexo de **Martín Pérez**:

Quería añadir un matiz al artículo de ordenación y en partícular a la ordenación de Strings.

En el artículo no se ha tenido en cuenta que el método compareTo() para Strings realiza la ordenación basándose en los códigos ASCII. Por lo tanto la salida no será correcta ya que :

- No va a tratar bien los acentos ni letras como la ñ
- Va a ordenar mal los números Ej : 1,10,110,2,3,4,45,5,....
- Las mayúsculas tienen código ASCII menor, por lo que saldrán antes que todas las entradas con minúsuculas, etc..

Por lo tanto si lo que quieres hacer es ordenar Strings yo te sugeriría algo como esto :

```
 Collator esCollator = Collator.getInstance(new Locale("es", "ES",
"EURO"));
 // EURO es para estar al dia
 Arrays.sort(myArray,esCollator);
```

Nota : Collator implementa la interfaz Comparator

¿ Ventaja frente a currarte tu propio compareTo() para ordenar Strings ?
Si en vez de coger el Locale Español coges el de la máquina tu programa ordenará bien en cualquier lenguaje.

Alberto Molpeceres es ahora mismo desarrollador de aplicaciones en ámbito cliente/servidor para la empresa **T-Systems - debis Systemhaus** en Munich (Alemania). Cuando no está trabajando o "metiendo caña" al resto de los integrantes de javaHispano intenta pasear con su novia, buscar la desaparecida lógica del idioma alemán o intenta olvidar la *pesadilla* que es buscar piso en Munich.
Para cualquier duda o tirón de orejas, e-mail a: al@javahispano.com