Estructuras de datos en Java : Collections

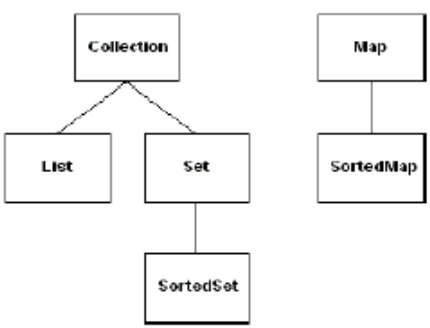
Martín Pérez Mariñán
martin@kristalnetworks.com

Una de las necesidades de cualquier lenguaje de programación es que proporcione una serie de utilidades que permitan al desarrollador crear estructuras de datos. El lenguaje Java mediante el API Collections pone a nuestra disposición un conjunto de clases destinadas a crear estructuras de datos que podremos utilizar en todas nuestras aplicaciones. En este artículo veremos las características fundamentales de este API y algunos trucos para mejorar su rendimiento.

Introducción al API Collections

El API Collections fue introducido con la versión 1.2 de Java 2 Standar Edition y su objetivo fundamental es proporcionar una serie de clases e interfaces que nos permitan manejar diversos conjuntos de datos de una manera estandar. Dentro de esta API nos vamos a encontrar tanto con estructuras de datos como con algoritmos para utilizar dichas estructuras.

La base del API Collections está formada por un conjunto de interfaces para trabajar con conjuntos de datos. Entender el funcionamiento de estas interfaces significará entender el funcionamiento de la totalidad de la API. En la siguiente figura podemos ver la estructura de estas interfaces.



Como se puede apreciar en la figura tenemos dos raices en la jerarquía : Collection y Map. Esto es debido a que existe una ligera diferencia de funcionamiento entre ambas, básicamente y sin entrar en detalles podríamos decir que una Collection trabaja sobre conjuntos de elementos singulares mientras que un Map trabaja sobre pares de valores, por lo tanto existen pequeñas diferencias entre los métodos necesarios por unas y otras; veremos más detalles en los siguientes apartados.

La interfaz Collection

Esta interfaz como vimos anteriormente es una de las raices de la jerarquía. Representa un conjunto de objetos de una manera generalizada. Esta interfaz define una serie de métodos que serán los que el resto de interfaces, o clases que nosotros realicemos, deban implementar :

- Métodos para agregar y elminar elementos

  boolean add(Object element)
  boolean remove(Object element)

- Métodos para realizar consultas

  int size()
  boolean isEmpty()
  boolean contains(Object element)

- Métodos para recorrer todos los elementos

  Iterator iterator()

- Métodos para realizar varias operaciones simultáneamente

  boolean containsAll(Collection collection)
  boolean addAll(Collection collection)
  void clear()
  void removeAll(Collection collection)
  void retainAll(Collection collection)

Estos métodos tienen un nombre bastante intuitivo que nos indica fácilmente la operación que realizan. Como se puede apreciar podemos añadir un elemento (**add**) o añadir varios elementos (**addAll**), así como eliminarlos (**remove/removeAll)**. Podemos consultar si uno o más elementos se encuentran en el conjunto de datos (**contains/containsAll**) así como ver el número de elementos de que disponemos (**size**) o ver si no tenemos ninguno (**isEmpty**).

Podemos eliminar todos los elementos (**clear**) y obtener una vista especial **iterator** para recorrer la colección de datos (lo veremos más adelante). El método **retainAll** elimina del conjunto de datos todos los elementos excepto los especificados.

Como se ve en el título de este apartado, Collection es una interfaz, es decir, no proporciona ninguna implementación concreta. La base para las clases concretas del API se encuentra pues en otro lugar diferente. Este lugar es la clase **AbstractCollection**. Esta clase proporciona implementaciones por defecto para la gran mayoría de los métodos definidos por la interfaz de Collection. Por lo tanto si queremos crear nuestra estructura de datos tenemos dos posibilidades :

- Crear una clase que implemente todos los métodos de Collection.
- Crear una clase que extienda a AbstractCollection y proporcionar la implementación para los métodos necesarios, como por ejemplo size().

A continuación veremos el conjunto de interfaces que heredan su funcionamiento de Collection.

La interfaz List

Esta interfaz define un conjunto de datos ordenados permitiendo elementos duplicados. Añade operaciones a nivel de índice de los elementos así como la posibilidad de trabajar con una parte de la totalidad de la lista.

A continuación podemos ver el conjunto de métodos que nos permiten acceder a elementos por su posición :

```
void add(int index, Object element)
boolean addAll(int index, Collection collection)
Object get(int index)
int indexOf(Object object)
int lastIndexOf(Object object)
Object remove(int index)
Object set(int index, Object element)
```

Como vemos tenemos operaciones para añadir uno o varios elementos (**add/addAll**), para obtener un elemento determinado (**get**), para obtener el índice de un elemento (**indexOf/lastIndexOf**) o para modificar o borrar un elemento (**remove/modify**).

Como he dicho antes, también disponemos de métodos que nos permiten trabajar con una parte de la lista :

```
ListIterator listIterator()
ListIterator listIterator(int comiento)
List sublist(int inicio, int fin)
```

Los dos primeros métodos nos devuelven una vista de la colección de datos (**Iterator**) que nos permite recorrer sus elementos. La interfaz **ListIterator** extiende a la ya mencionada **Iterator** permitiendonos realizar recorridos bidireccionales así como añadir o modificar elementos de la colección a la que representa. Veamos un ejemplo :

```
List lista = .....
ListIterator iterador = list.listIterator(list.size());
while (iterator.hasPrevious()) {
    Object element = iterator.previous();
    System.out.println(element);
}
```

Como vemos en el ejemplo hemos creado un iterador cuyo cursor se encuentra en la última posición de la lista y a continuación recorremos la lista en orden inverso mostrando los elementos por pantalla. Como se puede apreciar, la combinación de **List** junto con **ListIterator** nos proporciona una forma muy elegante de recorrer este tipo de estructuras de datos.

Nuevamente, como sucedía con la clase Collection, List tan sólo es una interfaz, es decir tan sólo define el conjunto de métodos que las implementaciones concretas han de proporcionar. En el API Collections disponemos de dos implementaciones concretas : **ArrayList** y **LinkedList**. Vamos a tratar cada una de estas clases con algo más de detalle, pero antes hablemos de otra clase.

La clase Vector

Antes de aparecer la API Collections, en la versión 1.1 de Java, disponíamos de dos clases que nos permitían manejar estructuras de datos : **HashTable** y **Vector**.

Vector es el equivalente en la versión 1.1 a lo que es ArrayList en la actualidad por lo tanto deberíamos siempre utilizar ArrayList. Vector se mantiene en la actualidad por compatibilidad con miles y miles de lineas de código que han sido creadas utilizándolo. En realidad las clases son muy similares, prácticamente iguales, con una pequeña diferencia que es fundamental : **Vector es thread-safe**. ¿ Qué quiere decir esto ?. Es muy sencillo, simplemente que es una clase que tiene todos sus métodos sincronizados, es decir, dos o más hilos al mismo tiempo no podrán ejecutar un método sobre un vector determinado.

Esto que en un principio puede parecer algo bueno en realidad tiene un gran coste computacional ya que además de no permitir operaciones simultáneas sobre una estructura de datos, requiere el estar continuamente trabajando con monitores y bloqueos lo que redunda en un muy pobre rendimiento.

La clase ArrayList

Para solucionar esto, en el API Collections se introdujo ArrayList que no tiene ningún método sincronizado, soportando por lo tanto el acceso concurrente a los datos. Además, si necesitamos hacer que nuestra estructura de datos sea segura con respecto a otros hilos, podremos sincronizarla a posteriori utilizando los métodos de utilidad proporcionados en la clase Collections como veremos más adelante.

Internamente ArrayList representa la estructura de datos utilizando un array de objetos. Este array comienza con un tamaño determinado que podemos especificar en el constructor y va creciendo dinámicamente a medida que vamos añadiendo elementos. Es decir, en el momento en que ya no cojan más elementos en nuestro array, la clase ArrayList aumentará su tamaño en un 50% o 100% dependiendo del SDK.

Optimización : Leyendo este párrafo anterior es fácil darse cuenta de que si nuestra estructura es medianamente grande el coste de aumentar el tamaño del array que guarda los elementos (al aumentar el tamaño no sólo habrá que crear un nuevo array del doble de tamaño sino que habrá que mover todos los elementos) crece considerablemente. Sin embargo, muchas veces sabemos a priori el tamaño que va a tener nuestro ArrayList o al menos podemos realizar una estimación, por lo tanto en estos casos es siempre recomendable utilizar el constructor **ArrayList(int size)** que creará un array con la capacidad especificada evitándonos todo ese trasiego de información.

Otro factor muy importanto ligado con el almacenamiento de elementos en un array es que al insertar un elemento tendremos que mover todos los elementos que se encontraban a partir de la posición en donde vamos a insertar, una posición más a la derecha. De esto se deduce que las inserciones al inicio son las más costosas y que las inserciones al final del array son las menos costosas.

Optimización : Insertar elementos siempre que podamos al final de la lista ya que es mucho más rápido.

Por último decir que al estar internamente los datos en un array el acceso indexado a elementos de la estructura es realmente rápido siendo este el fuerte del ArrayList. Por lo tanto si vamos a tener en nuestra aplicación un acceso indexado a los datos la estrucutra a elegir es claramente ArrayList. Sin embargo si en nuestra aplicación vamos a tener una gran cantidad de inserciones o borrados en diferentes zonas de la estructura entonces quizás nos debamos decantar por la siguiente clase.

La clase LinkedList

Esta lista está implementada utilizando una lista de nodos doblemente enlazada. En este caso para acceder a un elemento determinado deberemos recorrer todos los elementos previos de la lista, por consiguiente el rendimiento en acceso por índice a elementos es bastante peor que en un ArrayList. Por otra parte, en este caso el coste de eliminar o insertar un elemento en una determinada posición se reduce al coste de buscar dicho elemento, siendo muy pequeño cuando estas operaciones se realizan al principio de la lista y algo mayor al realizarse al final. Con esta clase nos evitamos pues el coste de andar moviendo datos o de tener que aumentar el tamaño de nuestra estructura mientras perdemos rendimiento realizando la búsqueda de los elementos.

Otra de las ventajas que nos puede ofrecer LinkedList es la intuitividad con la que podemos crear nuestras listas y colas (aunque con ArrayList sea también sencillo). Por ejemplo, si quisiésemos realizar una pila :

```
public class Pila extends LinkedList {
    public void push(Object elemento) {
        this.addFirst(elemento);
    }
    public Object pop() {
        return this.removeFirst();
    }
}
```

Como ejercio podéis intentar crear una cola, es igual de sencillo.

En cuanto a las listas por último decir que existen dos clases abstractas : **AbstractList** y **AbstractSequentialList** que proporcionan un implementación por defecto de algunos de los métodos de la interfaz List. Utilizaremos estas clases cuando querramos crear nuestras propias implementaciones de listas, aprovechando de esta manera los métodos que nos proporcionan. Por otra parte otra de las cosas que añaden estas clases abstractas es que sobreescriben los métodos **equals()** y **hashCode()** de modo que dos listas sean iguales cuando contienen los mismos elementos y en el mismo orden además de tener el mismo tamaño.

Para terminar veamos un ejemplo sencillo :

```
import java.util.*;

public class Ejemplo {
      public static void main(String[] args) {
            List list = new ArrayList();
            list.add("María");
            list.add("Rosa");
            list.add("Jose");
            list.add("Pedro");
            System.out.println(list);
            List list2 = new LinkedList(list);
            Iterator it = list2.iterator();
            while (it.hasNext()) {
                  System.out.print(it.next() + " ");
            }
      }
}
```

La salida de este ejemplo es la siguiente :

```
[María, Rosa, Jose, Pedro]
María Rosa Jose Pedro
```

La interfaz Set

Esta interfaz extiende a la interfaz Collection y no añade ningún nuevo método. Representa un conjunto de datos en el más puro estilo matemático, es decir, no se admiten duplicados. Por lo tanto si añadimos un elemento a un conjunto y ese elemento ya se encontraba en su interior no se producirá ningún cambio en la estructura de datos.

Para determinar si un dato se encuentra o no en el conjunto, las implementaciones que vienen en el API de Collections se basan en los métodos **equals()** y **hashCode()** de la clase Object. El API de Collections proporciona dos implementaciones de esta interfaz : **HashSet** y **TreeSet**. Existen pequeñas diferencias entre cada una de estas implementaciones como veremos a continuación:

La clase HashSet

Implementa el conjunto de datos utilizando un tabla hash (HashMap). Los elementos dentro de la interfaz no están ordenados y pueden variar su posición a lo largo del tiempo a medida que se vayan haciendo operaciones de balanceo en la estructura.

Esta clase ofrece un rendimiento similar para las operaciones básicas (añadir/eliminar/buscar..), siempre y cuando la función de hash realice una dispersión correcta de los elementos. La iteración a lo largo de los elementos requiere una cantidad de tiempo proporcional a la suma del tamaño del conjunto más el del HashMap que lo contiene (número de cubos).

Optimización : Es importante no establecer la capacidad inicial demasiado alta (o el factor de carga demasiado bajo) si nos interesa obtener un rendimiento alto al realizar una iteración sobre los elementos.

Importante : Al utilizar una estructura basada en una tabla hash para añadirle objetos es que si queremos añadirle nuestros propios objetos **hemos de sobreescribir el método hashCode()** de los mismos. Ya que en este caso para que dos elementos se consideren iguales, no sólo han de devolver true al utilizar equals sobre los mismos si no que ambos **han de devolver el mismo hashCode()**. Esto es importantísimo ya que sino nos podemos encontrar con cosas tan inesperadas como el siguiente ejemplo :

```
class MiObjeto {
      private int valor = -1;
      public MiObjeto(int valor) {
            this.valor = valor;
      }
      public boolean equals(Object o) {
            if (!(o instanceof MiObjeto)) return false;
            if ( ((MiObjeto)o).valor == this.valor)
                  return true;
            else
                  return false;
      }
}
.........
Set set = new HashSet();
set.add(new MiObjeto(5));
set.add(new MiObjeto(7));
System.out.println(set.contains(new MiObjeto(5));
```

Como vemos hemos creado una clase MiObjeto que encapsula a un entero. Además hemos sobreescribo el método equals() de modo que dos instancias de MiObjeto son iguales si tienen el mismo valor. Sin embargo la salida del fragmento de código que hemos puesto a continuación es false. Esto es debido a que no nos llega sólo con el método equals(). Como al método contains() le hemos pasado una instancia diferente de MiObjeto, esta instancia devolverá un valor de hashCode diferente que la instancia que se encuentra dentro del conjunto y que tiene el valor 5, y por lo tanto el método contains devolverá false. Para solucionar esto tendremos que modificar la clase objeto añadiéndole un método hashCode adecuado :

```
public int hashCode() {
    return valor;
}
```

Con este método hashCode aseguramos que dos elementos con el mismo valor retornarán el mismo código hash y por lo tanto serán iguales. En este caso si volvemos a ejecutar el programa con esa variación la salida ya será true.

Optimización : Hay que tener cuidado al sobreescribir el método hashCode para que distribuya equitativamente los códigos hash de modo que se pueda aumentar el rendimiento.

La clase TreeSet

Esta implementación utiliza un TreeMap para almacenar los objetos. Los objetos se guardan de una manera ordenada, en orden ascendente, según el orden natural de los objetos o según el orden del comparador que le pasemos en el constructor (veremos después los comparadores). Esta clase es útil pues cuando queremos recorrer los elementos de nuestro conjunto de una manera ordenada.

En cuanto a rendimiento, esta clase nos proporciona un coste de log(n) en las operaciones básicas (add/remove/contains).

Importante : Es muy importante que nuestro comparador sea consecuente con el método equals() de los elementos que tengamos dentro del conjunto de datos. Esto es debido a que aunque el funcionamiento de la estructura se basa en el método equals el algoritmo de ordenación se basa en el método **compareTo()** de la interfaz **Comparator**. Hay que tener mucho cuidado con esto ya que dos objetos que sean iguales con respecto al método compareTo() lo serán para el conjunto (y por lo tanto el segundo no se insertará) aunque el método equals() diga lo contrario.

Para terminar con los conjuntos, podemos ver a continuación un ejemplo :

```
import java.util.*;
```

```
public class Ejemplo {
    public static void main(String[] args) {
        Set set = new HashSet();
        set.add("Inés");
        set.add("Ana");
        set.add("Sonia");
        set.add("Laura");
        set.add("Rebeca");
        System.out.println(set);
        Set sortedSet = new TreeSet(set);
        System.out.println(sortedSet);
    }
}
```

Como vemos el ejemplo crea un conjunto con un HashSet y lo imprime, a continuación crea un conjunto ordenado utilizando el conjunto anterior y lo imprime. La salida es la siguiente :

```
[Laura, Inés, Sonia, Rebeca, Ana]
[Ana, Inés, Laura, Rebeca, Sonia]
```

Optimización : Si queremos recorrer un conjunto ordenado, normalmente es más eficiente crear el conjunto utilizando un HashSet y transformarlo en un TreeSet como en el ejemplo anterior para recorrerlo.

Por último, tenemos una clase abstracta **AbstractSet** que implementa algunos de los métodos de la interfaz Set. Adicionalmente sobreescribe el método equals y el método hashCode(). La sálida del método hashCode() para esta clase es la suma de la sálida de los métodos hashCode() de todos sus elementos por lo que no importa en que orden se encuentren los elementos, el conjunto siempre será el mismo independientemente del orden.

La interfaz Map

Esta interfaz es la raiz de una nueva jerarquía dentro del API de Collections. Por definición, la interfaz define una estructura de datos que mapeará claves con valores sin permitir claves duplicadas. Tenemos varios tipos de métodos :

- Modificación

```
Object put(Object clave, Object valor)
void putAll(Map mapa)
void clear()
Object remove(Object clave)
```

- Consulta

```
Object get(Object clave)
boolean containsKey(Object key)
boolean containsValue(Object value)
int size()
boolean isEmpty()
```

- Vista

```
Set keySet()
Collection values()
Set entrySet()
```

Por lo tanto, tenemos métodos para introducir uno o más pares clave-valor (**put/putAll**), para eliminar un elemento con una clave determinada o todos los elementos (**clear/remove**). Tenemos métodos para obtener un valor específico según la clave (**get**), métodos para ver si existe una clave o un valor (**containsKey/containsValue**) y métodos para consultar el tamaño (**size/isEmpty**).

Por último tenemos un método **keySet** que nos devuelve un conjunto de datos (Set) con todas las claves. Nos devuelve un conjunto ya que las claves son únicas. Como los valores de las claves no tienen por que ser diferentes, el método **values** nos devuelve una Collection. Por último el método **entrySet** nos devuelve un conjunto de elementos que implementan la interfaz **Map.Entry**.

Cada objeto en la el conjunto de elementos Map.Entry representa una pareja de clave-valor determinada. Iterando a través de esta colección podemos obtener cualquier clave o valor de cualquier elemento así como modificar los valores utilizando el método de esa interfaz **setValue()**.

Imporante : Si modificamos los valores de esa colección de entradas Map.Entry con otro método que no sea setValue() el comportamiento del Iterator no está determinado y el conjunto de valores pasa a ser incorrecto.

La API Collections proporciona dos implementaciones generales de la interfaz Map : **HashMap** y **TreeMap**. Al igual que con el resto de clases, tendremos que utilizar en cada momento la que más nos convenga ya que no hay una mejor que otra.

La clase HashMap

Esta clase implementa la interfaz Map utilizando una tabla hash. Esta implementación permite el valor null tanto como clave o valor. Al igual que en un HashSet (vimos que utilizaba una HashMap) esta clase no garantiza el orden de los elementos, ni siquiera que no puedan cambiar de orden por motivos de balanceo de carga.

Esta clase garantiza un tiempo constante de acceso para las operaciones básicas (get/put) siempre asumiendo que la función de hash realice una función correcta de los hashCode. En cuanto al rendimiento de la iteración sobre los elementos se cumple lo mismo que vimos previamente en la clase HashSet.

Una instancia de HashMap tiene dos parámetros que afectan a su rendimiento : capacidad inicial y factor de carga. La capacidad es el número de “cubos” de la tabla hash, por lo que la capacidad inicial es simplemente la capacidad que tiene la tabla cuando es creada. El factor de carga es una medida de la cantidad que podemos dejar llenarse a la tabla antes de que su tamaño deba aumentarse. Cuando el número de entradas en la tabla excede el producto del factor de carga y la capacidad actual, se doblará la capacidad llamando al método **rehash**. Como regla general, el factor de carga (75%) ofrece una buena relación entre tiempo de acceso y espacio ocupado. Valores mayores hacen disminuir el gasto de espacio pero incrementan el tiempo de acceso (reflejandose en muchas de las operaciones como get y put).

Optimización : De todo esto se deduce que el número de elementos del mapa y su factor de carga debería tomarse en cuenta al establecer su capacidad inicial para tratar de minimizar las operaciones de rehash. Si la capacidad inicial es mayor que el máximo número de entradas dividio por el factor de carga, la operación de rehash **nunca** ocurrirá.

La clase TreeMap

La implementación de esta clase ordena los elementos por orden ascendente de clave ya sea el orden natural de los objetos o el orden establecido por el comparador que le pasemos en el constructor. Esta clase garantiza un coste de log(n) para las operaciones get, put y contains. Como ocurría en la clase TreeSet el comparador ha de ser consistente con el método equals().

HasMap vs HashTable

Esta comparación es muy similar a la que vimos anteriormente de Vector frente a ArrayList. Al igual que ocurría antes, HashTable existe desde la versión 1.1 de Java y tiene todos sus métodos sincronizados por lo tanto todo lo comentado anteriormente sobre Vector se aplica sobre HashTable, quiere esto decir que su rendimiento es muchísimo menor en comparación

con la clase HashMap. Además, la clase HashTable no soporta claves ni valores nulos. Por último comentar que al igual que ArrayList y el resto de Collections estas clases pueden ser sincronizadas fácilmente utilizando métodos auxiliares.

Veamos a continuación un ejemplo de uso de la interfaz Map.

```
import java.util.*;

public class Ejemplo {
      public static void main(String[] args) {
            Map map = new HashMap();
            map.put("Luis","María");
            map.put("Pedro","Paula");
            map.put("Martín","Inés");
            System.out.println(mapa);
            Map sortedMap = new TreeMap(map);
            System.out.println(sortedMap);
      }
}
```

La salida del programa es la siguiente :

```
{Pedro=Paula, Martín=Inés, Luis=María}
{Luis=María, Martín=Inés, Pedro=Paula}
```

Al igual que ocurría con las listas y conjuntos disponemos de una clase abstracta **AbstractMap** que podremos extender y que sobreescribe el método equals() y hashCode(). En este caso dos AbstractMap serán iguales si tienen el mismo tamaño, contienen las mismas claves y cada clave tiene asociado el mismo valor. Para conseguir esto el método hashCode() devuelve la suma de todos los elementos donde cada elemento es una instancia de la clase Map.Entry. De este modo si se cumplen las condiciones anteriores, no importa el orden de los elementos, los mapas serán iguales.

Optimización : La clase WeakHashMap

Esta clase es una implementación especial de un HashMap donde se almacenarán tan sólo referencias débiles a objetos (WeakReferences). La ventaja de esta aproximación es que si la el objeto clave deja de ser referenciado fuera del HashMap, la entrada asociada a esa clave en el mapa pasa a estar disponible para el recolector de basura automáticamente. Estas estructuras son muy útiles para realizar registros temporales donde las claves pueden perder su utilidad con el paso del tiempo.

Algoritmos y estructuras de datos

Como he mencionado anteriormente, el API de Collections no sólo incluye las interfaces y clases necesarias para crear y utilizar estructuras de datos en java sino que también incluye un conjunto de clases que tienen ya implementadas una serie de algoritmos para utilizar con nuestros conjuntos de datos. A continuación vamos a ver estas clases auxiliares y los algoritmos que nos permiten utilizar, entre los que dedicaré un espacio especial a los algoritmos de ordenación.

Algoritmos de ordenación

Tradicionalmente la ordenación de datos ha sido y es una de las operaciones más habituales sobre estructuras de datos y sobre conjuntos de datos en general. En la versión 1.1 de Java no existía apenas soporte para realizar este tipo de algoritmos, sin embargo con la inclusión del API Collections en Java 1.2 muchos de los tipos de datos de Java pasaron a soportar implícitamente los algoritmos de ordenación al implementar la interfaz **Comparable**.

La interfaz Comparable

Esta interfaz se utiliza cuando los elementos de una de nuestras clases tienen un orden determinado. Dado un conjunto de objetos, esta interfaz nos permitirá ordenar esos objetos en un orden determinado. La interfaz Comparable define un único método :

```
public int compareTo(Object o)
```

Este método compara la instancia actual de la clase que implemente la interfaz Comparable con un objeto pasado como parámetro. Si la instancia actual precede en el orden al objeto pasado como parámetro el método devolverá un número negativo. Si sin embargo, la instancia actual sucede en el orden al objeto pasado como parámetro el valor devuelto será un número positivo y si por último ambos elementos ocupan la misma posición en el orden natural de la clase el valor devuelto será un cero. Es importante denotar esto último, ya que que el valor devuelto sea cero no quiere decir que los dos elementos sean iguales sino que ocupan la misma posición en el orden natural.

Ordenación de cadenas de caracteres

Una operación muy habitual es la ordenación de cadenas de caracteres. La clase String implementa por defecto la interfaz Comparable y si miramos la documentación del método compareTo(), veremos que define un orden lexicográfico. Obviamente este orden no nos sirve

de mucho en el mundo real ya que este orden está basado en el código ASCII de los diferentes caracteres. Para solucionar este problema disponemos de la clase **Collator** que nos permite realizar la comparación de cadenas de caracteres en base a la localización geográfica en la que nos encontremos. Veamos un ejemplo :

```
import java.text.*;
import java.util.*;

public class Ejemplo {
    public static void main(String[] args) {
        Collator collator = Collator.getInstance();
        CollationKey key1 = collator.getCollationKey("Martin");
        CollationKey key2 = collator.getCollationKey("martín");
        CollationKey key3 = collator.getCollationKey("Martín");
        CollationKey key4 = collator.getCollationKey("Martínez");
        CollationKey key5 = collator.getCollationKey("Martiño");

        Set set = new TreeSet();
        set.add(key1);
        set.add(key2);
        set.add(key3);
        set.add(key4);
        set.add(key5);

        Iterator iterator = set.iterator();
        while(iterator.hasNext()) {
            CollationKey key = (CollationKey)iterator.next();
            System.out.print(key.getSourceString() + " ");
        }
    }
}
```

Como vemos el funcionamiento es sencillo. Lo primero que tenemos que hacer es coger una instancia de la clase Collator; llamando al método estático getInstance() obtenemos una instancia asociada al idioma que tengamos activado en nuestro ordenador, en este ejemplo ha sido el español tradicional. A continuación obtenemos una clave con esa instancia de collator para cada una de las cadenas que queremos ordenar. Cada una de esas **CollationKey** implementa la interfaz Comparable y realiza una ordenación basándose en el Collator con la que fue creada, es decir el español en este caso. Por último si las queremos imprimir en pantalla tendremos que obtener las cadenas originales llamando al método getSourceString().

En este caso la salida fue la siguiente :

```
Martin martín Martín Martínez Martiño
```

Si no hubiésemos utilizado la clase Collator y hubiésemos agregado directamente las cadenas a nuestro conjunto la salida sería la siguiente :

Martin Martiño Martín Martínez martín

que claramente es incorrecta.

La interfaz Comparator

Cuando una clase por cualquier motivo no está diseñada para poder implementar la interfaz Comparable o cuando no nos gusta la interfaz Comparable que tenga definida una clase por defecto siempre podremos pasarle a los diferentes algoritmos de ordenación otra interfaz que previamente hayamos creado. Esta interfaz se denomina **Comparator** y define los siguientes métodos :

```
int compare(Object elemento1, Object elemento2)
boolean equals(Object o)
```

El valor de retorno del método compare() se obtiene del mismo modo que vimos anteriormente para compareTo(), sólo que en vez de comparar la instancia actual con el elemento pasado como parámetro compararemos los dos elementos pasados. El método equals() en este caso se utiliza para ver cuando dos implementaciones de Comparator son iguales.

Veamos como podríamos crear un comparador de cadenas de carácteres que ignorase las mayúsculas y minúsculas (para por ejemplo no utilizar la clase Collator) :

```
public Class MiComparador implements Comparator {
    public int compareTo(Object elemento1, Object elemento2) {
        String cadena 1 = ((String)elemento1).toLowerCase();
        String cadena2 = ((String)elemento2).toLowerCase();
        return cadena1.compareTo(cadena2);
    }
}
```

Como curiosidad decir que la clase **Collections** (que es donde tenemos métodos auxiliares de utilidad) tiene un método llamado **reverserOrder()** que nos devuelve un comparador que ordenará cualquier colección de elementos en el orden inverso al que se haría normalmente.

Por último comentar que en el API Collections existen dos interfaces definidas que permiten mantener elementos de una manera ordenada. Estas interfaces son **SortedSet** y **SortedMap** y sus implementaciones concretas son **TreeSet** y **TreeMap** que ya las hemos visto anteriormente por lo que no hablaré más de ellas.

La clase Collections

Como ya he comentado anteriormente dentro de la clase Collections vamos a tener un conjunto de métodos de utilidad que nos permitirán realizar operaciones sobre nuestras estructuras de datos. El equipo de desarrollo del API de Collections decidió separar todos estos métodos de las implementaciones concretas proporcionandolos en esta clase que actúa como un wrapper (también conocido como decorador). Vamos a ver a continuación algunas de las ayudas que nos aporta.

Colecciones de datos inmutables

Muchas veces nos encontraremos con que después de haber creado una estructura de datos nos gustaría proteger su acceso para evitar inconsistencias. La clase Collections nos brinda esta posibilidad con la introducción de seis métodos factoría :

```
Collection unmodifiableCollection(Collection collection)
List unmodifiableList(List list)
Map unmodifiableMap(Map map)
Set unmodifiableset(Set set)
SortedMap unmodifiableSortedMap(SortedMap map)
SortedSet unmodifiableSortedSet(SortedSet set)
```

Veamos un ejemplo :

```
.....
Set set = new HashSet();
set.add(“Rosa”);
set.add(“Clara”);
set.add(“Nuria”);
set = Collections.unmodifiableSet(set);
set.add(“Laura”);
```

Importante : Es importante reemplazar la referencia original a nuestra colección de datos como en el ejemplo ya que de otro modo todavía se podría utilizar la referencia original para modificar los datos. En este ejemplo la última llamada a set(), originará una excepción **UnsupportedOperationException** en tiempo de ejecución.

Colecciones de datos seguras

Como vimos anteriormente la gran diferencia de las nuevas estructuras de datos era que no eran thread-safe, es decir que no eran seguras en un entorno multihilo. Muchas veces necesitaremos garantizar esa seguridad en nuestras aplicaciones para evitar inconsistencias en los datos. Por este motivo la clase Collections nos proporciona otros seis métodos factoría para obtener colecciones de datos sincronizadas :

```
Collection synchronizedCollection(Collection collection)
List synchronizedList(List list)
Map synchronizedMap(Map map)
Set synchronizedSet(Set set)
SortedMap synchronizedSortedMap(SortedMap map)
SortedSet synchronizedSortedSet(SortedSet set)
```

Nuevamente debemos asegurarnos de que sobreescribimos o eliminamos la referencia a la colección de datos no sincronizada. Ejemplo :

```
Map map = Collections.synchronizeMap(new HashMap());
```

Colecciones inmutables inicializadas automáticamente

La clase Collections nos proporciona varios métodos para crear colecciones de datos con un único dato y además inmutables imitando al comportamiento de clases como Integer o String :

```
Set singleton(Object objeto)
Map singletonMap(Object clave, Object valor)
List singletonList(Object objeto)
```

Podemos ir un poco más allá aún y crear una colección de datos inmutable que contenga el mismo objeto n veces. Para ello utilizaremos el siguiente método :

```
List nCopies(int n, Object obj)
```

Esto que en un inicio no parece demasiado útil nos puede permitir crear e inicializar rápidamente colecciones de datos modificables simplemente pasándole nuestra colección inmutable a la nueva. Acaremos esto con un ejemplo :

```
List listaInmutable = Collections.nCopies(20, new Integer(10));
List listaMutable = new ArrayList(listaInmutable);
```

Con estas dos lineas de código hemos creado una lista que contiene veinte números enteros en su interior inicializados con el valor diez.

Colecciones de datos vacías

La clase Collections nos proporciona varias constantes que representan colecciones de datos vacías :

```
static List Collections.EMPTY_LIST
static Map Collections.EMPTY_MAP
static Set Collections.EMPTY_SET
```

Algoritmos de ordenación

Como hemos visto tenemos disponibles dos interfaces que nos definen conjuntos y mapas de datos ordenados. Sin embargo, no existe ninguna interfaz que define una lista ordenada. Para salvar éste y otros problemas, la clase Collections proporciona dos métodos para listas de datos :

```
void sort(List list)
void sort(List list, Comparator c)
```

El primer método lo utilizaremos cuando los elementos de la lista implementan la interfaz Comparable vista anteriormente y el segundo lo utilizaremos cuando querramos utilizar nuestro propio comparador o cuando no nos guste el funcionamiento del comparador por defecto de los elementos de nuestra lista.

Ambas versiones garantizan un coste de O(nlog(n)) y puede acercarse a un rendimiento lineal cuando las los elementos se encuentran cerca de su orden natural. El algoritmo utilizado es una pequeña variaciónd el algoritmo de mergesort y la operación que realiza es destructiva, es decir, no podremos recuperar el orden original si no hemos guardado la lista previamente.

Algoritmos de búsqueda

Cuando utilizamos el método contains() sobre una lista para obtener un elemento, este método asume que la lista no está ordenada y irá buscandolo secuencialmente. Si previamente hemos ordenado nuestra lista podemos obtener un gran aumento de rendimiento utilizando una búsqueda binaria. Tenemos un par de métodos en la clase Collections para realizar esto :

```
int binarySearch(List lista, Object clave)
int binarySearch(List lista, Object clave, Comparator comparador)
```

Como en la ordenación, utilizaremos una versión o la otra según más nos convenga. Si la lista no se encuentra ordenada los resultados son imprevisibles. Por otra parte, si la lista contiene varias veces el mismo elemento no tendremos ninguna garantía de cual es el que va a encontrar primero.

Manipulación de elementos

Por último la clase Collections nos proporciona varios métodos que nos permiten modificar los elementos de nuestras estructuras de datos, como por ejemplo :

```
void fill(List lista, Object elemento)
void copy(List origen, List destino)
void reverse(List lista)
void shuffle(List lista)
void shuffle(List lista, Random random)
```

Como se puede ver, podemos rellenar una lista con elementos (**fill**), copiar los elementos de una lista en otra diferente (**copy**), invertir el orden de los elementos de una lista (**reverse**) o permutar las posiciones de los elementos de una lista (**shuffle**).

La clase Arrays

No voy a entrar en detalles sobre esta clase ya que es muy parecida a la clase Collections. Básicamente esta clase nos proporciona algoritmos de ordenación, búsqueda, rellenado, etc. pero en vez de estar destinados a estructuras de datos están destinados a arrays de tipos primitivos de datos. De este modo y utilizando esta clase podremos realizar búsquedas binarias o ordenaciones en los elementos de nuestros arrays ya sean números enteros, bytes, doubles, etc.

El futuro del API Collections, Generics

Desde poco después de la aparición del API de Collections, muchos fueron los programadores que se dieron cuenta de que la abstracción y simplicidad con la que las clases de Collections tratan a los elementos que contienen más de una vez son más un engorro que una ayuda. Como se ha podido ver en este artículo, todas las clases del API de Collections tratan de una manera global a los objetos, es decir, utilizan siempre la clase **Object** para representarlos.

Esta abstracción tiene la ventaja de que podemos introducir en una estructura de datos objetos de diferentes tipos y de una manera transparente, cosa que es de gran ayuda al programador ya sea a la hora de crear una aplicación o cuando sea necesario mantener la aplicación creada por otra persona. Sin embargo, muchas veces ocurre que sabemos de antemano el tipo de los elementos que contiene una determinada estructura de datos y sin embargo nos veremos obligados a estar realizando continuos cast explícitos para obtener datos de los que sabemos con seguridad su tipo. Estos cast, no sólo son molestos a la hora de programar sino que también hacen que disminuya el rendimiento de la aplicación ya que no deja de ser una operación como cualquier otra con su coste computacional. Veamos un ejemplo :

```
ArrayList listaEnteros = new ArrayList();
listaEnteros.add(new Integer(5));
listaEnteros.add(new Integer(7));
Integer primero = (Integer)listaEnteros.get(0);
```

Como vemos en este código, nos vemos obligados a realizar un cast cuando sabemos con certeza que listaEnteros contiene números enteros en su interior. Una posible solución a esto es crear nuestras propias colecciones de datos que encapsulen todos estos cast, pero esta solución tan sólo evitará que tengamos que escribir en nuestras aplicaciones los cast sin solucionar el problema del rendimiento.

Debido en parte a estos problemas (y a otros), desde la aparición de Java 2, toda la comunidad de desarrollo ha estado pidiendo a gritos la inclusión de lo que se conoce como Generics, y que prácticamente con toda seguridad verá la luz con la versión 1.5 del J2SE.

El principal beneficio de añadir generalidad al lenguaje de programación Java (Generics es algo que afectará a todo el lenguaje, no sólo a las Collections) yace en el establecimiento de manera explícita de los parámetros y en que las operaciones de cast pasarán a ser implícitas. Esto es algo crucial para utilizar librerías como Collections de una manera flexible y segura.

Podemos ver a continuación un ejemplo de como definiríamos con la nueva API nuestro vector anterior :

```
ArrayList<Integer> listaEnteros = new ArrayList<Integer>();
listaEnteros.add(new Integer(5));
listaEnteros.add(new Integer(7));
Integer primero = listaEnteros.get(0);
```

Vuelvo a aclara que el añadir el API de Generics no es algo que afecte sólo a las Collections sino que afectará a todo el lenguaje (sin introducir incompatibilidades por supuesto), por lo que es una característica que ha costado mucho tiempo introducir. Actúalmente se puede bajar de la web de SUN una versión del compilador con soporte para la especificación de este API. El objetivo final de este API será facilitar más todavía la vida del programador, introduciendo esta generalidad de la que he hablado pero intentando ofuscar el lenguaje lo menos posible (cualquiera que haya trabajado con la librería de templates de C++ sabrá bien de lo que hablo).

Conclusión

En este pequeño artículo he tratado de mostrar todo lo que nos podemos encontrar dentro del API de Collections. En ningún modo se trata de una guía completa sino que el lector ha de adentrarse en la documentación de la API para ver que existen todavía más métodos, clases e interfaces que por espacio no he tratado. Como hemos visto Java nos simplifica el gran parte el trabajo con estructuras de datos, no sólo proporcionandonos algoritmos y clases que nos

permiten trabajar con las mismas sino ofreciéndonos también un método estandar, elegante y sencillo de trabajar (cualquiera que haya programado en lenguajes como C++ entenderá muy bien este punto). De todos modos como hemos visto el API de Collections no se encuentra ni mucho menos exento de problemas y toda la comunidad de desarrollo está esperando ansiadamente la llegada de los tipos genéricos de datos en el JDK 1.5 para que estos flecos que quedan todavía sin arreglar se solucionen definitivamente.

Bibliografía

[1] *Java Collections Framework, http://www.ibm.com/developerWorks/Java*

[2] *Java 2 Standard Edition API Documentation, http://java.sun.com/products/jdk/1.4/docs/*

[3] *Java Collections Tutorial, http://java.sun.com/products/jdk/1.2/docs/guide/collections/*

[4] *Design Patterns, Erig Gamma, Richard Helm, Ralph Johnson y John Vlissides, http://hillside.net/patterns/DPBook/DPBook.html*

[5] *Adding Generics to the Java Programming Language : Participant Draft Specification, April 2001*