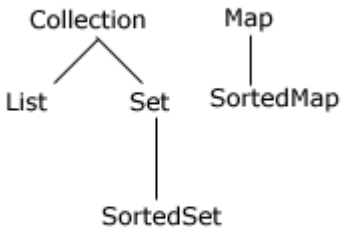
Java Collection Framework

por Lucas Videla

El Framework de colecciones de Java provee un conjunto de interfaces y clases para ordenar y manipular grupos de datos como una unidad simple: una *colección*.
Este Framework provee una interfaz para programar común a la mayoría de las abstracciones, sin apabullar al programador con muchas interfaces o procedimientos diferentes.
Está compuesto de un grupo de interfaces para trabajar con distintos grupos de clases, las cuales se asocian con alguna interfaz específica.
Existen dos interfaces principales en el Framework, las cuales no están vinculadas en una jerarquía: Las interfaces *Collection* y *Map*. En líneas generales, las clases que implementan la interfaz *Map* sirven para proveer acceso a los objetos almacenados, mediante claves. Pero ya veremos eso más adelante.

Aquí presentamos la jerarquía principal del Framework:



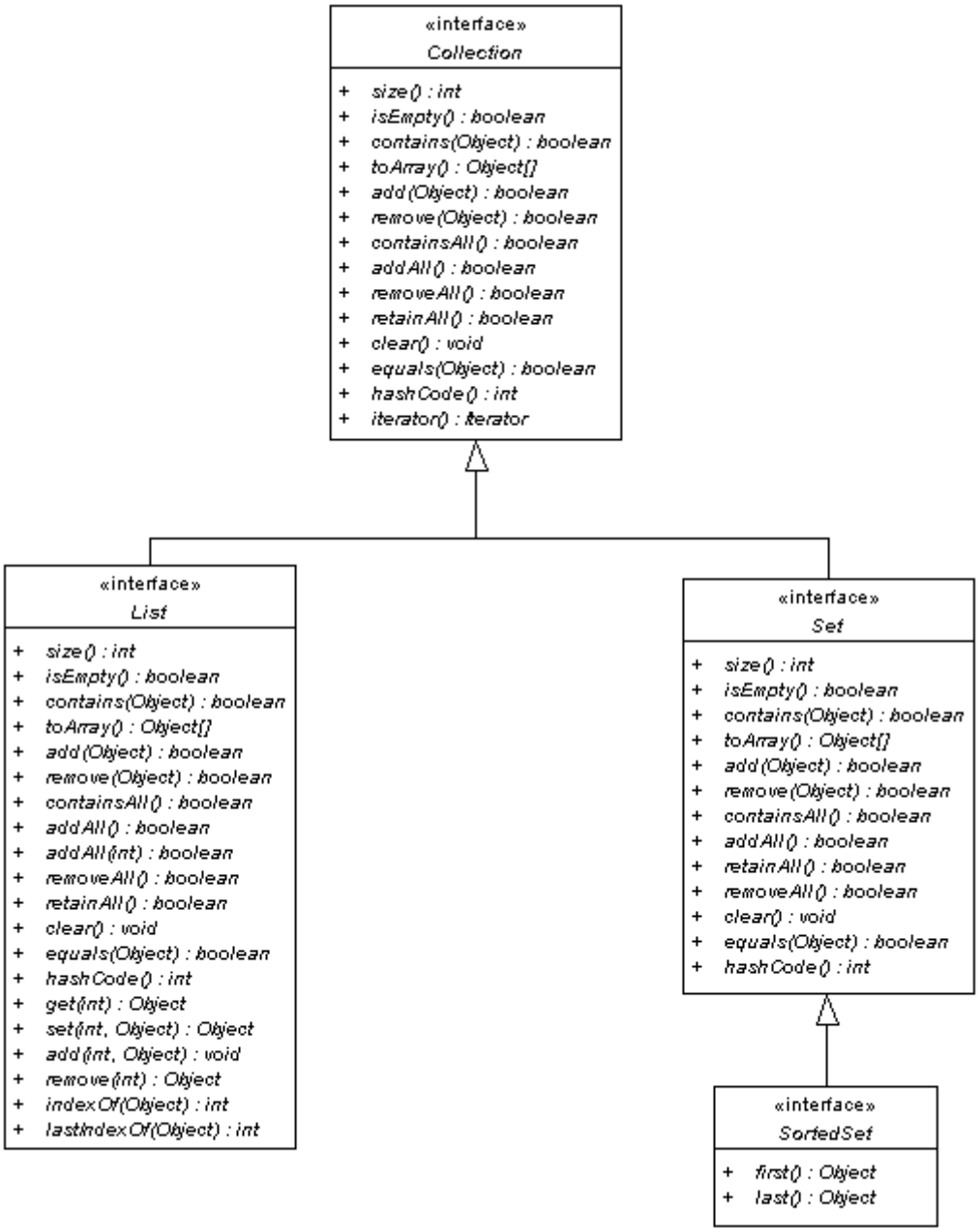
Debemos tener en cuenta ciertas características al momento de seleccionar qué elemento nos sirve para nuestro propósito:

- Cuando se habla de *Collection*, estamos dando por entendido que admitimos cualquier clase de elementos, incluyendo elementos duplicados.
- Cuando nos especificamos, y pasamos a hablar de *Set*, eliminamos la posibilidad de elementos duplicados.
- Si hablamos de *List*, además de admitir los duplicados (por la interfaz *Collection*), obtenemos la funcionalidad de tener los objetos indexados posicionalmente.
- *Map* es una interfaz aparte, que no extiende ninguna de las otras. Pero básicamente sirve para "mapear" datos por medio de una clave.

Hasta ahora hemos hablado de las interfaces del Framework, pero no hemos dicho nada sobre las clases concretas que debemos utilizar para nuestros programas. Estas son:

Clase concreta	**Interfaz que implementa**	**Anteriormente se utilizaba**
HashSet	Set	
TreeSet	Set	
LinkedList	List	Vector
ArrayList	List	Stack
HashMap	Map	Hashtable
TreeMap	Map	

Veamos la jerarquía de interfaces, lo cual nos dará una idea de los métodos que implementarán las clases concretas:



La documentación correspondiente la pueden conseguir siguiendo los links desde esta página:
http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collection.html

En líneas generales, ningún método de estas interfaces presentaría dificultades para ser entendido, pero no debemos dejar pasar el método *iterator(),* que devuelve un objeto del tipo *Iterator*, el cual vamos a utilizar mucho.

Colecciones tipadas

Se denominan colecciones tipadas a aquellas donde se especifica el tipo de dato que contienen, esto es posible hacerlo utilizando la notación <tipo de dato>. En caso de no especificar un tipo de datos, la colección asume Object como tipo de datos, y será necesario castear el contenido de la colección al tipo indicado al momento de utilizar un objeto contenido por la misma:

Colecciones tipadas

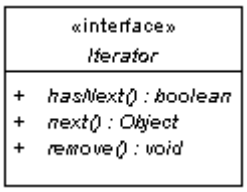
```
Collection<String> nombres = new LinkedList<String>();
Collection<Persona> personas = new LinkedList<Persona>();
```

Colecciones no tipadas

```
Collection nombres = new LinkedList();
Collection personas = new LinkedList();
```

Iteradores

Un iterador es un objeto que se desplaza a lo largo de una colección, devolviendo uno a uno los elementos de la misma. Veamos cómo es un iterador:



Posee tres métodos, los cuales nos garantizan que podremos recorrer la colección ¿Cómo lo vamos a hacer? Muy bien, aquí está un ejemplo:

```
Collection<String> lista = new ArrayList<String>();

Iterator<String> iterator = lista.iterator();
while(iterator.hasNext()){
	String cadaElemento = iterator.next();
	System.out.println(cadaElemento.length());
}
```

Esa es la forma más sencilla de utilizar un iterador. Tenemos otra, que es levemente más complicada:

```
for(Iterator<String> iterator = lista.iterator();iterator.hasNext();){
	String cadaElemento = iterator.next();
	System.out.println(cadaElemento.length());
}
```

No es particularmente más complicada por el uso, sino por la notación confusa. Recomendamos ampliamente la primera forma de utilizar el iterador.

For each:

Otro modo de iterar colecciones es utilizando un for-each, la ventaja de este tipo de estructura que además de colecciones nos permite iterar arrays

```
for(String cadaElemento : lista){
	System.out.println(cadaElemento.length());
}
```

El ”cómo” de los Set

Ya hemos mencionado que los *Set* no admiten duplicados, por lo que rechazan un objeto que ya existe al intentar agregarlo a la colección. Pero... ¿cómo hace para saber cuándo dos objetos son iguales? Recordemos que si debe guiarse por el operador de igualdad (==), podría fallar, ya que éste compara por la referencia respectiva de los objetos que compara; sabemos bien que dos objetos de referencias diferentes pueden ser el mismo.

Ahora bien, compara por medio del método *equals()* que DEBE ser reescrito para cada objeto que pretenda agregarse a un Set. Por lo tanto, aquí vemos otro método de Object que es necesario implementar. No tardaremos en encontrarnos con otros.

Por lo tanto, al recibir un objeto lo compara, por el método *equals()* contra todos los objetos existentes en la colección. Si no existe, lo agrega. Si existe, retorna falso y no agrega nada.

Ahora, como ejercicio, modelemos el comportamiento de la Unión, Intersección, Diferencia Simétrica y Diferencia entre conjuntos (Set).

Los Map, y su uso particular

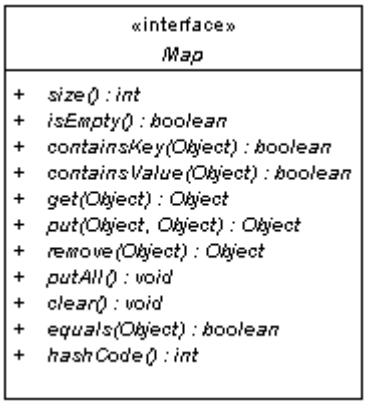
Hemos dicho que los *Map* admiten objetos, "mapeándolos" sobre una clave. Este comportamiento se demuestra por el método correspondiente *put(key, object)*, donde ambos parámetros son del tipo Object. Por lo tanto, podemos hacer una suerte de "vector asociativo", en base a un objeto utilizado como clave, y otro objeto utilizado como valor. Veamos cómo utilizarlo:

```
Map<String, Integer> tm = new TreeMap<String, Integer>();
tm.put("uno",new Integer(1));
tm.put("dos",new Integer(2));
tm.put("tres",new Integer(3));

System.out.println(tm.get("dos"));
```

Vemos cómo podemos agregar y acceder en forma asociativa a cada elemento del *Map*.

Complementamos con el diagrama de clase de la interfaz *Map*:



Los Sorted[Map/Set]

Hemos llegado a este punto sin discutir las colecciones que permiten añadir elementos en orden. Éstas, basan su comportamiento en la comparación de los objetos, a medida que van siendo agregados ¿Cómo saben cuál es mayor y cuál es menor? Fácil: utilizando la interfaz *Comparable*, o un objeto *Comparator*. Específicamente, cada elemento agregado a un Sorted[Map/Set] debe implementar esa interfaz, o al construir un Sorted, se debe especificar un objeto Comparator que se utilizará para ordenar los elementos que entren a la colección.
Un ejemplo:

```
SortedSet<String> ss = new TreeSet<String>();
ss.add("a");
ss.add("d");
ss.add("c");
ss.add("b");
System.out.println(ss);
```

La salida por pantalla estará ordenada, a pesar de que la entrada no lo estaba.

Los Constructores

En líneas generales, los constructores de las clases que implementan estas interfaces, admiten una forma que reciba como parámetro un objeto que responda a la interfaz *Collection*, lo cual nos ahorra un gran trabajo.
Supongamos que tenemos una lista con objetos repetidos, y deseamos sacar aquellos que efectivamente son repetidos. Simplemente debemos instanciar un Set basándonos en esa lista, y listo:

```
Collection<String> hs = new HashSet<String>(miListaConDuplicados);
```

Podremos comprobar que la lista, que sí tenía duplicados, al ser pasada a un Set, ya no los tiene ¿por qué? (ver definición de Set).

ArrayList contra LinkedList

Como sabemos, no existen cosas con nombres diferentes que hagan los mismo (o al menos no debería suceder). Por ello tenemos estas dos clases. Ambas representan una lista de objetos (admiten nulos y repetidos), pero tienen una diferencia:

Los ArrayList se construyen sobre un vector, por lo que proveen acceso aleatorio, mientras que los LinkedList se construyen enlazando objetos, por lo que proveen acceso secuencial, aunque tienen facilidades para insertar elementos delante, detrás y en medio de la lista

Esa diferencia, hace que en ciertos casos convenga el uso de una, y en otros, el uso de la otra.

Implementando una pila

Para finalizar, mostramos el código fuente de una Pila, implementada sobre una LinkedList. Si bien faltan muchos métodos, es una implementación básica:

```
public class Pila {
	private LinkedList<String> ll;
	public Pila(){
		this.ll = new LinkedList<String>();
	}
	public void push(String o){
		ll.addFirst(o);
	}
	public String pop(){
		return ll.removeFirst();
	}
	public String peek(){
		return ll.getFirst();
	}
	public String toString(){
		return ll.toString();
	}
}
```

Equals y Hashcode

Son dos métodos definidos en la clase Object, la firma de ambos es

- `public boolean equals(Object obj)`
- `public int hashCode()`

Estos métodos son muy importantes cuando se comparan clases o cuando se agregan clases a colecciones.

public boolean equals(Object obj)

Este método verifica si un objeto pasado como argumento es igual (según el negocio) al cual se le invoca el método. La implementación por defecto en la clase Object simplemente verifica si las referencias a ambos objetos coinciden (x == y)

Para hacer una comparación mas profunda las clases deberían sobrescribir estos métodos.

Es necesario sobrescribir el método hashcode cuando se sobrescribe este método, para poder mantener el contrato entre ambos, el cual indica que si dos objetos son iguales, también deben serlo sus hashcodes.

public int hashCode()

Este método devuelve un valor entero de hashcode que es utilizado en las colecciones basadas en métodos de hash: hashMap. Este método debe ser sobrescrito en las clases donde se sobrescriba el método equals.

Siempre el método hashcode de un objeto debe retornar el mismo valor.
Si dos objetos son iguales según equals deben retornar el mismo hashcode
Si dos objetos no son iguales según equals no necesariamente deben retornar diferente hashcode.

```
public class Test {
	private int numero;

	public int getNumero() {
		return this.numero;
	}

	public boolean equals(Object obj) {
		// verifico si es la misma referencia.
		if (this == obj)
			return true;
		// Verifico si el objeto recibido es null
		// o si son de clases distintas.
		if ((obj == null) || (obj.getClass() != this.getClass()))
			return false;
		// A esta altura estoy seguro que son de la misma clase.
		Test test = (Test) obj;
		return numero == test.getNumero();
	}

	public int hashCode() {
		return this.numero;
	}
}
```

Lectura adicional:

http://java.sun.com/docs/books/tutorial/collections/index.html

http://www.allapplabs.com/java/java_collection_framework.htm

http://java.sun.com/j2se/1.5.0/docs/guide/collections/