INSTITUTO POLITÉCNICO NACIONAL

UNIDAD PROFESIONAL INTERDISCIPLINARIA DE INGENIERÍA Y CIENCIAS SOCIALES Y ADMINISTRATIVAS

Material para el curso de Certificación en Java 6 SE

Contenido

© Lic. Fernando Luque M.

Certificación

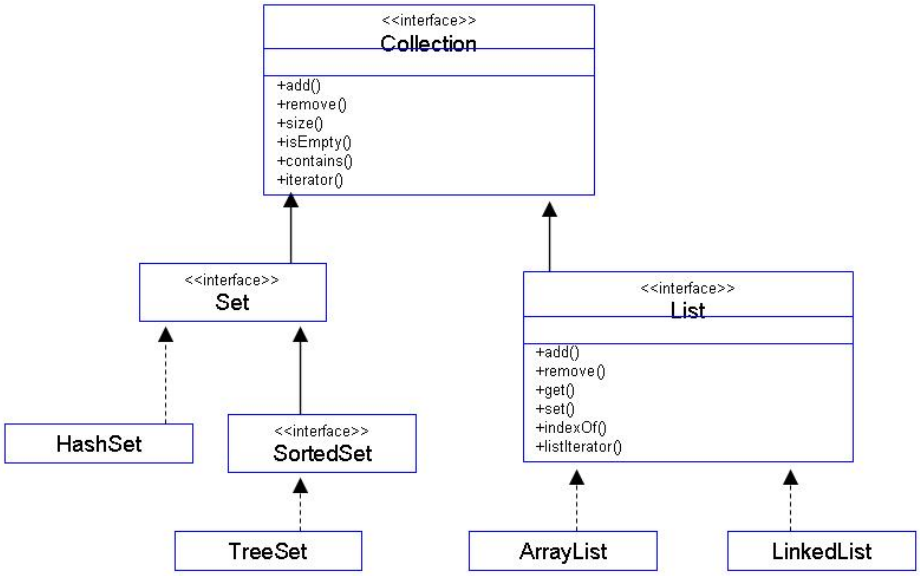
Collections

Son un único objeto que maneja a un grupo de otros objetos que descienden de un tipo particular, llamados elementos. Tenemos por ejemplo las siguientes interfaces:

- Collection. Cada implementación determina si existe un orden especifico, y si se permiten o no los duplicados.
- Set. Una colección desordenada que no permite duplicados.
- List. Una colección ordenada que si permite duplicados.

Antes de la versión 5.0 las colecciones eran de objetos tipo Object, lo que implicaba hacer casting después de recuperar cada elemento. Sin embargo, gracias a las nuevas características de colecciones genéricas, podemos especificar los tipos de objeto que se almacenaran, evitando el casting.

Java ofrece todo un API para manejarlas:



El HashSet es un ejemplo de una clase que provee la implementación de la interfase Set. Las clases que implementen SortedSet imponen un ordenamiento total en sus elementos.

© Lic. Fernando Luque M.

Implementaciones

Existen varias implementaciones de propósito general de las 4 interfaces primarias (Set, List, Map, y Deque). Las más usuales son:

	Tabla de Hash	Arreglo ajustable de tamaño	Árbol balanceado	Lista ligada	Tabla de Hash, y Lista ligada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Un ejemplo de Set:

```
import java.util.*;
public class EjemploSet {
        public static void main(String[] a) {
                Set set = new HashSet();
                set.add("uno");
                set.add("segundo");
                set.add("3");
                set.add(new Integer(4) );
                set.add(new Float(3.4F) );
                set.add("segundo");         // duplicado, no se agrega, devuelve
falso
                set.add(new Integer(4) );           // duplicado, no se agrega
                System.out.println(set);
        }       // main
}
```

El orden en que se muestran los elementos no es necesariamente el orden en que se agregaron

© Lic. Fernando Luque M.

Un ejemplo de List:

```
import java.util.*;
public class EjemploLista {
        public static void main(String[] a) {
                List list = new ArrayList();
                list.add("uno");
                list.add("segundo");
                list.add("3ero");
                list.add(new Integer(4) );
                list.add(new Float(5.0F) );
                list.add("segundo");        // duplicado, sí se agrega, devuelve true
                list.add(new Integer(4) );          // duplicado, si se agrega
                System.out.println(list);
        }       // main
}
```

Este programa escribe:

[uno, segundo, 3ero, 4, 5.0, segundo, 4]

ya que se respeta el orden de inserción (estas clases sobrescriben el método *toString* agregando los corchetes y separando con comas).

Algunas tienen comportamiento hibrido, como *LinkedHashMap*, que implementa algoritmos de hash para que las búsquedas sean rápidas, y también mantiene internamente una lista doble para poder regresar objetos con un iterador en orden significativo.

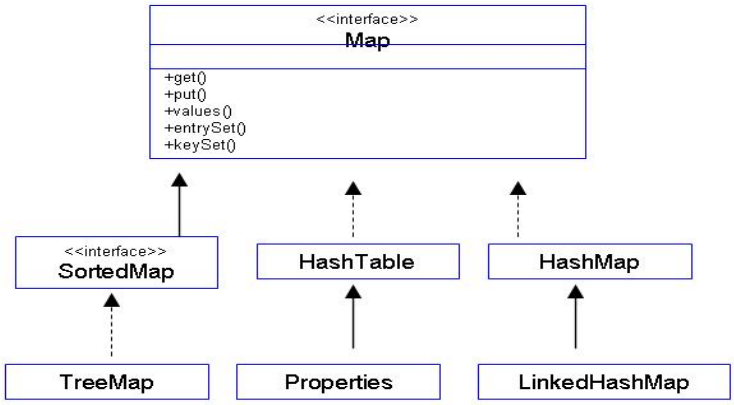
La interfase Map

Un objeto Map describe precisamente una relación o mapeo de llaves a valores, por eso a veces son llamados "arreglos asociativos". No se permiten llaves duplicadas, y cada llave solo puede mapear a un valor. Tenemos 3 métodos para manejarlos:

- entrySet, devuelve un Set que contiene todos los pares llave-valor
- keySet, devuelve un Set que contiene todas los llaves
- values, devuelve un Collection que contiene todos los valores del mapa

No hereda de Collection porque representa algo más complejo que una simple colección. El orden presentado por los iteradores de cada clase es específico de esa clase.

© Lic. Fernando Luque M.



Un ejemplo de Map:

```
import java.util.*;
public class EjemploMap {
        public static void main(String[] a) {
                Map map = new HashMap();
                map.put("uno", "primero");
                map.put("segundo", new Integer(2) );
                map.put("tres", "3ero");
                map.put("tres", "III");         // sobreescribe el valor de la llave "tres"
                Set set1, set2;
                set1 = map.keySet();         // todas las llaves
                Collection col = map.values();      // valñores
                set2 = map.entrySet();               //
                System.out.println(set1 + "\n" + col + "\n" + set2);
        }       // main
}
```


© Lic. Fernando Luque M.

Este programa escribe:

[tres, segundo, uno]

[III, 2, primero]

[tres=III, segundo =2, uno=primero]

Clases de Collection heredadas

Las clases de las versiones 1.0 y 1.1 tienen la misma interfase pero los cambios en su funcionalidad son:

- La clase *Vector* implementa la interfase *List*
- La clase *Stack* hereda de *Vector* y tiene las operaciones típicas: push, pop, y peek
- La clase *HashTable* implementa la interfase *Map*
- La clase *Properties* hereda de *HashTable* y solo usa *Strings* como llaves y valores
- Todas ellas tienen el método elements() que devuelve un objeto *Enumeration*, que es una interfase similar (pero incompatible) a *Iterator*

Ordenando las colecciones

La primera de estas 2 interfases nos sirve para impartir un ordenamiento natural, la segunda se usa para especificar la relación de orden, permitiendo alterar el ordenamiento natural.

Comparable

Cuando declaramos una clase, la JVM no tiene manera de saber el ordenamiento que pretendemos para los objetos de esa clase. Implementando esta interfase podemos proveer orden a los objetos de cualquier clase. Por ejemplo las clases envolventes, *Date* y *String* la implementan, cuyos criterios de implementación son respectivamente numerico, cronologico, y alfabetico. Si pasamos una List al método estático sort () de una *Collection*, el orden resultante dependerá del tipo de elementos de la lista.

En este ejemplo el método compareTo () permite que los estudiantes sean ordenados en base a su promedio:

© Lic. Fernando Luque M.

```
public class Estudiante implements Comparable {
        String nom, ape;
        int id =0;
        double promedio =0.0;       // ojo, no es private
        public Estudiante(String nom, String ape, int id, double promedio) {
                if (nom ==null || ape ==null || id ==0 || promedio == 0.0)
                        throw new IllegalArgumentException();
                this.nom = nom;
                this.ape = ape;
                this.id =id;
                this.promedio = promedio;
        } // cons
        public int compareTo(Object o) {
                double f = promedio -( (Estudiante) o). promedio;
                if (f == 0.0)
                        return 0;       // significa iguales
                else if (f < 0.0)
                        return -1;      // significa menor que, o antes de
                else return 1;
        }
}       // clase
```

Este programa crea 4 objetos *Estudiante* y los imprime. Ya que los agrega a un TreeSet, que esta ordenado, los sortea:

```
import java.util.*;
public class EjemploComparable {
        public static void main(String[] a) {
                TreeSet s = new TreeSet();
                s.add(new Estudiante("Miguel", "Hdez", 101, 4.0) );
                s.add(new Estudiante("Juan", "Lopez", 102, 2.8) );
                s.add(new Estudiante("Jaime", "Perez", 103, 3.6) );
                s.add(new Estudiante("Kelly", "Montiel", 104, 2.3) );
                Object[] arreglo = s.toArray();
                Estudiante es;
```


© Lic. Fernando Luque M.

```
            for (Object obj: arreglo) {
                es =( Estudiante) obj;
                System.out.printf("Nombre = %s %s Id= %d Promedio = %.1f\n", es.getNom(), es.getApe(), es.getId(), es.getPromedio()  );
            }
        }       // main
}
```

Al ir ordenando los elementos, el *TreeSet* se fija si los objetos tienen su orden natural, y en ese caso, usa el método CompareTo para compararlos.

Algunas colecciones como la anterior están ordenadas. La implementación *TreeSet* necesita saber como ordenar los elementos. Como ya vimos, si estos tienen un orden natural, *TreeSet* usa ese orden. En caso contrario, tenemos que especificarlo. Por ejemplo, el *TreeSet* tiene un constructor que recibe un parámetro *Comparator*, que crea un nuevo tree set vacío, ordenado de acuerdo al comparador especificado.

Comparator

© Lic. Fernando Luque M.

Referencias

“Java Básico”
Capacitación Integral Y Servicios Avanzados de Cómputo

“Mastering Java 2.0”
Sun Microsystems

© Lic. Fernando Luque M.