Colecciones

Antonio Espín Herranz

Contenidos

- List, Map, Set
- Collections
- Arrays
- Enumeraciones

Enumeraciones

- Permite definir un conjunto de posibles valores o estados, que luego podremos utilizar donde queramos.
- Se definen con la palabra reservada **enum**
- **También nos permiten agrupar ctes. De la misma temática, mejor que tener las ctes separadas.**

```
public enum Color { Red, White, Blue };
Color miColor = Color.Red;

System.out.println("El color es:" + miColor);
```

Colecciones de datos

- Están en el paquete java.util.
- Ofrecen una manera más completa y orientada a objetos para almacenar conjuntos de datos de tipo similar.
- Las Colecciones tienen su propia asignación de memoria y posibilidad de una nueva asignación para ampliarlas.
- Tienen interfaces de método para su iteración y recorrido.

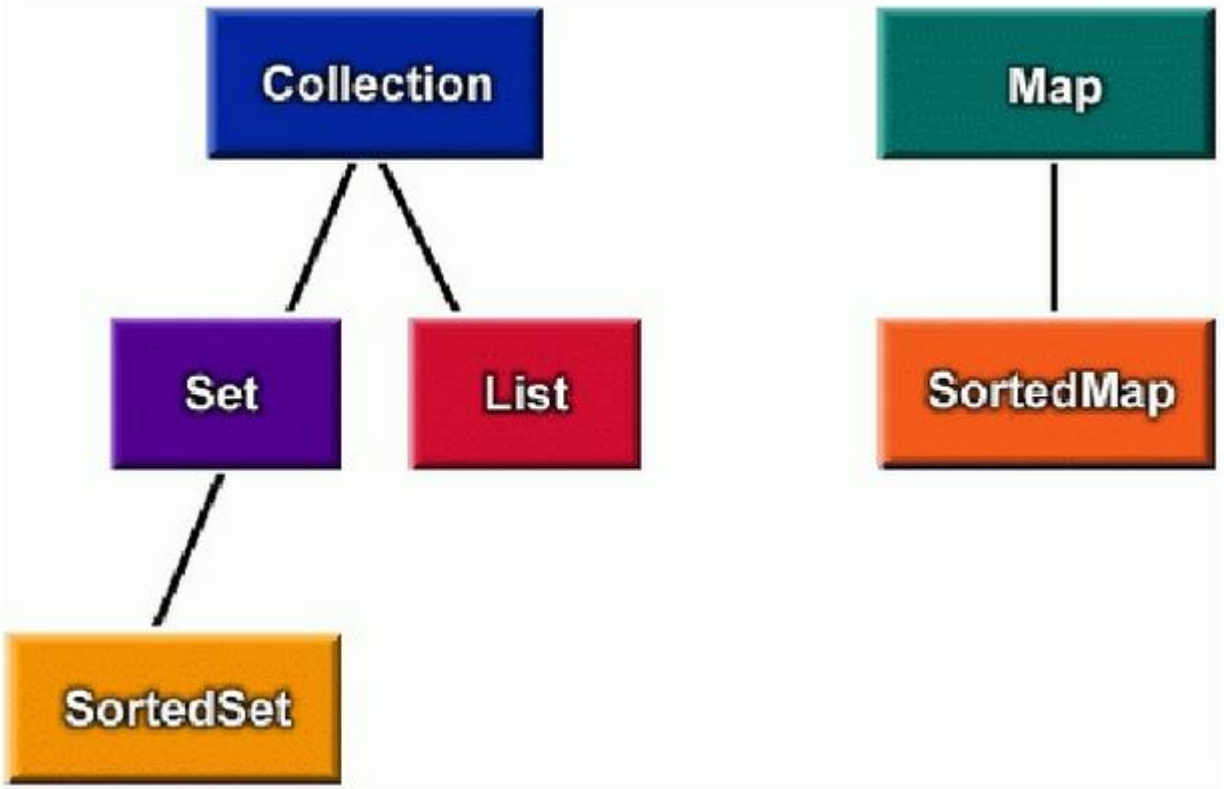
Las colecciones a partir de la versión 1.5

- En las colecciones como ArrayList ahora tenemos que indicar el tipo de los elementos en el momento de crear la colección objetos.
- Ahora los elementos serán todos del mismo tipo, ya no puede haber varios tipos de objetos dentro de la colección.
- Esto evita errores. Ahora al definir una colección indicaremos el tipo de los objetos que vamos a introducir.
- Al recuperar estos objetos ya no tendremos que utilizar un casting.

Tipos de colecciones

- **Collection**
  - Contenedor simple de objetos no ordenados.
  - Los duplicados son permitidos.
- **List**
  - Contenedor de elementos ordenados.
  - Los duplicados son permitidos.
- **Set**
  - Colección desordenada de objetos.
  - Los duplicados no son permitidos.
- **Map**
  - Colección de pares: clave/valor.
  - La clave es usada para indexar el elemento.
  - Los duplicados no son permitidos.

Tipos de Colecciones



Interface List <E>

- Esta interfaz recoge las secuencias de datos en las que:
  - Se respeta el orden en el que se insertan elementos pueden haber elementos duplicados.
  - El tamaño de la lista se adapta dinámicamente a lo que haga falta.
  - El resultado es una especie de *array* de tamaño adaptable.

Implementaciones

- class **ArrayList** implements List
- class **LinkedList** implements List
- NINGUNA de estas implementaciones está preparada para hilos.
- Utilizar métodos wrapper de la clase Collections.
- class **Vector** implements List
- **ArrayList** será la implementación que usemos en la mayoría de situaciones. Sobretodo, varían los tiempos de inserción, búsqueda y eliminación de elementos, siendo en unos casos una solución más óptima que la otra.

ArrayList

- Es una implementación muy eficiente en cuanto a uso de memoria.
- Es rápida en todas las operaciones, excepto en las que afectas a elementos intermedios: inserción y borrado.
- Puede decirse que es un "**array**" de tamaño dinámico.

LinkedList

- Es una implementación basada en listas encadenadas.
- Esto ofrece una **buena velocidad en operaciones sobre términos intermedios (inserción y borrado)** a cambio de ralentizar las demás operaciones.
- Se basa en una lista doblemente enlazada.

Vector

- Soporta métodos **sincronizados**, si no se va utilizar con un Thread es mejor utilizar ArrayList proporciona un mejor rendimiento.

Sincronización

- Ninguna de estas implementaciones son sincronizadas; es decir, no se garantiza el estado del **List** si dos o más hilos acceden de forma concurrente al mismo.
  - List list = Collections.**synchronizedList**(new ArrayList());
  - List list = Collections.**synchronizedList**(new LinkedList());

Métodos de List

- boolean add(E elemento) : añade un elemento al final de la lista.
- void add(int posicion, E elemento) : inserta un elemento en la posición indicada.
- void clear() : vacía la lista.
- boolean contains(E elemento): true si hay en la lista un elemento “equals” al indicado.
- boolean equals(Object x): una lista es igual a otra si contienen en las mismas posiciones elementos que son respectivamente “equals”.
- E get(int posicion): devuelve el elemento en la posición indicada.
- int indexOf(E elemento) : devuelve la posición en la que se haya el primer elemento “equals” al indicado; o -1 si no hay ningún elemento “equals”.
- boolean isEmpty(): true si no hay elementos.

Métodos II

- Iterator <E> iterator() : devuelve un iterador sobre los elementos de la lista.
- E remove(int posicion): elimina el elemento en la posición indicada, devolviendo lo que elimina.
- boolean remove(E elemento) : elimina el primer elemento de la lista que es “equals” el indicado;
  devuelve true si se elimina algún elemento.
- E set(int posicion, E elemento) : reemplaza el elemento X que hubiera en la posición indicada;
  devuelve lo que había antes, es decir X.
- int size() : devuelve el número de elementos en la lista.
- Object[] toArray() : devuelve un array cargado con los elementos de la lista.

Ejemplo

```
List <Integer> lista = new ArrayList <Integer>();

lista.add(1);
lista.add(9);
lista.add(1, 5);
System.out.println(lista.size());
System.out.println(lista.get(0));
System.out.println(lista.get(1));
System.out.println(lista.get(2));

for (int n: lista) {
      System.out.print(n + " ");
}
System.out.println();
```

Map <K, V>

- Mapea **claves únicas** a valores.
  - Una clave es un objeto que se usara para recuperar un valor mas adelante.
- Reciben el nombre de Arrays asociativos.
- A partir de una clave / valor se puede almacenar un elemento en un mapa.
- Después se puede recuperar el elemento a partir de su clave.

Implementaciones de Map

- class **HashMap** implements Map
- class **LinkedHashMap** implements Map
- class **TreeMap** implements Map
- class **Hashtable** implements Map

HashMap

- Es una implementación muy eficiente en cuanto a uso de memoria.
- Es rápida en todas las operaciones.
- Puede decirse que es un "**array asociativo**" de tamaño dinámico.
- ***Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.***

LinkedHashMap

- Es una implementación basada en listas encadenadas.
- Respeta el orden de inserción, a cambio de ser más lenta.
- Es mas lenta que HashMap

TreeMap

- Es una implementación que garantiza el orden de las claves cuanto se itera sobre ellas.
- Es más voluminosa y lenta.
- Cuando se inicializar el TreeMap se le puede pasar una clase que implemente **Comparator<T>** para que el orden de los elementos se mantengan según el criterio de la clase.
  - Map<Object1,Object2> col = TreeMap<>(new ClaseOrden())
  - class ClaseOrden implements Comparator<Object1>{}

Hashtable

- Similar a “HashMap” pero con **métodos sincronizados**, lo que permite ser usada en programas concurrentes.
- Todo es más lento que con una HashMap.

Interface SortedMap<K,V>

- firstkey(), lastKey()
- Set<k> keySet()
- SortedMap<K,V> subMap(k fromKey, k toKey)
- SortedMap<K,V> headMap(K tokey)
- SortedMap<K,V> tailMap(k fromKey)

Métodos de Map

- void clear(): elimina todas las claves y valores.
- boolean containsKey(Object clave): devuelve true si alguna clave es “equals” a la indicada.
- boolean containsValue(Object valor) :devuelve true si algún valor es “equals” al indicado.
- boolean equals(Object x): devuelve true si contiene las mismas claves y valores asociados.
- V get(Object clave): devuelve el valor asociado a la clave indicada.
- boolean isEmpty(): devuelve true si no hay claves ni valores.

Métodos II

- Set<K> keySet() : devuelve el conjunto de claves.
- V put(K clave, V valor) : asocia el valor a la clave; devuelve el valor que estaba asociado anteriormente, o NULL si no había nada para esa clave.
- V remove(Object clave): elimina la clave y el valor asociado;
  devuelve el valor que estaba asociado anteriormente, o NULL si no había nada para esa clave.
- int size() número de asociaciones { clave, valor }
- Collection<V> values() devuelve una estructura de datos iterable sobre los valores asocia

Ejemplo

```
Map <String, String> mapa = new HashMap <String, String>();

    // Agrega elementos al mapa:
    mapa.put("uno", "one");
    mapa.put("dos", "two");
    mapa.put("tres", "three");
    mapa.put("cuatro", "four");
    mapa.put("tres", "33");

    // Extraer el tamaño del mapa, y partir de las claves obtiene los
    // valores:
    System.out.println(mapa.size());
    for (String clave: mapa.keySet()) {
          String valor = mapa.get(clave);
          System.out.println(clave + " -> " + valor);
     }
```

Interface Set <E>

- Esta interfaz recoge los conjuntos de datos que se caracterizan porque:
  - No se respeta el orden en el que se insertan elementos (dependiendo de la implementación elegida).
  - No pueden haber elementos duplicados.
  - El tamaño del conjunto se adapta dinámicamente a lo que haga falta.

Implementaciones de Set

- El propio paquete **java.util** proporciona algunas implementaciones de la interface **Set**, sin perjuicio de que se puedan programar otras.
  - class **HashSet** implements Set
  - class **TreeSet** implements Set
  - class **LinkedHashSet** implements Set
  - **NINGUNA** de estas implementaciones está sincronizada.

HashSet

- Es la que mejor rendimiento tiene.
- Es una implementación muy eficiente en cuanto a uso de memoria.
- Es rápida en todas las operaciones.
- No mantiene el orden de inserción de los elementos.

TreeSet

- Es una implementación más lenta y pesada; pero presenta la ventaja de que el iterador recorre los elementos del conjunto en orden.
- Los elementos deben implementar la interfaz **Comparable**.
- O también al construir TreeSet le podemos pasar una clase que implemente **Comparator<E>**
- Internamente mantiene los elementos en un árbol, operaciones simples rendimiento log(N)

LinkedHashSet

- Mantiene el orden de inserción.
- Mas costosa y lenta que HashSet.
- Internamente mantiene una lista doblemente enlazada.

Interface SortedSet<E>

- Otro interface que hereda de Set, que dispone de los métodos:
  - first(), last();
  - headSet(E toElement)
    - Los elementos menores del elemento pasado.
  - subSet(E fromElement, E toElement)
  - tailSet(E fromElement)
    - Los elementos mayores del elemento pasado.

Métodos Set

- boolean add(E elemento): añade un elemento al conjunto, si no estaba ya; devuelve true si el conjunto crece.
- void clear() : vacía el conjunto.
- boolean contains(E elemento) : devuelve true si existe en el conjunto algún elemento “equals” al indicado.
- boolean equals(Object x): devuelve true si uno y otro conjunto contienen el mismo número de elementos y los de un conjunto son respectivamente “equals” los del otro.
- boolean isEmpty(): devuelve true si el conjunto está vacío.
- Iterator <E> iterator(): devuelve un iterador sobre los elementos del conjunto.
- boolean remove(E elemento): si existe un elemento “equals” al indicado, se elimina;
  devuelve true si varía el tamaño del conjunto.
- int size(): número de elementos en el conjunto

Ejemplo

```
Set <Integer> conjunto = new HashSet <Integer>();

 conjunto.add(1);
 conjunto.add(9);
 conjunto.add(5);
 conjunto.add(9);
 System.out.println(conjunto.size());
 for (int n: conjunto) {
        System.out.print(n + " ");
 }
 System.out.println();
```

Iterator / ListIterator

- Iterator: Provee un mecanismo básico para iterar a través de los elementos de una colección. Solo se mueve hacia delante en la lista.
- ListIterator: Provee soporte para la iteración a través de una lista. Permite recorrer una lista tanto hacia delante como hacia atrás.

Interface Iterator <E>

- Se utiliza para recorrer de forma ordenada los elementos de una colección.

- Métodos:
  - boolean hasNext(): Devuelve true si quedan elementos.
  - E next(): Devuelve el objeto actual y avanza.
  - void remove(): Elimina de la colección el último elemento por next().

Ejemplo

```
// Utilización en bucles:

for (Iterator <E> ite = ...; ite.hasNext(); ) {
    E elemento = ite.next();
    ...
 }

 Iterator <E> ite = ...;
while (ite.hasNext()) {
    E elemento = ite.next();
     ...
}
```

ListIterator

- Es mas completo que Iterator, permite avanzar, retroceder por una colección y aparte puede añadir y modificar elementos.

```
LinkedList l=new LinkedList();
l.add("laxman");
l.add("chandu");
l.add("ravi");
l.add("raju");
System.out.println(l);
ListIterator lt=l.listIterator();
while(lt.hasNext()){
String s=(String)lt.next();
if(s.equals("laxman")){
//lt.remove();
lt.set("scjp");
}
}
System.out.println(l);
}
```

Comparable

- Cuando tenemos colecciones de una clase desarrollada por nosotros, nos puede interesar que estos objetos se ordenen dentro de una colección, para ello la clase debe implementar el interface **Comparable**.
- En este caso tendremos que implementar el método **compareTo**.
- Dentro de este método diremos cuando dos objetos son iguales, mayor o menor.

Ejemplo

- class Persona implements Comparable {
  //....
  public int compareTo(Persona o) {
  return this.nombre.compareTo(o.nombre);
  //... }

- cuando usemos una colección para ordenar, el ordenamiento será automático:

- Set personas = new TreeSet();
  personas.add(new Persona(1, "Mario"));
  personas.add(new Persona(2, "Fernando"));
  personas.add(new Persona(3, "Omar"));
  personas.add(new Persona(4, "Juana"));

  System.out.println("conjunto ordenado de personas: "+personas);

Comparator

- Nos sirve también para comparar pero en este caso puede que nos interese comparar según otro criterio.
- ¿si en otro momento deseamos que sea ordenado por fecha de nacimiento u otro campo sin afectar el campo de ordenamiento predeterminado?
- Para ello debemos utilizar un comparador de elementos. Un comparador es una clase de apoyo que será utilizada para los métodos de ordenamiento. Esto se logra implementando la interfaz java.util.Comparator

Ejemplo

```
class OrdenarPersonaPorId implements Comparator {

        public int compare(Persona o1, Persona o2) {
                return o1.getIdPersona() - o2.getIdPersona();
        }
}
// Desde una lista lo podemos utilizar:
List otrasPersonas = Arrays.asList(new Persona(4, "Juana"),
    new Persona(2, "Fernando"),
    new Persona(1, "Mario"),
    new Persona(3, "Omar"));
    Collections.sort(otrasPersonas, new OrdenarPersonaPorId());
    System.out.println("lista de personas ordenadas por ID:" + otrasPersonas);


// Desde un TreeSet
Set conjuntoPersonas = new TreeSet(new OrdenarPersonaPorId());
    conjuntoPersonas.add(new Persona(3, "Omar"));
    conjuntoPersonas.add(new Persona(4, "Juana"));
    conjuntoPersonas.add(new Persona(2, "Fernando"));
    conjuntoPersonas.add(new Persona(1, "Mario"));

    System.out.println("conjunto de personas ordenadas por ID:" + conjuntoPersonas);
```

Implementación de Algoritmos

- La estructura Pila → Primero en entrar ultimo en Salir:
  - Pila de los objetos que queramos.

- La estructura Cola → Primero en entrar – Primero en salir.
  - Cola de los objeto que queremos.

Practicas: Colecciones

El nuevo bucle for

- for (tipo_primitivo var : array)

  // Instrucciones.

- for (clase obj : colección_objetos)

  // Instrucciones.

- Se traduce por para cada objeto …

- Controla automáticamente el final del array.

Ejemplo del bucle for

```
public class PruebaV5 {

        public static void main(String args[]){

                int []numeros = { 1, 2, 3};

                // Para cada entero dentro del array:
                for (int i: numeros)
                        System.out.println(i);
        }
}
```

Arrays

- Dispone de métodos generales para el uso de arrays y colecciones.

- Tiene métodos (son **static**) para:
  - Para cada tipo primitivo java.
  - **asList**: Para inicializar una lista.
  - **binarySearch**: realizar búsquedas.
  - **copyOf**: para copiar un array.
  - **copyOfRange**: copiar un rango de un array
  - **equals**: para comparar dos arrays.
  - **toString**: para convertir un array en String
  - **sort**: Para ordenar un array.
  - **fill**: Para rellenar los elementos de un array

Collections

- Dispone de métodos generales (son static) para trabajar con colecciones.
- Por ejemplo:
  - **addAll**: para añadir elementos a una colección.
  - **binarySearch**: buscar un elemento en la colección.
  - **copy**: Para copiar colecciones.
  - **disjoint**: comprueba si las dos colecciones no tienen elementos en común.
  - **Métodos empty** con cada tipo de colección que crean List, Set, Map, vacíos e inmutables.
  - **fill**: Rellenar los elementos de una colección.
  - **indexOfSubList**: Buscar el inicio de una lista dentro de otra lista.
  - **lastIndexOfSubList**: Como la anterior, pero busca por el final.
  - **max, min**: localizar el máximo y el mínimo.
  - **replaceAll**: Remplazar todos los elementos de la lista, indicando un valor y el nuevo.
  - **sort**: Para ordenar una lista, se puede indicar una clase que implemente **Comparator**.