Colecciones - parte 2

Interfaz Queue

Es una colección diseñada para contener elementos anteriormente a que se procesen. Ademas de las operaciones basicas de una coleccion, provee operaciones adicionales de inserción, extracción e inspeccion. Cada uno de estos métodos existen en dos formas: una arroja una Exception en caso de que la operación falle, la otra devuelve un valor especial (null o False, dependiendo de la operacion).
Las colas típicamente, pero no necesariamente, ordenan sus elementos de la forma FIFO (Firs In Firs Out). También se encuentran las colas LIFO (Last In First Out, tambien llamadas Stacks o Pilas).
Más allá del orden que se utilice, en todas las colas la cabeza o head de la cola es el elemento que se remueve llamando al método remove() o poll().
Podemos destacar dos implementaciones para la interfaz Queue:

1)LinkedList
2) PriorityQueue

LinkedList

Como vimos anteriormente LinkedList implementa la interfaz List, pero si observamos la imagen con la jerarquía de interfaces observaremos que tambien implementa Queue.

[code]

```
        public static void main(String args[]){

        Queue queueA = new LinkedList();

        // Metodos que arrojan execpciones ante una falla
        queueA.add("element 1"); // inserta un elemento en la cola
        queueA.add("element 2");
        queueA.add("element 3");
        queueA.remove(); // remueve y devuelve el head de una cola
        queueA.element(); //  devuelve, pero no remueve, el head de una cola

        // Metodos que devuelven null o false ante una falla.
        queueA.offer("element 4"); // inserta un elemento en la cola,  puede fallar en el caso de una cola
con capacidad limitada
        queueA.poll(); // remueve y devuelve el head de una cola
1       queueA.peek(); //  devuelve, pero no remueve, el head de una cola
   }
```

[/code]

PriorityQueue

Las priority queues ordenan sus elementos acorde a un comparador provisto, o el orden natural de los elementos.

[code]

```
public static void main(String[] args){
```

```
        Comparator<String> comparator = new StringLengthComparator();
        PriorityQueue<String> queue = new PriorityQueue<String>(10, comparator);
        queue.add("short");
        queue.add("very long indeed");
        queue.add("medium");
        while (queue.size() != 0)
        {
            System.out.println(queue.remove());
        }
    }
```
[/code]

Interfaz Deque

Una Deque interfaz también conocida como double-queue provee insercion y extraccion de elementos tanto del final como del principio de la cola.

Iteradores

Definición

Es un patrón de diseno utilizado para recorrer las colecciones, abstrayendo al usuario de la implementación de la colección. En Java es una interfaz denominada Iterator. Esta formado por tres métodos:

boolean hasNext(), retorna true en caso de haber más elementos y false en caso de llegar al final del iterador.

Object next(), retorna el siguiente elemento en la iteración.

void remove(), remueve un objeto de la colección.

En las clases Vector, ArrayList, HashSet y TreeSet un iterador se consigue a través del método: Iterator iterator()

Utilización

Construye un contenedor, en este caso un ArrayList

Ejemplo:

[code]
```
 ArrayList lasPersonas = new ArrayList();
  lasPersonas.add("Pepe");
  lasPersonas.add("Juan");
  lasPersonas.add("Sabrina");
  lasPersonas.add("Cecilia");
  Iterator it = lasPersonas.iterator();
  while(it.hasNext()){
            String unaPersona = (String)it.next();
  }
```
[/code]

La interfaz Map

La interfaz Map, asocia claves a valores. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.

La interfaz Map provee tuplas del tipo Key-value, los objetos Map contienen claves o Keys asociadas con valores o values. Los Maps no pueden contener claves duplicadas y una clave puede ser asociada como máximo a un elemento. Existen tres implementaciones destacables:

1)HashMap
2)TreeMap
3)LinkedHashMap

HashMap

Una HashMap es una implementación del tipo HashTable de la interfaz MAp, al contrario de una HashTable puede contener claves y valores null. HashMap no garantiza que el orden de los objetos será el mismo a través del tiempo.

[code]
```
  public static void main(String args[]){
    HashMap<Integer, String> hashMap = new HashMap<Integer, String>();
    hashMap.put(0,"value1");
    hashMap.put(1,"value2");
    hashMap.put(2,"value3");
    System.out.println(hashMap);
    hashMap.remove(0);
    System.out.println(hashMap);
  }
```
[/code]

TreeMap

TreeMap provee una implementación de Map basada en una estructura de datos del tipo arbol red-black.

[code]
```
 public static void main(String args[]){

        TreeMap<String, Integer> treeMap = new TreeMap<String, Integer>();

        treeMap.put("key1",123);
        treeMap.put("key2",435);
        treeMap.put("key3",654);
        System.out.println(treeMap);

        treeMap.remove("key1");
        System.out.println(treeMap);
    }
```
[/code]

LinkedHashMap

Es una implementación de Map con una HashTable y una LinkedList. LinkedHashMap tiene una doble

lista linkeada a través de sus elementos.

[code]
public static void main(String args[]){

HashMap<Integer, String> linkedHashMap = new HashMap<Integer, String>();
linkedHashMap.put(0,"value1");
linkedHashMap.put(1,"value2");
linkedHashMap.put(2,"value3");

System.out.println(linkedHashMap);
linkedHashMap.remove(0);
System.out.println(linkedHashMap);
}
[/code]

Collection Implementations

Las clases que implementan la interfaz Collection por lo general tienen nombres que indican el tipo de implementación a bajo nivel, este dato es muy importante porque la performance de cada una de las clases varía de acuerdo a como se la vaya a utilizar.

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

COMPARABLE Y COMPARATOR

Comparable

Los elementos en una colección en Java se puede ordenar de la siguiente forma:

[code]
List<String> nombres=Arrays.asList("Cesar","Amanda","Dario","Benjamin");
System.out.println("lista original:"+nombres);

Collections.sort(nombres);
System.out.println("lista ordenada:"+nombres);
[/code]

El método Collections.sort() ordena cualquier lista.

Veamos también el siguiente ejemplo, donde se agregan elementos a un conjunto (java.util.Set). Recordemos que la clase TreeSet mantiene sus elementos ordenados:

[code]
```
    Set nombres = new TreeSet();
    nombres.add("Mario");
    nombres.add("Fernando");
    nombres.add("Omar");
    nombres.add("Juana");

    System.out.println("conjunto ordenado:" + nombres);
```
[/code]

Esto funciona correctamente, porque los elementos de las colecciones son comparables entre sí. Para que un objeto sea comparable, su clase debe implementar la interfaz java.lang.Comparable.
De esta forma, si queremos que una lista de objetos (o un conjunto) tenga sus elementos ordenados, y esos objetos son de una clase que hemos programado, es necesario que nuestra clase implemente la interfaz java.lang.Comparable.
Veamos el siguiente ejemplo, una clase Persona, que tiene algunas propiedades:

[code]
```
class Persona {

  private int id;
  private String nombre;
  private java.util.Date fechaNacimiento;

  public Persona() {
  }

  public Persona(int id, String nombre) {
     this.id = id;
     this.nombre = nombre;

  }

  public int getId() {
     return id;
  }

  public void setId(int id) {
     this.id = id;
  }

  public String getNombre() {
     return nombre;
  }

  public void setNombre(String nombre) {
```

```
        this.nombre = nombre;
    }

    public java.util.Date getFechaNacimiento() {
        return fechaNacimiento;
    }

    public void setFechaNacimiento(java.util.Date fechaNacimiento) {
        this.fechaNacimiento = fechaNacimiento;
    }

    @Override
    public String toString() {
        return String.format("persona{id:%1s,nombre:%2s}", id, nombre);
    }
}
[/code]
```

Para que los objetos al guardarse en una colección (Set o List) estén ordenados por el campo nombre, debemos implementar la interfaz Comparable.

```
class Persona implements Comparable{....
```

Y ademas implementar su método de comparación, aquí es donde definimos qué campo vamos a utilizar para ordenar:

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

De esta forma, al utilizar 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);
```

El ordenamiento natural será a través del campo **nombre** de **Persona.**

Comparator

Si deseamos ordenar una colección por un criterio diferente al natural debemos utilizar un comparador de elementos. Esto se logra implementando la interfaz java.util.Comparator
Para continuar con nuestro ejemplo, creemos la siguiente clase que implemente la interfaz Comparator:

```
class OrdenarPersonaPorId implements Comparator {

  public int compare(Persona o1, Persona o2) {
     return o1.getId() - o2.getId();
  }
}
```

El método compare() debe devolver lo siguiente:

Sí o1 < o2 debe devolver un número menor a cero.
Sí o1 == o2 debe devolver cero.
Sí o1 > o2 debe devolver un número mayor a cero.

Para utilizar este comparador, debemos usar el parámetro adicional de Collections.sort()

```
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);
```

Sí queremos utilizar un TreeSet podemos pasar como parámetro el comparador en el constructor de java.util.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);
```