LPS: Colecciones

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

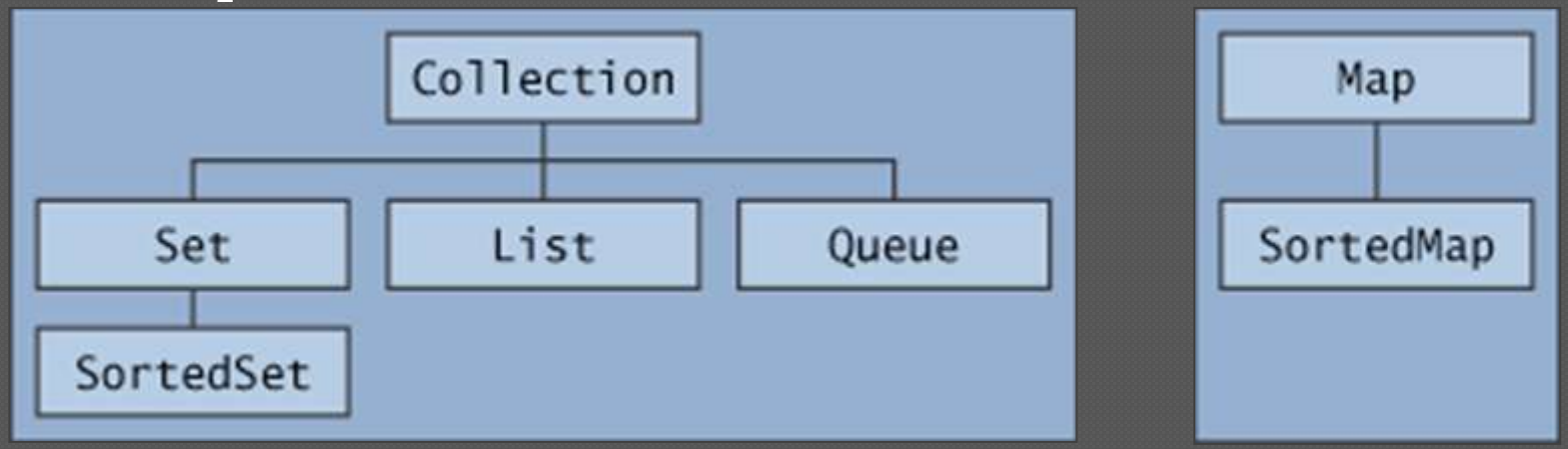
Universidad Complutense de Madrid
www.ucm.es

Colecciones

- Toda colección es un “grupo” de “elementos”
- Conjunto de interfaces y clases Java que representan estructuras de datos habituales
  - *Java Collections Framework*
  - La mayoría están en el paquete `java.util`
- Tutoriales Java sobre colecciones
  http://java.sun.com/docs/books/tutorial/collections/index.html
- Thinking in Java (4th Edition)
  - Holding your objects (páginas 275– 311)

Organización

- Las colecciones se organizan en un interfaz por cada tipo de estructura y varias clases (a veces *abstractas*) con diversas implementaciones posibles de cada interfaz
  - Jerarquía de interfaces:



  - Ejemplo: `List<String> l = new ArrayList<String>();`

Previo: Genéricos

- Mecanismo introducido en Java 1.5 para definir clases "genéricas" que admiten ser "especificadas" mediante otra clase adicional
  - `ClaseA<ClaseB>` ó incluso `ClaseA<? extends ClaseB>` …
  - Recuerda a las *plantillas* de C++ aunque no es un mecanismo tan potente
  - Es sólo una notación útil en tiempo de compilación (sólo para compiladores de Java >=1.5), pero no cambia nada a nivel de ejecución
    - Nos ahorra estar constantemente haciendo conversiones (*castings*) y capturando las posibles excepciones
    - Sirve principalmente sirve para tener colecciones "genéricas"

Colecciones y genericidad

- Antiguamente (sin usar clases "genéricas")
  - Se asumía que todas las colecciones tenían elementos de tipo *Object*
  - Al sacarlos, debíamos hacer una conversión al tipo concreto

```
public Coche primerCoche(List l) {
        Coche c = (Coche) l.get(0);
        return c;
}
```

- Actualmente (usando clases "genéricas")
  - Las colecciones "genéricas" permiten especificar el tipo de sus elementos
  - Esto permite corregir errores de tipos en tiempo de compilación

```
public Coche primerCoche(List<Coche> lcoches) {
        Coche c = lcoches.get(0);
        return c;
}
```

Interfaz Collection

```
public interface Collection<E> extends Iterable<E> {

    // Operaciones básicas
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();

    // Operaciones masivas
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Operaciones de arrays
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Recorridos de colecciones

- Sentencia for-each (desde Java 1.5)
  - Bucle para recorrer colecciones de elementos
  - Se puede también usar con arrays primitivos

```
Vector<String> nombres = new Vector<String>(10);
for (String s:nombres)
   System.out.println(s);
```

- Enumeraciones
  - Interfaz Enumeration para acceder al siguiente elemento de una enumeración (método **`nextElement`**) y consultar si hay más elementos por recorrer (método **`hasMoreElements`**)
  - Podemos obtener enumeraciones a partir de muchas colecciones (métodos **`Vector.elements`**, **`Hashtable.elements`**, **`Hashtable.keys`** ...)

```
Vector<String> nombres = new Vector<String>(10);
for(Enumeration e = nombres.elements(); e.hasMoreElements(); )
   System.out.println(e.nextElement());
```

Recorridos de colecciones

- Iteradores
  - Interfaz Iterator de acceso al siguiente elemento a recorrer (método **`next`**) y consulta de siguiente (método **`hasNext`**)
  - Método **`remove`** para borrar el último elemento recorrido
    - Sólo se puede llamar una vez por cada llamada a **`next`**
    - ¡Único método fiable para eliminar elementos en un recorrido!
  - Muchas colecciones tienen un método **`iterator`** que devuelve el iterador para recorrerla (si los elementos siguen algún orden o no, *depende de la colección*)

```
Vector<String> nombres = new Vector<String>(10);
Iterator<String> it = nombres.iterator();
while (it.hasNext())
    System.out.prinln(it.next());
```

Listas

Listas

- Interfaz List para colecciones secuenciales
  - Acceso por posición: `get(int)`, `set(int, Object)` y `add(int, Object)`
  - Búsquedas: `indexOf(Object)` y `lastIndexOf(Object)`
  - Recorridos con iteradores: `listIterator()` y `listIterator(int)`
  - Selección de sublistas: `subList(int, int)`
- Implementaciones principales
  - ArrayList: Como los arrays pero con redimensión automática
  - Vector: Como ArrayList pero que permite acceder de forma fiable desde varios hilos de ejecución
  - LinkedList: Listas doblemente enlazadas con inserción por delante y por detrás (se puede usar a modo de cola)
  - Stack: Pila implementada usando internamente la clase Vector

Algoritmos para listas

- Las colecciones también proporcionan algunos algoritmos que, generalmente, se aplican sobre listas
- Son métodos estáticos de la clase **`java.util.Collections`**
  - Ordenación: **`sort(List, Comparator)`**

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

  - Desorden (aleatorio): **`shuffle()`**
  - Búsqueda binaria: **`binarySearch(List<T>, T, Comparator)`**
  - Obtención de "datos estadísticos": **`frequency(Collection, Object)`**, **`max(Collection)`** y **`min(Collection)`**
  - Manipulación de datos: **`reverse`**, **`fill`**, **`copy`**, **`swap`** ...

Colas

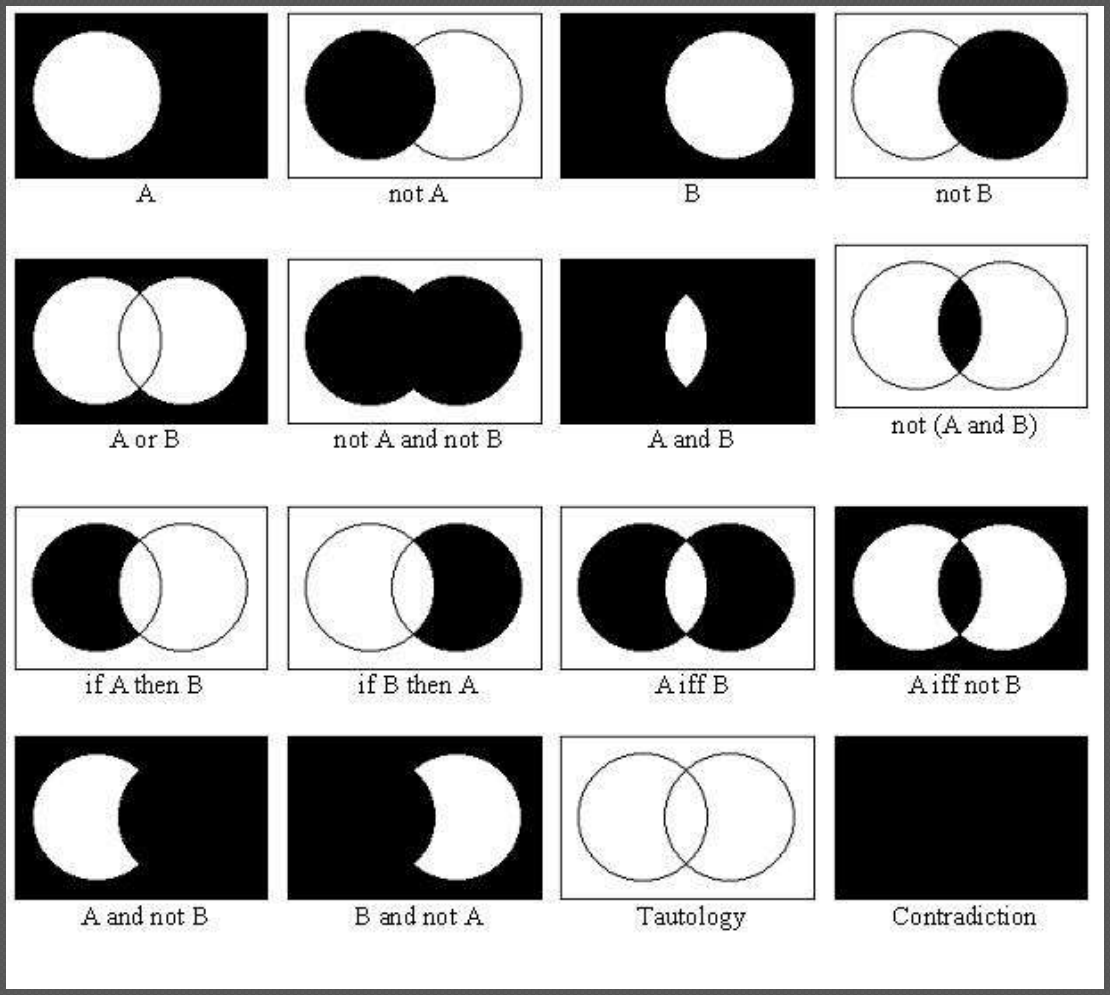
Curioso lo que sale en Google…

Colas

- Interfaz Queue para colecciones según orden de llegada (normalmente FIFO - *First In First Out*)
- Varias implementaciones disponibles, incluso de colas de prioridad (PriorityQueue)
- Métodos para añadir elementos a la cola y extraer de la cabeza de la cola según dos filosofías
  1. Lanzando excepciones en caso de error
  2. Devolviendo valores "especiales" en caso de error

	Excepciones	**Valores especiales**
Inserción	`add(e)`	`offer(e)`
Extracción	`remove()`	`poll()`
Acceso/Revisión	`element()`	`peek()`

Conjuntos


A
not A
B
not B
A or B
not A and not B
A and B
not (A and B)
if A then B
if B then A
A iff B
A iff not B
A and not B
B and not A
Tautology
Contradiction

Conjuntos

- Interfaz Set para colecciones sin repeticiones
  - Un par de conjuntos siempre se pueden comparar, sin necesidad de que tengan la misma implementación
- Implementaciones principales
  - HashSet: Como una tabla *hash*
    - La más eficiente
    - Recorrido mediante iteradores *sin ningún orden*
  - TreeSet: Como un árbol de tipo "rojo-negro"
    - La menos eficiente
    - Garantiza recorridos *ordenados de acuerdo a sus elementos*
  - LinkedHashSet: Tabla *hash* más lista enlazada
    - Coste algo mayor que HashSet
    - Garantiza recorridos ordenados *por orden de inserción*

Mapas


FORODWAITH
Eredd Mithrin
Iron Hills
The Lost Realm Of
ARNOR
HITHAEGLIR
ERED
LUIN
Eriador
(The Lonely
Mountain)
Esgaroth on the
Long Lake
Weather Hills
Rhudaur
RHÛN
ARTHEDAIN
Old Forest
The Shire
Bree
Rivendell
Mirkwood
East Bight
Blue Mountains
Harlindon
Sea of
Rhûn
ERIADOR
Enedwaith
MINHIRIATH
Dunland
RHOVANION
Fangorn
Isengard
The Brown Lands
ROHAN
ENEDWAITH
Helms
Deep
Eredd Lithui (Ash Mountains)
Gladden
Fields
Burial
Plateau of
Gorgoroth
MORDOR
GONDOR
NURN
Anfalas
Lebennin
Pelennor
Fields
The West Of
Middle Earth
At The End Of
The Third Age

Mapas

- Interfaz Map para tablas "de dispersión" (*hash*) que relacionan claves con sus valores respectivos
  - No permite claves duplicadas
  - Debería traducirse como "tablas de correspondencia" ☹
- Tres métodos disponibles para ver el mapa en forma de colección
  - **`keySet`**: Devuelve las claves como conjunto
  - **`values`**: Devuelve los valores como colección
  - **`entrySet`**: Devuelve los pares clave-valor como conjunto
- Implementaciones
  - HashMap, TreeMap y LinkedHashMap
  - EnumMap: Especializada para claves de tipo enumerado

Críticas, dudas, sugerencias…

Federico Peinado

www.federicopeinado.es