Estructures de dades en Java

Algorismes i estructures de dades

Un tema que apareix freqüentment en la programació és el d'organitzar una col·lecció d'objectes en memòria. Aquest tema també és conegut amb el títol "estructures de dades", i es relaciona també amb diferents algorismes per a agilitar l'accés.

La primera estructura de dades és la variable. Emmagatzema un sol objecte, i el temps d'accés és ínfim. La segona estructura de dades que sempre s'aprèn és array. Aquesta col·lecció ens permet obtenir el valor d'un element, donada la seva posició en una taula. És molt veloç ja que aquesta estructura té un fort paral·lel amb com s'organitza internament la memòria d'una computadora.

Un array en Java es declara així: T [] llista;

```
String [] noms = new String[10];
```

Exemple

Si volem emmagatzemar les notes dels alumnes d'una classe, podem utilitzar un array d'elements Nota, cadascun dels quals encapsula una nota.

```
class Nota
{
    String alumne;
    int nota;
    public Nota(String alumne, int nota)
    {
        this.alumne = alumne;
        this.nota = nota;
    }
    public int getNota() { return nota; }
    public String getAlumne() { return alumne; }
    public void setNota(int nota) { this.nota=nota; }
    public void setAlumne(String alumne) { this.alumne=alumne; }
}
```

Ara emmagatzemem les notes de 1000 alumnes:

```
Nota [] notes = new Nota[1000];
notes[0] = new Nota("Carles", 3);
notes[1] = new Nota("Antoni", 8);
notes[2] = new Nota("Pere", 4);
notes[3] = new Nota("Joan", 6);
// ...
notes[499] = new Nota("Francesc", 9);
```

Per buscar un element en concret, haurem de fer una **cerca seqüencial**, atès que no coneixem la seva posició.

```
int buscaNota(Nota notes, String alumne)
{
    int i = 0;
    while ( !notes[i].getAlumne.equals(alumne) )  i++;
    if (i<notes.length-1) return i;
    else return -1;
}
```

En el cas que els alumnes estiguin ordenats alfabèticament, podem comparar l'alumne buscat amb el que ocupa la posició central (500). Si és menor, comparem amb el que ocupa la posició central de la subllista esquerra (el 250), en cas contrari amb el de la dreta (750). Repetint aquest procediment successivament, podem trobar-lo més ràpidament. Aquest algorisme s'anomena **recerca dicotòmica**.

El principal inconvenient amb les llistes ordenades és el de la inserció de nous elements.

Col·leccions en Java

Java té des de la versió 1.2 tot un joc de classes i interfícies per a guardar col·leccions d'objectes. Totes les entitats conceptuals estan representades per interfícies, i les classes s'usen per a proveir implementacions d'aquestes interfícies. Una introducció conceptual ha d'enfocar-se primer en aquestes interfícies.

La interfície ens diu què podem fer amb un objecte. La interfície no és la descripció sencera de l'objecte, solament un mínim que ha de complir.

Com correspon a un llenguatge tan orientat a objectes, aquestes classes i interfícies estan estructurades en una jerarquia. A mesura que es va descendint a nivells més específics augmenten els requeriments i el que se li demana a aquest objecte que sàpiga fer.

Collection

La interfície més important és **Collection**. Una Collection és tot allò que es pot recórrer (o iterar) i del que es pot saber la grandària. Moltes altres classes estendran Collection imposant més restriccions i donant més funcionalitats. És de notar que el requisit "que se sàpiga la grandària" fa inconvenient utilitzar aquestes classes amb col·leccions d'objectes de les quals no se sàpiga "a priori" la quantitat.

No es pot instanciar cap objecte de classe Collection. Les operacions bàsiques d'una Collection són:

add(T)	Afegeix un element
iterator()	Obté un iterador que permet recórrer la col·lecció visitant cada element una vegada
size()	Obté la quantitat d'elements que aquesta col·lecció emmagatzema
contains(t)	Pregunta si l'element t ja està dintre de la col·lecció

Amb la classe Collection hi ha moltes coses que no es poden assumir d'antuvi. No es pot assumir que l'ordre amb què recorrem la col·lecció sigui rellevant, és a dir, que si la recorrem de nou vegem els elements en el mateix ordre en el qual els varem veure la primera vegada. Tampoc no podem assumir que no hi ha duplicats. No podem assumir característiques de rendiment: Preguntar si existeix un objecte en la col·lecció pot trigar des de molt poc a molt de temps, segons l'organització de les dades en memòria.

Una capacitat d'un objecte Collection és la de poder ser recorregut. Com que a aquest nivell no està definit un ordre, l'única manera és proveint un **iterador**, mitjançant el mètode **iterator()**. Un iterador és un objecte "passejador" que ens permet anar obtenint tots els objectes en anar invocant successivament el seu mètode **next()**. També, si la col·lecció és modificable, podem eliminar un objecte durant el recorregut mitjançant el mètode **remove()** de l'iterador. El següent exemple recorre una col·lecció de `Integer` esborrant tots els zeros:

```
void esborrarZeros(Collection<Integer> zeros)
{
    Iterator<Integer> it = zeros.iterator();
    while ( it.hasNext() )
    {
```

```
            int i = it.next();
            if ( i == 0 )
            it.remove();
        }
}
```

En aquest exemple es fa ús de la conversió automàtica entre Integer i int.

A partir de **Java 6** hi ha una manera simplificada de recórrer una Collection (que serveix si no necessitem esborrar elements). Es fa mitjançant un nou ús d'una sentència **for**:

```
void mostrar(Collection col)
{
    for ( Object o : col )
        System.out.println(o);
}
```

Internament aquest codi no fa altra cosa que obtenir l'iterador, però queda molt més elegant i llegible d'aquesta manera.

Hi ha quatre interfícies que estenen Collection: **List**, **Set**, **Map** i **Queue**. Cadascuna d'elles agrega condicions sobre les dades que emmagatzema i com a contrapartida ofereix més funcionalitat.

List

Una **List**, o simplement **llista**, és una Collection que manté un ordre dels elements i permet accedir als elements per ordre. Podríem dir que en una llista, en general, l'ordre ve donat. És a dir, l'ordre és informació important que la llista també ens està emmagatzemant. En canvi no hi ha cap mètode en Collection per a obtenir l'element amb un número d'ordre determinat.

Una llista llavors agrega els següents mètodes:

get(int i)	Obté l'element en la posició i
set(int i, T t)	Posa l'element t en la posició i

Set

Un **Set** és una Collection destinada a representar el que matemàticament es coneix com conjunt. Es tracta bàsicament d'una Collection amb la restricció que **no pot haver-hi duplicats**.

En general, en un Set no hi ha cap relació d'ordre establerta.

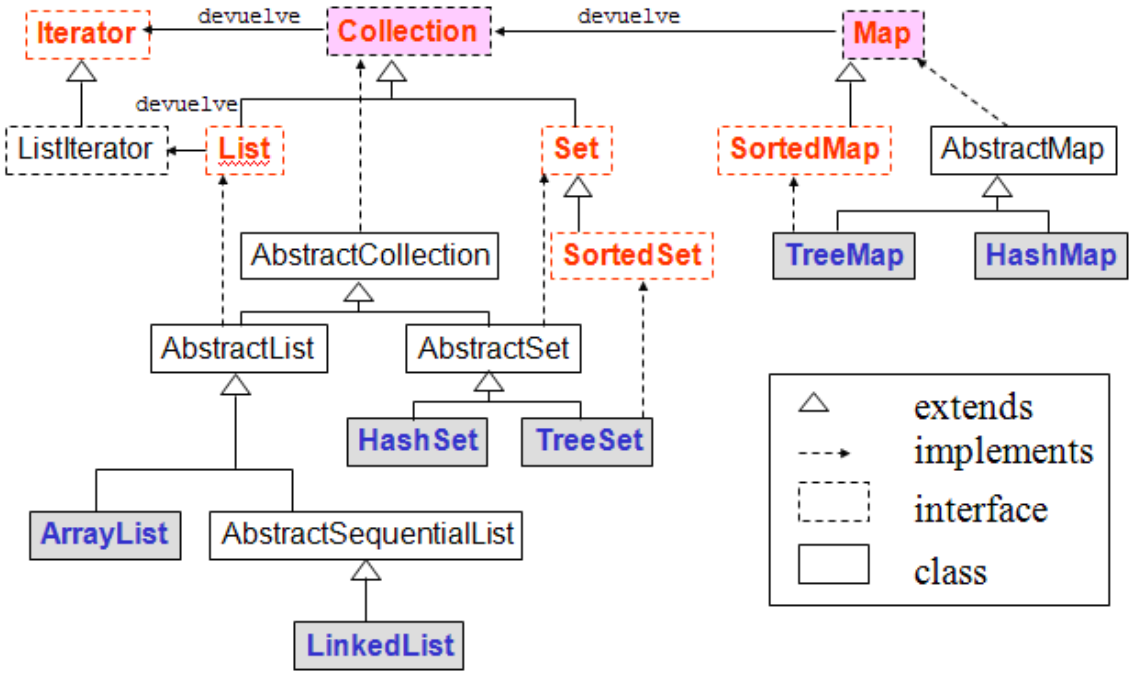
Map

Un **Map** representa el que en altres llenguatges es coneix com a diccionari. Es tracta d'un conjunt de parelles **clau : valor**. Les claus permeten accedir als valors i no poden estar duplicades. Sol associar-se amb el concepte de **taula hash**.

Un Map no és una Collection ja que aquesta interfície és unidimensional, mentre que el Map és bidimensional.

Jerarquia de les col·leccions

Des de Collection i Map, existeixen tot un seguit de classes abstractes i interfaces fins a unes classes, les quals són les que habitualment instanciarem, tal com es pot veure a la figura següent.



Les classes d'ús més habitual seran les ArrayList, LinkedList, HashSet, TreeSet, TreeMap, HashMap.

ArrayList s'implementa amb un array, mentre que LinkedList s'implementa amb una llista enllaçada. El primer és més eficient per a l'accés directe, en canvi el segon és més eficient per a la modificació de l'ordenació (inserció i extracció pel mig de la llista). Els Map són parelles clau:valor. Els Hash s'emmagatzemen amb una clau hash, la qual permet un accés molt ràpid.

Recorregut d'una col·lecció

El següent exemple il·lustra diferents formes d'efectuar el recorregut sobre els elements d'una col·lecció. En aquest cas, es recorre un ArrayList d'enters. A partir de la versió 6 de Java, les col·leccions són plantilles (templates) que representen en realitat una varietat de classes diferents, segons la classe dels elements que les componen. La classe (tipus) de l'element s'especifica entre els símbols <>.

```
/**
 * VectorAp.java
 * Il·lustra l'ús de col·leccions en Java
 * @author Jose Moreno
 * @version
 */
import java.util.*;
public class VectorAp {
	public static void main(String[] args) {
		List<Integer> vect = new ArrayList<Integer>();
		vect.add(1);
		vect.add(2);
		System.out.println( "Impressió amb toString" );
		System.out.println( vect.toString() );
		System.out.println( "Recorregut amb un iterador" );
		for ( Iterator it=vect.iterator(); it.hasNext(); )
			System.out.println( it.next() );
		System.out.println( "Recorregut amb bucle for (a partir del java 6)" );
		for ( Object o: vect )
```

```
                System.out.println( o.toString() );
        }
}
```