**Tema 10: COLECCIONES**

**INTRODUCCIÓN**

Hay veces en que los arrays no son adecuados o suficientes por las características del problema que haya que resolver. Por ejemplo, cuando se necesiten almacenar elementos pero no se conozca cuántos. Para solucionar estos problemas surgen las llamadas Collections (Colecciones) que son estructuras de almacenamiento que crecen dinámicamente. Esto quiere decir que no tienen un tamaño determinado: se les puede asignar un tamaño de inicio, pero éste variará automáticamente si se introducen más elementos a la estructura.

Laversión1.2 del JDK introdujo el Java Framework Collections o “estructura de colecciones de Java” (JCF).

Son un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos, también se las conoce como contenedores de objetos, a diferencia de los arrays no permiten hacer uso de tipos primitivos,. Por tanto, siempre que necesitemos trabajar con ellos habrá que hacer uso de los Wrappers de Tipos Primitivos.

Los ***Wrappers*** (*envoltorios*) son clases diseñadas para ser un ***complemento*** de los ***tipos primitivos***. En efecto, los tipos primitivos son los únicos elementos de ***Java*** que no son objetos. Existe una clase ***Wrapper*** para cada uno de los tipos primitivos numéricos, esto es, existen las clases ***Byte***, ***Short***, ***Integer***, ***Long***, ***Float*** y ***Double*** (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de ***Java***).

Todas las colecciones se encuentran en el paquete java.util.\*;

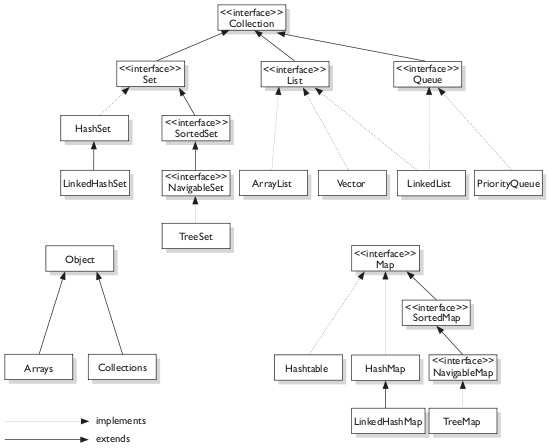
java.util.Collection es la raíz de la jerarquía de las colecciones, es una interfaz NO una clase.

De entre las interfaces que extienden Collection<E> las más interesantes son List<E>, Set<E> y Queue<E> que definen, respectivamente, listas, conjuntos y colas.

La lista es una de las colecciones más básicas, se trata de una estructura de datos secuencial, en la que cada elemento tiene una posición o índice, y que permite elementos duplicados. Set<E> es la interfaz utilizada para modelar los conjuntos matemáticos; como en estos, los elementos no tienen un orden, y no se permiten elementos duplicados. Una cola o Queue<E>, por último, es una estructura de datos de tipo FIFO (First in first out) lo que significa que el primer elemento en introducirse en la cola será el elemento que se devolverá al extraer por primera vez de la cola, y así sucesivamente (en realidad existen implementaciones de Queue<E> que no son FIFO, pero eso queda fuera del enfoque de esta entrada).

Otra estructura de datos que forma parte del framework, aunque no deriva de Collection<E>, dado que no se trata de una colección per se sino de un mapeo, es la que define la interfaz Map<K,V>, el conocido diccionario o matriz asociativa, en el que cada valor tiene asociado una clave que usaremos para recuperar el elemento, en lugar de un índice como el caso de las listas. Por ejemplo podríamos tener un mapeo en el que las claves fueran los días de la semana y los valores, el número de líneas de código que escribimos.

**JERARQUÍA DE INTERFACES Y CLASES**



Collections es considerado un framework que contiene:

* **Interfaces:** Tipos de datos abstractos que representan las collections.
* **Implementaciones**: Implementación especializada de las interfaces del Framework Collections
* **Algoritmos**: Métodos que permite realizar operaciones como ordenamientos, búsquedas, etc

**INTERFAZ Collection**

Esta interfaz define métodos muy interesantes para trabajar con listas que diversas clases implementan. Entre ellos:

|  |  |
| --- | --- |
| método | uso |
| **boolean add(Object** o**)** | Añade el objeto a la colección. Devuelve **true** si se pudo completar la operación. Si no cambió la colección como resultado de la operación devuelve **false** |
| **boolean remove(Object** o**)** | Elimina al objeto indicado de la colección |
| **int size()** | Devuelve el número de objetos almacenados en la colección |
| **boolean isEmpty()** | Indica si la colección está vacía |
| **boolean contains(Object** o**)** | Devuelve **true** si la colección contiene al objeto indicado ***o*** |
| **void clear()** | Elimina todos los elementos de la colección |
| **boolean addAll(Collection** otra**)** | Añade todos los elementos de la colección ***otra*** a la colección actual |
| **boolean removeAll(Collection** otra**)** | Elimina todos los objetos de la colección actual que estén en la colección ***otra*** |
| **boolean retainAll(Collection** otra**)** | Elimina todos los elementos de la colección que no estén en la otra |
| **boolean containsAll(Collection** otra**)** | Indica si la colección contiene todos los elementos de otra |
| **Object[] toArray()** | Convierte la colección en un array de objetos. |
| **Iterator iterator()** | Obtiene el objeto iterador de la colección, se explica en el punto siguiente |

Hay que tener en cuenta que en la interfaz se considera que las colecciones las forman objetos genéricos (de la clase **Object**) que al ser padre de cualquier clase permite manipular sin problemas objetos del tipo que sea; no obstante habrá que hacer continuos ***castings*** para convertir tipos genéricos en el tipo concreto a manipular.

**INTERFAZ Iterator**

Un iterador es un objeto que nos permite obtener uno a uno todos y cada uno de los elementos de una colección.

Los iteradores permiten recorrer colecciones. Disponen de un conjunto de métodos que permiten avanzar sobre la colección y obtener los objetos de ésta durante un recorrido para su tratamiento.

Existen dos interfaces declarados en el JCF, que son ***java.util.Iterator*** (dispone de métodos para recorrer una colección y para borrar elementos) y ***java.util.ListIterator*** (permite recorrer una lista en ambos sentidos) siendo el segundo descendiente del primero

Todas las colecciones ofrecen una implementación de Iterator por medio del método:

**public Iterator iterator();**

Sus métodos son:

* **boolean hasNext();** // Indica si hay un elemento siguiente (y así evita la excepción)..
* **Object next();** // Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo:
  + NoSuchElementException (que deriva a su vez de RunTimeException)
* **void remove();** // Elimina el último elemento devuelto por next

Ejemplo:

***//****suponiendo que***colecciónString** *es una colección de textos*

**Iterator** it=colecciónString.iterator();

**while(**it.hasNext()**){**

**String** s=(**String**)it.next();***//puesto que next devuelve objetos***

***//Object, es necesario el casting***

**System**.out.println(s);

**}**

**for each PARA COLECCIONES**

Existe una versión de instrucción **for**, disponible desde la versión 1.5 de Java que permite usar de forma más fácil un bucle de recorrido por una colección de la misma forma que hace un objeto iterador.

Ejemplo:

***//****suponiendo que* **colecciónString***es una colección de textos*

**for(Object** o**:**colecciónString**){**

**String** s=(**String**)o;

**System**.out.println(s);

**}**

Hay que tener en cuenta que sólo es posible utilizar este bucle si deseamos avanzar por todos los elementos de una lista. Pero puesto que se trata de algo muy habitual, es un bucle muy utilizado.

**INTERFAZ Comparable**

Se dice que las clases que implementan esta interfaz cuentan con un “orden natural”.

Este orden es total, es decir, que siempre han de poder ordenarse dos objetos cualesquiera de la clase que implementa este interfaz. Todas las clases de Java tienen un orden natural (String, Double, Integer…..)

La interfaz **Comparable** contiene un método llamado compareTo(obj) , que toma sólo un argumento y se compara con otra instancia u objetos de la misma clase.

**public int compareTo(Object obj)**

Este método devuelve un entero *negativo*, *cero* o *positivo* según el argumento implícito (*this*) sea *anterior*, *igual* o *posterior* al objeto *obj*.

**INTERFAZ Comparator**

Si una clase ya tiene una ordenación natural y se desea realizar una ordenación diferente, por ejemplo descendente, dependiente de otros campos o simplemente requerimos varias formas de ordenar una clase, haremos que una clase distinta de la que va a ser ordenada implemente este interfaz.

La interfaz **Comparator** contiene un método llamado **compare(obj1,obj2)** que toma dos argumentos y compara el valor de dos objetos de las clases iguales o diferentes

**public int compare(Object o1, Object o2)**

El método ***compare()*** devuelve un entero negativo, cero o positivo según su primer argumento sea anterior, igual o posterior al segundo (Así asegura un orden ascendente).

**¿Cuándo usar cada uno?**

En una colección de objetos, éstos pueden ser ordenados por diferentes criterios. Dependiendo de la clase que realiza la comparación, se implementará la interfaz Comparable o Comparator.

El primero es cuando tengo que ordenar cosas simples, por ejemplo con una sola clave de ordenamiento, y el segundo cuando tal vez quiera ordenar por más de un campo.

Por ejemplo utilizaremos **Comparable**, si queremos ordenar una clase Persona que tenga los campos: DNI, Apellido, Nombre y la queremos ordenar solo por DNI, en este caso Persona implementará Comparable.

Si queremos ordenarlo por **cualquiera** de los campos, es decir, si se desea tener varios métodos de ordenación, puedes usar el **Comparator** implementado en una clase ajena a Persona.

**INTERFAZ Set**

El interfaz Set hereda del interfaz Collection. Pero no añade la definición de ningún método nuevo

**Representa colecciones que no permiten tener elementos duplicados**. Es el método **equals** el que se encarga de determinar si dos objetos son duplicados.

Las dos clases **HashSet** y **TreeSet** implementan la interfaz Set. Eso hace que estas dos colecciones sólo guarden objetos que no estén repetidos.

## Clase HashSet

Un HashSet es una estructura de datos que contiene un conjunto de objetos. Permite buscar un objeto dentro del conjunto de forma rápida y fácil. Internamente gestiona un array y guarda los objetos utilizando un índice calculado con un código hash del objeto. Por defecto el array tiene un tamaño de 16. Utiliza internamente una tabla de tipo hash que asocia claves a conjuntos de valores.

- Los elementos de un **HashSet no están ordenados**  
- Para añadir un elemento al HashSet se utiliza el método **add(Object obj)**.  
- Para borrar un elemento se utiliza **remove(Object obj)**.  
- Para borrar todos los elementos se utiliza **clear()**.  
- El tamaño del HashSet se puede obtener con la función **size()**

**Formato: nombre\_clase <tipo\_elemento> nombre\_coleccion**

**Ejemplo:**

**import java.util.\*;**

**public class TestHashSet**

**{**

**public static void main(String[] args)**

**{**

**HashSet <String>ciudades = new HashSet<String>();**

**ciudades.add("Madrid");**

**ciudades.add("Barcelona");**

**ciudades.add("Malaga");**

**ciudades.add("Barcelona");//repetido**

**ciudades.add("Vigo");**

**ciudades.add("Sevilla");**

**ciudades.add("Madrid"); // Repetido.**

**Iterator it = ciudades.iterator();**

**while(it.hasNext())**

**System.out.println("Ciudad: " + it.next());**

**System.out.println("la coleccion tiene "+ ciudades.size() + " elementos");**

**System.out.println("Ahora borro los elementos");**

**ciudades.clear();**

**System.out.println("la coleccion tiene "+ ciudades.size() + " elementos");**

**}**

**}**

**Salida:**

**Ciudad: Sevilla**

**Ciudad: Malaga**

**Ciudad: Barcelona**

**Ciudad: Madrid**

**Ciudad: Vigo**

**la coleccion tiene 5 elementos**

**Ahora borro los elementos**

**la coleccion tiene 0 elementos**

**Nota: Si el tipo de elemento es un Wrappers** la lista automáticamente se genera sin elementos repetidos.

## Clase TreeSet

**Un TreeSet mantiene los objetos ordenados** en lo que se conoce como un red-black tree, es decir, en un árbol binario balanceado (cada padre tiene como máximo 2 hijos, y cuando se inserta una entrada se autobalancea de forma que quede un árbol binario simétrico).

Un TreeSet permite hacer búsquedas rápidas. No tanto como un HashMap, pero el TreeSet tiene la ventaja de estar ordenado por clave.

Ejemplo:

import java.util.\*;

**public class TestTreeSet {**

public static void main(String[] args){

TreeSet<String> ciudades= new TreeSet<String>();

ciudades.add("Madrid");

ciudades.add("Barcelona");

ciudades.add("Malaga");

ciudades.add("Barcelona"); //Repetido

ciudades.add("Vigo");

ciudades.add("Sevilla");

ciudades.add("Madrid"); // Repetido.

Iterator it = ciudades.iterator();

while(it.hasNext())

System.out.println("Ciudad: " + it.next());

System.out.println("la coleccion tiene "+ ciudades.size() + " elementos");

}

}

Si contiene muchos elementos resulta muy costoso recorrerlos mediante un Iterator, no se garantiza el orden.

**Salida:**

**Ciudad: Barcelona**

**Ciudad: Madrid**

**Ciudad: Malaga**

**Ciudad: Sevilla**

**Ciudad: Vigo**

**la coleccion tiene 5 elementos**

**Nota: Al igual que con HashSet si el tipo de elemento es un Wrappers** la lista automáticamente se genera sin elementos repetidos.

Ejemplo: Introducir los elementos de una agenda en esos tipos de estructura

Creamos la clase Persona, para utilizar el TreeSet obligatoriamente tiene que implementar la interfaz Comparable y reescribir el método CompareTo, para saber porque campo tiene que ordenar en el momento de insertar en el árbol.

**public class Persona implements Comparable<Persona>{**

private String nombre;

private String telefono;

public Persona(String nombre,String telefono){

this.telefono=telefono;

this.nombre=nombre;

}

public void setNombre(String nombre){

this.nombre=nombre;

}

public void setTelefono(String telefono){

this.telefono=telefono;

}

public String getNombre(){

return nombre;

}

public String getTelefono(){

return telefono;

}

public int compareTo(Persona obj){

return this.nombre.compareTo(obj.nombre);

}

@Override

public String toString(){

return (nombre+" "+telefono);

}

}

Su ejecución :

import java.util.\*;

**public class TestPersonaCollection2 {**

public static void main(String[] args){

Scanner teclado=new Scanner(System.in);

Set <Persona> agenda = new TreeSet <Persona>(); // **A**

**// Set <Persona> agenda = new HashSet <Persona>(); // B**

String nb;

String telf;

System.out.println("introduce nombre o fin:");

nb=teclado.nextLine();

while(!nb.equals("fin"))

{

System.out.println("introduce telefono:");

telf=teclado.next();

Persona p = new Persona(nb,telf);

agenda.add(p);

teclado.nextLine();

System.out.println("introduce nombre o fin:");

nb=teclado.nextLine();

}

Iterator <Persona> it = agenda.iterator();

while(it.hasNext())

System.out.println("Nombre: " + it.next());

}

}

Para utilizar el HashSet, solo tendríamos que cambiar la línea **A** por la **B ,** además no hubiera hecho falta implementar Comparable en la clase Persona.

En el ejemplo anterior hemos “perdido” una de las características de estas colecciones la de implementar listas dinámicas **sin duplicados,** esto es debido a que no hemos utilizado un tipo **Wrappers** como tipo de los elementos**,** entonces tendremos que recurrir a implementar el método **equals** en las clases de los objetosigual que hicimos anteriormente con comparateTo(), pero no es el único requisito, de hecho el fundamental es que se defina también el método heredado **hashCode.**

La razón es que es el **hashCode** es el código pensado para este tipo de lista, de hecho es el identificador en una lista HashSet, por ello los objetos que consideremos iguales en contenido deben devolver el mismo hashCode, es decir el mismo número entero.

El método **equals** hay que definirlo para saber cuándo dos objetos de la lista son iguales.

Ponemos de nuevo la clase Persona con los métodos implementados:

public class Persona implements Comparable<Persona>{

private String nombre;

private String telefono;

public Persona(String nombre,String telefono){

this.telefono=telefono;

this.nombre=nombre;

}

public void setNombre(String nombre){

this.nombre=nombre;

}

public void setTelefono(String telefono){

this.telefono=telefono;

}

public String getNombre(){

return nombre;

}

public String getTelefono(){

return telefono;

}

@Override

public int compareTo(Persona obj){

return this.nombre.compareTo(obj.nombre);

}

@Override

public boolean equals(Object obj){

if(obj instanceof Persona){

Persona otroIgual=(Persona)obj;

return otroIgual.nombre.equals(nombre)&& otroIgual.telefono.equals(telefono);

}

else return false;

}

@Override

public int hashCode(){

return nombre.hashCode()+ telefono.hashCode();

}

@Override

public String toString(){

return (nombre+" "+telefono);

}

}

Si ejecutamos ahora **TestPersonaCollection2** veremos que las dos colecciones nos salen ordenadas

**INTERFAZ List**

El interfaz List hereda del interfaz Collection por lo que tiene disponible todos sus métodos, pero además aporta otros mucho más potentes. Representa colecciones con elementos en secuencia. Es decir, con orden.

Permite tener duplicados. Es accesible mediante índice, de manera que se puede:

* Acceder a un elemento concreto de una posición.
* Insertar un elemento en una posición concreta.

Los métodos que añade de este interfaz son:

**Acceso posicional**:

* Object **get**(int **indice**); // Obtiene el elemento almacenado en la colección en la posición que indica el índice.
* Object **set**(int **indice**, Object **elemento**); // **Sustituye** el elemento número índice por uno nuevo. Devuelve además el elemento antiguo ese elemento.
* void **add**(int **indice,** Object **elemento**); // **Añade** el elemento indicado en la posición índice de la lista
* Object **remove**(int **indice**); // Elimina el elemento cuya posición en la colección la da el parámetro índice
* boolean **addAll**(int **indice,** Collection **elemento**); // Añade todos los elementos de una colección a una posición dada.

**Búsqueda:**

* int **indexOf**(Object **elemento**); // Devuelve la posición del elemento. Si no lo encuentra, devuelve -1
* int **lastIndexOf**(Object **elemento**); // Devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1

**Subcolecciones:**

* List **subList**(int **desde**, int **hasta**); // Devuelve una lista con los elementos comprendidos entre ambas posiciones.

Las clases que implementan la interfaz List son: ArrayList, Vector y LinkedList.

Tanto **ArrayList** como **Vector** utilizan un objeto Array para almacenar los elementos internamente. Las colecciones de tipo Array, sin embargo, tienen un tamaño fijo que se determina de antemano al llamar al constructor. Esto implica que cuando sobrepasamos dicho tamaño hay que crear un nuevo Array y copiar el antiguo, lo que puede ser muy costoso computacionalmente. Por otro lado, si la capacidad máxima inicial del array fuera demasiado grande, estaríamos desperdiciando espacio.

**Por defecto ArrayList y Vector utilizan arrays con capacidad para 10 elementos.** **Cuando el número de elementos sobrepasa la capacidad disponible Vector dobla el tamaño del array interno, mientras que ArrayList utiliza la fórmula (capacidad \* 3) / 2 + 1.**

Para indicar valores que se ajusten lo máximo posible al uso que vamos a hacer de nuestra lista tanto Vector como ArrayList cuentan, además de con un constructor vacío y un constructor al que pasar una colección con los elementos con los que inicializar el vector o la lista, con un constructor al que pasar un entero con la capacidad inicial de la colección.

Vector cuenta también con un constructor extra mediante el que indicar el incremento en capacidad a utilizar cuando el número de elementos rebase la capacidad del vector, lo cuál puede ser muy interesante.

En cualquier caso, independientemente del constructor que utilicemos, al crear cualquier colección es una buena práctica utilizar el tipo más genérico posible para referirnos al nuevo objeto. Por ejemplo, sería preferible utilizar:

1. List<String> lista = new ArrayList<String>();

a

1. ArrayList<String> lista = new ArrayList<String>();

ya que al utilizar la primera versión, si es necesario en un futuro, podremos migrar fácilmente a una implementación distinta de List<E> gracias al polimorfismo.

**Clase Vector**

La clase **Vector** implementa la interfaz *List*. Es una clase veterana casi calcada a la clase *ArrayList*. En las primeras versiones de Java era la única posibilidad de implementar arrays dinámicos. Actualmente sólo se recomienda su uso si se utiliza una estructura dinámica para usar con varios *threads*.

Un vector es similar a un array, la diferencia estriba en que un vector crece automáticamente cuando alcanza la dimensión inicial máxima. Además, proporciona métodos adicionales para añadir, eliminar elementos, e insertar elementos entre otros dos existentes.

Mirar: <http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/colecciones/vector.htm>

import java.util.\*;

**public class TestVector{**

public static void main(String[] args) {

Vector<String> vectorStrings= new Vector<String>();

vectorStrings.add("Jorge");

vectorStrings.add("Ana");

vectorStrings.add("Luis");

vectorStrings.add("Pedro");

vectorStrings.add("Laura");

for(String str : vectorStrings){

System.out.println(str);

}

System.out.println("el primer elemnto del vector es:" +vectorStrings.firstElement());

System.out.println("el ultimo elemnto del vector es:" +vectorStrings.lastElement());

if(vectorStrings.contains("Luis")){

System.out.println("Luis se encuentra en el vector en la posición."

+vectorStrings.indexOf("Luis"));

}else

System.out.println("Luis NO se encuentra en el vector ");

System.out.println("Eliminamos a Pedro");

vectorStrings.remove("Pedro");

//insertamos un elemento en una posicion determinada, desplaza los que haya despues

vectorStrings.add(2,"Pepe");

//visualizar un elemento determinado

System.out.println("el elemento de la posicion 2 es: " + vectorStrings.get(2));

System.out.println("El nuevo vector");

for(String str : vectorStrings){

System.out.println(str);

}

//Cuantos elementos tenemos y cual es su capacidad, por defecto es 10

System.out.println("el vector tiene : "+ vectorStrings.size()+" elementos");

System.out.println("La capacidad del vector es:"+vectorStrings.capacity());

}

}

**Clase ArrayList**

Está pensada para crear tanto listas simples como doblemente enlazadas. Está disponible desde la versión 1.2 y es la clase fundamental para representar colecciones de datos

Es una estructura de datos de tipo Array dinámica. A diferencia de los arrays clásicos (arrays estáticos), un ArrayList permite aumentar el tamaño del vector indefinidamente (hasta lo que la memoria permita) y agregar o quitar elementos.

A diferencia de la LinkedList, la ArrayList permite acceder a cualquier elemento de la lista directamente mediante su índice, lo que la hace especialmente adecuada para búsquedas rápidas.

Posee tres constructores:

* **ArrayList().** Constructor por defecto. Simplemente crea un ArrayList vacío
* **ArrayList(int** capacidadInicial**).** Crea una lista con una capacidad inicial indicada.
* **ArrayList(Collection** c**).** Crea una lista a partir de los elementos de la colección indicada.

**Ejemplos:**

import java.util.\*;

**public class TestArrayList**

{

public static void main(String[] args)

{

List<String> ciudades = new ArrayList<String>();

ciudades.add("Madrid");

ciudades.add("Barcelona");

ciudades.add("Malaga");

ciudades.add("Vigo");

ciudades.add(1,"Sevilla");

ciudades.add("Madrid"); // Repetido.

Iterator it = ciudades.iterator();

while(it.hasNext())

{

String elemento=(String) it.next();

System.out.println("Ciudad: " + elemento);

}

}

}

import java.util.\*;

**public class ArrayListMain** {

public static void main(String[] args){

Collection<String> c=new ArrayList<String>(); //creo un objeto ArrayList y lo moldeo a for(int i=0; i<10;i++){ // una Collection

c.add(Integer.toString(i));

}

Iterator it =c.iterator();

while(it.hasNext()){

System.out.println(it.next());

}

}

}

**Ejemplo : Programa que pide el nombre del alumno y tres calificaciones para luego calcular su promedio. Se puede agregar cualquier cantidad de elementos a la lista.**

**import java.util.\*;**

**public class TestListaAlum {**

**static double media;**

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

**Scanner leer = new Scanner(System.in);**

**int op;**

**ArrayList<NodoLista> lista = new ArrayList <NodoLista> ();**

**do{**

**NodoLista nodo= new NodoLista();**

**System.out.println( "Ingrese el nombre del alumno:" );**

**nodo.nom = leer.next();**

**System.out.println( "Ingrese la primera calificación:" );**

**nodo.calif1 = leer.nextInt();**

**System.out.println( "Ingrese la segunda calificación:" );**

**nodo.calif2 = leer.nextInt();**

**System.out.println( "Ingrese la tercera calificación:" );**

**nodo.calif3 = leer.nextInt();**

**nodo.media= promedio(nodo.calif1, nodo.calif2, nodo.calif3);**

**lista.add(nodo);**

**System.out.println( "¿Desea ingresar otro alumno?" );**

**System.out.println( "1.-Si\t 2.-No" );**

**op = leer.nextInt();**

**} while(op != 2);**

**Iterator it = lista.iterator();**

**while (it.hasNext()){**

**System.out.println(it.next()+"");**

**}**

**}**

**private static double promedio(int calif1, int calif2, int calif3){**

**int suma = calif1 + calif2 + calif3;**

**media = suma/3;**

**return media;**

**}**

**}**

**La clase NodoLista**

**public class NodoLista {**

**String nom;**

**int calif1;**

**int calif2;**

**int calif3;**

**double media;**

**@Override**

**public String toString(){**

**return (nom+" "+calif1+" "+calif2+" "+calif3+" "+"media= "+ media);**

**}**

**}**

**Ejercicio: Llenar un arrayList con los siguientes datos:**

**private String nombre;**

**private String apellidos;**

**private String dni;**

**private Date fechaNacimiento;**

**a) visualizarlo sin ordenar**

**b) ordenado**

**c) ordenado por fecha de nacimiento**

**La clase Persona deberá implementar la interfaz Comparable para poder ordenar**

**import java.text.\*;**

**import java.util.\*;**

**public class Persona implements Comparable{**

**private String nombre;**

**private String apellidos;**

**private String dni;**

**private Date fechaNacimiento;**

**public Persona(String nombre, String apellidos, String dni, Date fechaNacimiento) {**

**this.nombre = nombre;**

**this.apellidos = apellidos;**

**this.dni = dni;**

**this.fechaNacimiento = fechaNacimiento;**

**}**

**public String getApellidos() {**

**return apellidos;**

**}**

**public void setApellidos(String apellidos) {**

**this.apellidos = apellidos;**

**}**

**public String getDni() {**

**return dni;**

**}**

**public void setDni(String dni) {**

**this.dni = dni;**

**}**

**public Date getFechaNacimiento() {**

**return fechaNacimiento;**

**}**

**public void setFechaNacimiento(Date fechaNacimiento) {**

**this.fechaNacimiento = fechaNacimiento;**

**}**

**public String getNombre() {**

**return nombre;**

**}**

**public void setNombre(String nombre) {**

**this.nombre = nombre;**

**}**

**@Override**

**public int compareTo(Object o) {**

**Persona persona = (Persona)o;**

**if(this.apellidos.compareToIgnoreCase(persona.apellidos) == 0) {**

**if(this.nombre.compareToIgnoreCase(persona.nombre) == 0) {**

**return this.dni.compareTo(persona.dni);**

**}**

**else {**

**return this.nombre.compareToIgnoreCase(persona.nombre);**

**}**

**} else {**

**return this.apellidos.compareToIgnoreCase(persona.apellidos);**

**}**

**}**

**@Override**

**public String toString() {**

**return this.apellidos + " " + this.nombre + " " + this.dni + " "**

**+ DateFormat.getDateInstance(DateFormat.DEFAULT,**

**Locale.getDefault()).format(this.fechaNacimiento);**

**}**

**}**

**Si queremos que la lista se ordene también por fecha de nacimiento, utilizaremos la interfaz Comparator, que la implementaremos en una nueva clase**

**import java.util.\*;**

**public class NacimientoPersonaComparator implements Comparator{**

**public int compare(Object o1, Object o2) {**

**Persona persona1 = (Persona)o1;**

**Persona persona2 = (Persona)o2;**

**return persona1.getFechaNacimiento().compareTo(persona2.getFechaNacimiento());**

**}**

**}**

**Nuestra ejecución será:**

**import java.util.\*;**

**public class AppListaPersona {**

**public static void pintaLista(List lista) {**

**for(int i=0;i<lista.size();i++) {**

**System.out.println(lista.get(i));**

**}**

**}**

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

**List<Persona> lista = new ArrayList();**

**Calendar cal = Calendar.getInstance();**

**cal.set(1976,3,21);**

**Persona persona1 =**

**new Persona("Francisco Javier","Martínez Páez","11111111A",cal.getTime());**

**cal.set(1973,8,12);**

**Persona persona2 =**

**new Persona("Roberto","Canales Mora","22222222B",cal.getTime());**

**cal.set(1975,7,23);**

**Persona persona3 =**

**new Persona("Alejandro","Pérez García","33333333C",cal.getTime());**

**cal.set(1980,6,14);**

**Persona persona4 =**

**new Persona("Germán","Jiménez Centeno","44444444D",cal.getTime());**

**cal.set(1981,5,8);**

**Persona persona5 =**

**new Persona("Pablo","Blanco Criado","55555555E",cal.getTime());**

**cal.set(1981,9,30);**

**Persona persona6 =**

**new Persona("Ana","Blanco Criado","66666666F",cal.getTime());**

**lista.add(persona1);**

**lista.add(persona2);**

**lista.add(persona3);**

**lista.add(persona4);**

**lista.add(persona5);**

**lista.add(persona6);**

**System.out.println("---------- SIN ORDENAR ------------");**

**pintaLista(lista);**

**//RECORRERLA CON ITERADOR**

**System.out.println(**

**"---------- ORDEN NATURAL DEFINIDO en compareTo ------------");**

**Collections.sort(lista);**

**pintaLista(lista);**

**System.out.println("---------- POR FECHAS DE NACIMIENTO ------------");**

**Collections.sort(lista, new NacimientoPersonaComparator());**

**pintaLista(lista);**

**}**

**}**

**Ejercicio: Modificar el ejemplo de la lista de alumnos de forma que se introduzcan las notas junto con el nombre en la lista y visualice la lista ordenada por nombre junto con las calificaciones y la media.**

**Solución:**

**package colecciones;**

**public class Alumno implements Comparable<Alumno>{**

**private String nom;**

**private int calif1;**

**private int calif2;**

**private int calif3;**

**public Alumno(){}**

**public Alumno(String nom, int calif1, int calif2, int calif3){**

**this.nom=nom;**

**this.calif1=calif1;**

**this.calif2=calif2;**

**this.calif3=calif3;**

**}**

**public void setNombre(String nombre){**

**nom=nombre;**

**}**

**public void setCalif1(int cali){**

**calif1=cali;**

**}**

**public void setCalif2(int cali){**

**calif2=cali;**

**}**

**public void setCalif3(int cali){**

**calif3=cali;**

**}**

**public String getNom() {**

**return nom;**

**}**

**public int getCalif1(){**

**return calif1;**

**}**

**public int getCalif2(){**

**return calif2;**

**}**

**public int getCalif3(){**

**return calif3;**

**}**

**@Override**

**public String toString(){**

**return ("Nombre:"+nom+" Calificaciones :"+calif1+" "+calif2+" "+calif3);**

**}**

**public int compareTo(Alumno obj){**

**return this.nom.compareTo(obj.nom);**

**}**

**}**

**package colecciones;**

**import java.util.\*;**

**import java.util.Scanner;**

**public class ApListaAlumno {**

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

**Scanner leer = new Scanner(System.in);**

**int op;**

**ArrayList<Alumno> lista = new ArrayList<Alumno>();**

**do{**

**Alumno nodo = new Alumno();**

**System.out.println("Ingrese el nombre del alumno:");**

**nodo.setNombre(leer.next());**

**System.out.println("Ingrese la primera calificacion:");**

**nodo.setCalif1(leer.nextInt());**

**System.out.println("Ingrese la segunda calificacion:");**

**nodo.setCalif2(leer.nextInt());**

**System.out.println("Ingrese la tercera calificacion:");**

**nodo.setCalif3(leer.nextInt());**

**lista.add(nodo);**

**System.out.println("Desea ingresar otro alumno?");**

**System.out.println("1.-Si\t 2.-No");**

**op = leer.nextInt();**

**}while(op!=2);**

**for (Object obj : lista) {**

**System.out.println(obj + " Media:" + promedio(obj));**

**}**

**//ordenamos la lista por nombre**

**Collections.sort(lista);**

**System.out.println("----LISTA ORDENADA-----");**

**for (Object obj : lista) {**

**System.out.println(obj + " Media:" + promedio(obj));**

**}**

**}**

**public static int promedio(Object obj){**

**int suma=(((Alumno)obj).getCalif1()+((Alumno)obj).getCalif2()+**

**((Alumno)obj).getCalif3());**

**return (suma/3);**

**}**

**}**

**Clase LinkedList**

Es una clase heredera de las anteriores e implementa métodos que permiten crear listas de adición tanto por delante como por detrás (**listas dobles**). Desde esta clase es sencillo implantar estructuras en forma de pila o de cola.

Añade los métodos:

|  |  |
| --- | --- |
| **Métodos** | **Funcionalidad** |
| Object **getFirst()** | Obtiene el primer elemento de la lista |
| Object **getLast()** | Obtiene el último elemento de la lista |
| void **addFirst(**Object o) | Añade el objeto al principio de la lista |
| void **addLast(**Object o) | Añade el objeto al final de la lista |
| void **removeFirst**() | Borra el primer elemento |
| void **removeLast**() | Borra el último elemento |

Los métodos están pensados para que las listas creadas mediante objetos **LinkedList** sirvan para añadir elementos por la cabezo o la cola, pero en ningún caso por el interior. Luego sirven principalmente para crear **pilas** y **colas**

Por ejemplo, si quisiésemos realizar una pila :

*public class Pila extends LinkedList {*

*public void push(Object elemento) {*

*this.addFirst(elemento);*

*}*

*public Object pop() {*

*return this.removeFirst();*

*}*

*}*

Como ejercicio podéis intentar crear una cola, es igual de sencillo.

**INTERFAZ Map**

Es la raíz de todas las clases capaces de implementar mapas. Representa colecciones con parejas de elementos: clave y valor.

**No permite tener claves duplicadas**. Pero si valores duplicados.

Esta interfaz no deriva de Collection por lo que no usa iteradores ni ninguno de los métodos vistos anteriormente. La razón es que la obtención, búsqueda y borrado de elementos se hace de manera muy distinta. Los mapas no permiten insertar objetos nulos (provocan excepciones de tipo NullPointerException).

Los métodos que define esta interfaz son:

**Operaciones Básicas**

* **Object put(Object clave, Object valor);** // Coloca el par clave-valor en el mapa (asociando la clave a dicho valor). Si la clave ya existiera, sobrescribe el anterior valor y devuelve el objeto antiguo. Si esa clave no aparecía en la lista, devuelve **null** .
* **Object get(Object clave);** // Devuelve el objeto que posee la clave indicada.
* **Object remove(Object clave);** // Elimina de la lista el valor asociado a esa clave. Devuelve el valor que tuviera asociado esa clave o **null** si esa clave no existe en el mapa
* **boolean containsKey(Object clave);** // Comprueba la existencia de una clave.
* **boolean containsValue(Object valor);** // Comprueba la existencia de un valor
* **int size();** // Devuelve el número de pares clave-valor del mapa
* **boolean isEmpty();** // Si no contiene ninguna pareja.

**Operaciones masivas**

* **void putAll(Map t); //** Añade todo el mapa indicado, al mapa actual**.**
* **void clear(); //** Elimina todas las parejas**.**

**Obtención de colecciones**

* **public Set keySet();** // Devuelve las claves en un Set.
* **public Collection values()**; // Obtiene la colección de valores del mapa, permite utilizar el HashMap como si fuera una lista normal al estilo de la clase Collection (por lo tanto se permite recorrer cada elemento de la lista con un iterador)

Las operaciones fundamentales son **get**, **put** y **remove**. El conjunto de claves no puede repetir la clave. Hay que tener en cuenta que las claves se almacenan en una tabla hash (es decir es una estructura de tipo **Set**) por lo que para detectar si una clave está repetida, la clase a la que pertenecen las claves del mapa deben definir (si no lo está ya) adecuadamentelos métodos **hashCode** y **equals**.

**INTERFAZ Map.Entry<k,v>**

La interfaz **Map.Entry** se define de forma interna a la interfaz **Map** y representa un objeto de par clave/valor. Es decir mediante esta interfaz podemos trabajar con una entrada del mapa. Tiene estos métodos:

* **K getKey();** // Obtiene la clave Del elemento actual Map.Entry
* **V getValue();** // Obtiene el valor
* **V setValue (V valor)**; // Cambia el valor
* **boolean equals(Obj obj**); // Devuelve verdadero si el objeto es un Map.Entry cuyos pares clave-valor son iguales que los del Map.Entry actual

**Clase HashMap**

Esta clase implementa la interfaz Map utilizando una tabla hash. Esta implementación permite el valor null tanto como clave o valor. Al igual que en un HashSet (vimos que utilizaba una HashMap ) esta clase no garantiza el orden de los elementos.

Es la clase más utilizada para implementar mapas

Ejemplo:

import java.util.\*;

public class TestHashMap {

public static void main(String[] args){

HashMap codigos = new HashMap();

codigos.put("01","Urgente");

codigos.put("02","Importante");

codigos.put("03","Normal");

codigos.put("04","Baja prioridad");

System.out.println(codigos);

System.out.println("Aleatorio 03: " + codigos.get("03"));

System.out.println("Aleatorio 01: " + codigos.get("01"));

Set s = codigos.keySet(); //Paso claves a Set para poder utilizar iterator

Iterator it = s.iterator();

while(it.hasNext())

{

String aux = (String)it.next();

System.out.println(aux + ": " + codigos.get(aux));

}

Map sortedMap = new TreeMap(codigos);

System.out.println(sortedMap);

}

}

Salida:

{04=Baja prioridad, 01=Urgente, 02=Importante, 03=Normal}

Aleatorio 03: Normal

Aleatorio 01: Urgente

04: Baja prioridad

01: Urgente

02: Importante

03: Normal

{01=Urgente, 02=Importante, 03=Normal, 04=Baja prioridad}

**Clase TreeMap**

Se trata de una estructura de tipo árbol binario, que permite que los elementos del mapa se ordenen en sentido ascendente según la clave. En ese sentido es una clase muy parecida a **TreeSet**.

**TreeMap** implementa la interfaz **SortedMap** que, a su vez, es heredera de **Map**, por lo que todo lo dicho sobre los mapas funciona con las colecciones de tipo **TreeMap**.

Lo que aportan de nuevo es que los datos en el mapa se ordenan según la clave. Sólo eso y para ordenarlos (al igual que ocurre con la clase **TreeSet**) la clase de las claves tiene que implementar la interfaz **Comparable** o bien durante la creación del **TreeMap** indicar un objeto **Comparator.**

La salida del ejercicio anterior utilizando TreeMap es la siguiente:

{01=Urgente, 02=Importante, 03=Normal, 04=Baja prioridad}

Aleatorio 03: Normal

Aleatorio 01: Urgente

01: Urgente

02: Importante

03: Normal

04: Baja prioridad

{01=Urgente, 02=Importante, 03=Normal, 04=Baja prioridad}

Que como vemos resulta ordenada por la clave.

/\*Vamos a crear una agenda telefónica en la que vamos a vincular al nombre del contacto, el numero de teléfono. Como veis, la estructura es clara, el nombre será la clave, mientras que el numero de teléfono será el valor. \*/

package colecciones2;

import java.util.\*;

public class AgendaTelefonica

{

// la agenda se guarda en un mapa

private Map<String,Integer> agenda;

// Constructor que deja la agenda vacia

public AgendaTelefonica(){

agenda = new HashMap<String,Integer>();

}

// Añadir un nombre con su telefono

public void anadeTelefono(String nombre, int telefono){

agenda.put(nombre,new Integer(telefono));

}

//Consultar un nombre; retorna el telefono o 0 si el nombre no existe

public int consulta (String nombre) {

Integer tel= agenda.get(nombre);

if (tel==null) {

return 0;

} else {

return tel.intValue();

}

}

//Saber si un nombre está en el diccionario

public boolean estaIncluido(String nombre) {

return agenda.containsKey(nombre);

}

//Mostrar la lista de toda la agenda

public void mostrarNumeros() {

Set<Map.Entry<String,Integer>>lista=agenda.entrySet(); //paso a Set

System.out.println();

System.out.println("Nombre - Telefono:");

for (Map.Entry<String,Integer> e:lista) {

System.out.println(e.getKey()+" - " +e.getValue());

}

}

}

/ \* Programa de prueba de la agenda telefonica \*/

package colecciones2;

public class PruebaAgenda

{

public static void main (String[] args) {

AgendaTelefonica agenda = new AgendaTelefonica();

agenda.anadeTelefono("a",1);

agenda.anadeTelefono("b",2);

agenda.anadeTelefono("c",3);

agenda.anadeTelefono("d",4);

agenda.anadeTelefono("e",5);

agenda.anadeTelefono("f",6);

agenda.anadeTelefono("g",7);

// cambiamos un numero

agenda.anadeTelefono("d",44);

System.out.println("Consulta a: "+agenda.consulta("a"));

System.out.println("Consulta d: "+agenda.consulta("d"));

System.out.println("Consulta g: "+agenda.consulta("g"));

System.out.println("Consulta x: "+agenda.consulta("x"));

System.out.println("Consulta y: "+agenda.consulta("y"));

System.out.println("Esta a: "+agenda.estaIncluido("a"));

System.out.println("Esta x: "+agenda.estaIncluido("x"));

System.out.println("Esta d: "+agenda.estaIncluido("d"));

agenda.mostrarNumeros();

}

}