Tecnicatura
Superior en Programación

LABORATORIO III

UNIDAD IV:
COLECCIONES EN JAVA.

Índice

Colecciones de Java

1. Introducción

En este tema vamos a estudiar un conjunto de clases e interfaces de Java que forman el Java Collections Framework y que constituyen un conjunto de herramientas que aumentan las capacidades del lenguaje a la hora de trabajar con estructuras de datos.
El interrogante es que diferencia existe entre las arreglos que hemos estudiado y las variantes de colecciones que se presentan en este apartado?

La diferencia fundamental es que es los arreglos en Java pueden almacenar tipos de datos primitivos como un int, short o un byte; además también podemos almacenar referencias a objetos como por ejemplo: si queremos almacenar información de los alumnos para su posterior procesamiento. En el caso de las colecciones "NO" se pueden almacenar datos de tipos primitivos, es decir solo se almacena referencias de la Clase Object. Todas las clases que hemos implementado en la ejercitación desarrollada en el laboratorio, heredan características y comportamiento de la misma.

En la siguiente imagen se muestra la jerarquía de interfaces (y clases que las implementan en cursiva) del JCF.

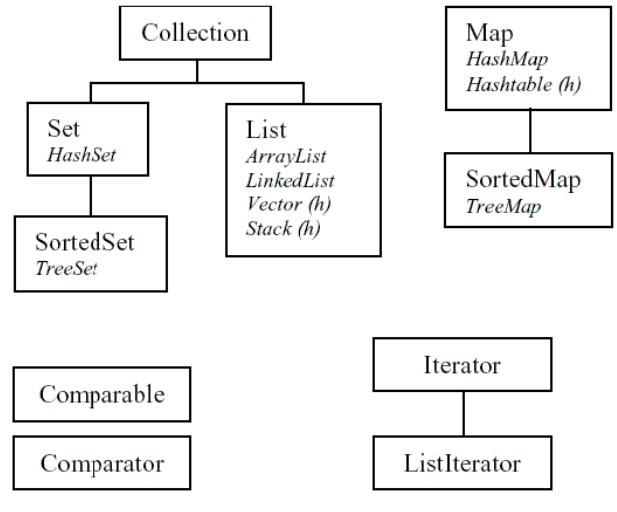

Figura 1. Jerarquía de colecciones en Java.

En el diseño de la JCF las interfaces son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interfaz se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases ArrayList y LinkedList disponen exactamente de los mismos métodos y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que ArrayList almacena los objetos en un array, la clase LinkedList los almacena en una lista enlazada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

Interfaces de la JCF: Constituyen el elemento central de la JCF.

- Collection: define métodos para tratar una colección genérica de elementos.
- Set: colección que no admite elementos repetidos
- SortedSet: set cuyos elementos se mantienen ordenados según el criterio establecido.
- List: admite elementos repetidos y mantiene un orden inicial
- Map: conjunto de pares clave/valor, sin repetición de claves
- SortedMap: map cuyos elementos se mantienen ordenados según el criterio establecido

Interfaces de soporte:

- **Iterator**: Dispone de métodos para recorrer una colección y para borrar elementos.
- **ListIterator**: deriva de Iterator y permite recorrer lists en ambos sentidos.
- **Comparable**: declara el método compareTo(). La clase que implemente dicho interfaz definirá el llamado orden natural para sus objetos el cual será utilizado en las estructuras de datos.
- **Comparator**: declara el método compare(). Proporciona una forma más flexible de tener distintas formas de comparar objetos pero es más costoso en tiempo de codificación. Podemos crear distintas clases que implementen este interfaz y tener distintos métodos compare() para una clase. Después, a la hora de crear la estructura de datos, se le dice que comparador queremos utilizar.

Clases de propósito general: Son las implementaciones de las interfaces de la JFC.

- **TreeSet**: Interfaz SortedSet implementada mediante un árbol binario ordenado.
- **ArrayList**: Interfaz List implementada mediante un array.

- **LinkedList**: Interfaz List implementada mediante una lista vinculada.
- **HashMap**: Interfaz Map implementada mediante una hash table.
- **TreeMap**: Interfaz SortedMap implementada mediante un árbol binario.
- **HashSet**: Interfaz Set implementada mediante una hash table.

Nota: Una Tabla Hash es un tipo de colección formado por pares de clave y valor, que se almacenan dentro de una tabla. La clave se utiliza para obtener el índice de ubicación del valor dentro de la tabla, utilizando para ello una función denominada función hash.

Así pues, una estructura hash está formada por tres elementos básicos:
1) una tabla o array de tamaño N donde se almacenan los elementos.
2) una función hash que, dada una clave, nos da el índice de la tabla donde se encuentra almacenado el dato asociado a esa clave.
3) una función de resolución de colisiones, que determina qué hacer cuando para dos claves distintas se obtiene el mismo índice en la tabla.

2. Interfaz Collection

La interfaz Collection es implementada por los conjuntos (sets) y las listas (lists). Esta interfaz declara una serie de métodos generales utilizables con **Sets** y **Lists**.
Para los ejemplos vamos a utilizar dicho interfaz con la clase HashSet. Esta clase representa a un conjunto de objetos no repetidos, en el cual no hay ningún orden establecido.
Esta clase tiene los siguientes tres constructores:

HashSet(): Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).
HashSet(Collection c): Constructs a new set containing the elements in the specified collection.
HashSet(int initialCapacity): Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and default load factor, which is 0.75.
HashSet(int initialCapacity,float loadFactor): Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.

Vamos a crear nuestro conjunto de elementos para utilizar en los ejemplos:

```
Collection co = new HashSet();
```

```
Boolean bEnvoltura=new Boolean("TrUe");

Character cEnvoltura=new Character('a');

Integer iEnvoltura=new Integer("5");

Double dEnvoltura=new Double("89.8");

Double dEnvoltura2=dEnvoltura;

public interfaz java.util.Collection

{

public abstract boolean add(Object); // opcional

public abstract boolean addAll(java.util.Collection); // opcional

public abstract void clear(); // opcional

public abstract boolean contains(Object);

public abstract boolean containsAll(java.util.Collection);

public abstract boolean equals(Object);

public abstract int hashCode();

public abstract boolean isEmpty();

public abstract java.util.Iterator iterator();

public abstract boolean remove(Object); // opcional

public abstract boolean removeAll(java.util.Collection); // opcional

public abstract boolean retainAll(java.util.Collection); // opcional

public abstract int size();

public abstract Object toArray()[];

public abstract Object toArray(Object[])[];

}
```

Los métodos indicados como "// opcional" pueden no estar disponibles en algunas implementaciones, como por ejemplo en las clases que no permiten modificar sus objetos. Por supuesto dichos métodos deben ser definidos, pero lo que hacen al ser llamados es lanzar una UnsupportedOperationException.

Métodos para agregar y eliminar elementos.

- boolean add(Object element): Añade un elemento a la colección, devolviendo true si fue posible añadirlo y false en caso contrario (por ejemplo, un set que ya contenga el elemento).
- boolean remove(Object element): Elimina un único elemento (si lo encuentra), y devuelve true si la colección ha sido modificada.

Métodos para realizar consultas.

- int size(): Devuelve el número de elementos disponibles.
- boolean isEmpty(): Devuelve true si la colección es vacía.

Muy útil a la hora de recorrer la estructura.

- boolean contains(Object element). Devuelve true si el elemento pertenece a la colección.

Métodos para recorrer todos los elementos.

- Iterator iterator(): Devuelve una referencia Iterator que permite recorrer una colección con los métodos next() y hasNext(). Permite también borrar el elemento actual con remove().

```
Ejemplo:
Iterator it=co.iterator();
```

Más adelante estudiaremos el interfaz ITERATOR.

Métodos para realizar varias operaciones simultáneamente.

- boolean containsAll(Collection collection): Igual que contains(), pero con un conjunto de elementos.
- boolean addAll(Collection collection): Igual a add(), pero añade un conjunto de datos si es posible.
- void clear(): Elimina todos los elementos.
- void removeAll(Collection collection): Igual que remove(), pero elimina el conjunto de elementos que se pasa como parámetro.
- void retainAll(Collection collection): Elimina todos los elementos menos los especificados por la colección pasada como parámetro.

Otros Métodos

- boolean **equals**(Object): Implementa la igualdad o equivalencia. Retorna true si el objeto que llama al método es equivalente al que se pasa como parámetro: son los dos un conjunto, tienen el mismo tamaño y contienen los mismos elementos.
- int **hashCode**():A la hora de acceder, añadir, o eliminar un objeto contenido en un hashtable, la implementación de dicha estructura invocará este método sobre el objeto para obtener un int que pueda ser utilizado en la elaboración del índice en el hashtable. Durante la ejecución de un programa este método ha de retornar el mismo int siempre que sea invocado sobre el mismo objeto. Siempre y cuando sea factible ha de ser único para cada objeto. Por ello, aunque esta implementación no es requerida, el método devuelve la dirección física del objeto en memoria.
- Object **toArray**(): Permiten convertir una colección en un array.

3. Interfaz Comparable y Comparador

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 del la clase que implementa este interfaz. La interfaz Comparable declara el método compareTo() de la siguiente forma:

public int compareTo(Object obj)

Que compara su argumento implícito con el que se le pasa por ventana. Este método devuelve un entero negativo, cero o positivo según el argumento implícito (this) sea anterior, igual o posterior al objeto obj.
Las estructuras cuyos objetos tengan implementado el interfaz Comparable podrán ejecutar ciertos métodos basados en el orden tales como: sort o binarySearch.
Si queremos programar el método compareTo debemos hacerlo con cuidado: ha de ser coherente con el método equals y ha de cumplir la propiedad transitiva.

4. 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.
Su principal método se declara en la forma:
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 oren ascendente).
Es muy importante que compare() sea compatible con el método equals() de los objetos que hay que mantener ordenados. Su implementación debe cumplir unas condiciones similares a las de compareTo().

Los objetos que implementa este interfaz pueden ser utilizados en las siguientes situaciones (especificando un orden distinto al natural):

- Como argumento a un constructor TreeSet o TreeMap (con la idea de que las mantengan ordenadas de acuerdo con dicho Comparator)
- Collections.sort(List, Comparator) , Arrays.sort(Object[], Comparator)
- Collections.binarySearch(List, Object, Comparator),
- Arrays.binarySearch(Object [] v, Object key, Comparator c) (Object también ha de implementarlo).

5. Interfaz Iterator

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. Veamos con más detalle cada uno de ellos.
☐ Interfaz Iterator.

```
public interfaz java.util.Iterator
{
public abstract boolean hasNext();
```

```
public abstract Object next();

public abstract void remove();}
```

- hasNext(): devuelve un cierto si el elemento actual tiene siguiente en la colección, es decir, mientras existan elementos no tratados mediante el método next.
- next(): devuelve una referencia al siguiente elemento en la colección, es decir, coloca el iterador en el elemento siguiente y lo devuelve (Object). Es el método utilizado para acceder a los elementos de una colección. Lanza NoSuchElementException si se invoca un número de veces superior al número de elementos existentes en la colección.
- remove(): es un método opcional. Elimina de la colección el último elemento retornado por next. Solo puede ser llamado una vez por cada llamada a next(), y siempre después de aquel. Es la única forma segura de eliminar un elemento mientras se está recorriendo una colección. Eleva IllegalEstateException si no se cumplen las condiciones expuestas para la llamada; y UnsupportedOperationException si la implementación de este interfaz no incluyó este método (ya que es opcional).

Como vimos en apartados anteriores, la interfaz Collection dispone de un método denominado iterator(), que devuelve un Iterator situado antes del primer elemento de la colección, (es decir, si it es un iterador it.next() devolverá el primer elemento). Este método iterator() es utilizado para inicializar los iteradores antes de comenzar el recorrido de la colección. Así, el recorrido básico de una colección es el siguiente:

```
iterator it = colección.iterator();

while (it.hasNext())

{

Object obj = it.next();

tratar obj

}
```

Mientras el iterador sea útil, es decir, mientras estemos recorriendo con él una colección, no podremos modificar/añadir/eliminar objetos de la collección salvo con el método remove del mismo. Si lo hicieramos, se elevaría la excepción: ConcurrentModificationException.

6. Interfaz List

6.1 Introducción.

Podemos definir una lista como un conjunto de elementos los cuales ocupan una determinada posición de tal forma que cada elemento tendrá un único sucesor y un único antecesor, menos el primero y el último.

Ejemplo: la cola del cine, o del autobús.

El tamaño de estas estructuras es dinámico y va cambiando a lo largo del programa: se añaden elementos al final de la lista, se insertan en una determinada posición, se eliminan elementos, etc..

Las operaciones generales que se llevan a cabo en las listas son las siguientes:

- Insertar, eliminar o localizar un elemento.
- Determinar su tamaño: número de sus elementos.
- Recorrer la lista en ambos sentidos: para localizar un elemento.
- Clasificar sus elementos: en orden ascendente o descendente.
- Unir: dos o más listas en una sola.
- Dividir: una lista en varias sublistas.
- Copiar, la lista.
- Borrar, la lista.

Para utilizar una estructura de este tipo Java dispone de un interfaz llamado LIST, que deriva de COLLECTION. Este interfaz está ya implementado en Java de distintas formas. Escogeremos la que más se adecue a nuestras necesidades, a saber:

Como array de elementos:

- Acceso directo a los elementos.
- Necesidad de desplazamiento de información en borrado e inserción.
- Problemas de tamaño de las listas.

Como array de elementos y array de índices (enlaces):

- Problemas de acceso directo.
- No hay necesidad de desplazamiento en inserción y borrado.
- Problema de tamaño de las listas.

Como lista (dinámica) de nodos enlazados:

- Problemas de acceso directo.

- No hay necesidad de desplazamiento en inserción y borrado.
- No hay problemas de tamaño de las listas.

6.2 ArrayList

Sabido es que dos clases que implementan la misma interfaz se pueden utilizar de similar forma. Las clases **ArrayList** y **LinkedList** disponen de casi los mismos métodos y ambas implementan a nuestra interfaz. Pero mientras que ArrayList almacena los objetos en una tabla, la clase LinkedList los almacena en una lista enlazada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda lo será para borrar e insertar elementos.
Hay que observar que los elementos que van a contener nuestras listas han de ser objetos, por tanto pueden ser String, pero no los tipos básicos (int, float,..). Por ello para usar una lista de enteros, por ejemplo, hay que usar las clases envoltorio (wrappers) o construir una.
Cada instancia de ArrayList tiene una capacidad. Dicha capacidad es el tamaño de la tabla que se usa para almacenar los elementos en la lista. Es siempre, al menos tan grande, como el tamaño de la lista. Al añadir elementos a la tabla, su capacidad crece automáticamente.

Constructores
ArrayList()
Construye una lista vacía.
ArrayList(collection c)
Construye una lista que contiene los elementos de la colección, en el orden en que son devueltos por el iterator de la colección.
ArrayList(int capacidadInicial)
Construye una lista vacía con la capacidad inicial dada.

Métodos más usuales

- boolean add(Object o): Añade un elemento al final de la lista.
- void add(int in, Object o): Inserta el elemento dado en el lugar especificado. (LinkedList). Si el índice es mayor que el tamaño de la lista se eleva la excepción IndexOutOfBoundsException.
- boolean addAll(Collection c): Añade todos los elementos de la colección al final de la lista, en el orden en que son devueltos por el iterador de la colección. (LinkedList)
- boolean addAll(int in,Collection c): Inserta todos los elementos de la colección al final de la lista, comenzando por la posición dada.(LinkedList)

- void clear(): Elimina todos los elementos de la lista. (LinkedList) o Object clone(). Añade Devuelve una copia superficial (shallow) de la lista. (LinkedList) o boolean contains(Object o). Devuelve true si la lista contiene el elemento. (LinkedList)
- void ensureCapacity (int capacidadMi): Aumenta la capacidad de la instancia de ArrayList antes de una inserción masiva de elementos para reducir la cantidad de veces que Java tiene que modificar el tamaño del array que contiene los elementos (la redistribución de memoria es tareacostosa). El parámetro que recibe hace referencia al número de elementos que aumentaría la lista.
- Object get(int in): Devuelve el elemento de la lista que ocupa el lugar especificado. (LinkedList)
- int indexOf(Object o): Devuelve –1 si el elemento no pertenece a la lista, y el lugar de la primera aparición si el elemento está en la lista.(LinkedList)
- Boolean isEmpty(): Devuelve true si la lista está vacía. (LinkedList)
- int lastIndexOf(Object o):Devuelve –1 si el elemento no pertenece a la lista, y el lugar de la última aparición si el elemento está en la lista.(LinkedList)
- Object remove(int in): Elimina el elemento de la posición dada.(LinkedList)
- Object set(int in, Object o): Reemplaza el elemento en la posición dada por el nuevo elemento. (LinkedList)
- int size(): Devuelve el número de elementos disponibles. (LinkedList)
- Object [] toArray(): Devuelve una tabla conteniendo todos los elementos de la lista en el mismo orden. (LinkedList)
- void trimToSize(): Poda la capacidad de esa instancia para ajustarla al tamaño de la lista real.

6.3 Clase Vector

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.

Cuando creamos un vector u objeto de la clase Vector, podemos especificar su dimensión inicial, y cuanto crecerá si rebasamos dicha dimensión.

```
Vector vector=new Vector(20, 5);
```

Tenemos un vector con una dimensión inicial de 20 elementos. Si rebasamos dicha dimensión y guardamos 21 elementos la dimensión del vector crece a 25.

Al segundo constructor, solamente se le pasa la dimensión inicial.

```
Vector vector=new Vector(20);
```

Si se rebasa la dimensión inicial guardando 21 elementos, la dimensión del vector se duplica. El programador ha de tener cuidado con este constructor, ya que si se pretende guardar un número grande de elementos se tiene que especificar el incremento de la capacidad del vector, si no se quiere desperdiciar inútilmente la memoria el ordenador.

Con el tercer constructor, se crea un vector cuya dimensión inicial es 10.

```
Vector vector=new Vector();
```

La dimensión del vector se duplica si se rebasa la dimensión inicial, por ejemplo, cuando se pretende guardar once elementos.

Hay dos formas de añadir elementos a un vector. Podemos añadir un elemento a continuación del último elemento del vector, mediante la función miembro *addElement*.

```
v.addElement("uno");
```

Podemos también insertar un elemento en una determinada posición, mediante *insertElementAt*. El segundo parámetro o índice, indica el lugar que ocupará el nuevo objeto. Si tratamos de insertar un elemento en una posición que no existe todavía obtenemos una excepción del tipo *ArrayIndexOutOfBounds*. Por ejemplo, si tratamos de insertar un elemento en la posición 9 cuando el vector solamente tiene cinco elementos. Para insertar el string "tres" en la tercera posición del vector *v*, escribimos

```
v.insertElementAt("tres", 2);
```

El acceso a los elementos de un vector no es tan sencillo como el acceso a los elementos de un array. En vez de dar un índice, usamos la función miembro *elementAt*. Por ejemplo, *v.elementAt(4)* sería equivalente a *v[4]*, si *v* fuese un array.
Para acceder a todos lo elementos del vector, escribimos un código semejante al empleado para acceder a todos los elementos de un array.

```
for(int i=0; i<v.size(); i++){
    System.out.print(v.elementAt(i)+"\t");
}
```

7. Interfaz Set

El interfaz SET representa a un tipo de conjunto en el que los elementos no están repetidos. Esta interfaz hereda de Collection y no añade ningún método nuevo a los ya conocidos.
Vimos, al estudiar el interfaz Collection, como se comportaba una colección de tipo HashSet.
Por lo que pasaremos a explicar el interfaz SortedSet, implementado por la clase TreeSet.

8. Interface SortedSet

Un conjunto ordenado es una colección de objetos ordenados según su orden natural o algún criterio distinto de ordenación. En un conjunto ordenado no hay elementos repetidos. Para que los objetos que forman parte de un conjunto se puedan ordenar, un requisito que deben cumplir estos objetos es que sean comparables entre ellos. Esto se traduce en que la clase a la que pertenecen estos objetos debe implementar la interfaz Comparable y por consiguiente los métodos:

public boolean equals(Object o);
public int compareTo(Object o);

Si se quiere que los elementos estén ordenados según algún criterio distinto al orden natural de los elementos habría que crear una clase donde se definan estos criterios que implemente la interfaz Comparator y por consiguiente los métodos:

public boolean equals(Object o);
public int compare(Object o1, Object o2);

La clase **TreeSet** implementa la interfaz SortedSet. Esta clase garantiza que los elementos del conjunto estén ordenados en orden ascendente según el orden natural de los elementos o un criterio de ordenación distinto al natural indicado a través de un Comparator. Esta clase dispone de los siguientes constructores:

TreeSet(): Construye un nuevo conjunto vacío ordenado según el orden natural de los elementos.
TreeSet(Collection c): Construye un nuevo conjunto que contiene los elementos de la colección pasada como parámetro, y ordenado según el orden natural de los elementos.
TreeSet(Comparator c): Construye un nuevo conjunto vacío ordenado según el comparator que se pasa como parámetro.
TreeSet(SortedSet s): Construye un nuevo conjunto que contiene los mismos elementos del conjunto ordenado que se pasa como parámetro, y ordenado según el mismo orden.

La interfaz SortedSet deriva de la interfaz Set y proporciona los métodos siguientes:

```
public interface SortedSet extends Set {
SortedSet subSet (Object fromElemento, Object toElemento);
SortedSet headSet (Object toElemento);
SortedSet tailSet (Object fromElemento);
Object first ();
Object last ();
Comparator comparator (); }
```

9. Ejercitación Propuesta

En la parte práctica nos enfocaremos en las colecciones Arraylist y Vector.

9.1 Cree un programa que permite almacenar correos electrónicos en el orden en que se reciben.

Esta entidad posee los siguientes datos: origen(String), destino(String), mensaje(String), cantidad de palabras(int), prioridad(int).
Se pide desarrollar métodos que faciliten la inserción de mail, borrado, mostrar los mails recibidos, la cantidad de palabras que tiene un determinado mail y la cantidad total de palabras de todos los mails recibidos.(ArrayList)

```
Resolución.
//CorreoElectronico.java
public class CorreoElectronico
{ private String origen;
  private String destino;
  private String mensaje;
  private int cantPalabras;
  private int prioridad; //(0-Alta 1-Normal)
    public CorreoElectronico(String o, String d, String m,int c)
  { origen=o;
       destino=d;
       mensaje=m;
       cantPalabras=c;
  }
```

```
  public String getOrigen()
  { return origen;}

  public String getDestino()
  { return destino;}

  public String getMensaje()
  { return mensaje;}

  public int getCantPalabras()
  { return cantPalabras;}
  public int getPrioridad()
  { return prioridad;}

  public void setPrioridad(int p)
  { prioridad=p;}

  public String toString()
  { return origen+" " + destino + " "+ mensaje + " "+cantPalabras+" "+
prioridad;}  }


import java.util.*;
public class GestorCorreo
{ private ArrayList listacorreo;
  public GestorCorreo()
  { listacorreo=new ArrayList();}

  public void AgregarCorreo(CorreoElectronico mail)
  { listacorreo.add(mail);}

  public void BorrarMensaje(int indice)
  { listacorreo.remove(indice);}

  public String MostrarMails()
  { Iterator itr=listacorreo.iterator();
    CorreoElectronico ce;
```

```
  String aux="Listado de mails....\n";
  String aux1="";
  while(itr.hasNext())
  { //se toma el mail.
    ce=(CorreoElectronico)itr.next();
    //System.out.println(listacorreo.indexOf(ce));
    aux1+=ce.getOrigen()+ "**" + ce.getDestino() + "**" +ce.getMensaje()+ "\n";

  }
  return aux+aux1;
}

public int MostrarCantPalabras(int indice)
{ Iterator itr=listacorreo.iterator();
     CorreoElectronico ce;
     int cantP=0;

     while(itr.hasNext())
     { //se toma el mail
       ce=(CorreoElectronico)itr.next();
       if(indice==listacorreo.indexOf(ce))
         {  cantP=ce.getCantPalabras();
            break;
         }
     }
     if(cantP!=0)
       return cantP;
     else
       return -1;
}

public int TotalPalabras()
{ Iterator itr=listacorreo.iterator();
     CorreoElectronico ce;
     int cantT=0;

     while(itr.hasNext())
     { //se toma el mail
```

```
        ce=(CorreoElectronico)itr.next();
        cantT+=ce.getCantPalabras();
      }
      return cantT;
  }

  //Mostrar la cantidad de mails con prioridad alta.
}

public class Aplicacion
{ public static void main(String args[])
  { GestorCorreo gc=new GestorCorreo();
    gc.AgregarCorreo(new CorreoElectronico("cynthia","claudio","xxx",3));
      gc.AgregarCorreo(new CorreoElectronico("juan","maria","yyya",4));
      gc.AgregarCorreo(new CorreoElectronico("rosa","hector","hello!",5));

      System.out.println(gc.MostrarMails());
      int valor=gc.MostrarCantPalabras(1);//Muestra cant. palabras del mensaje1
      System.out.println("La cant. de caracteres del mensaje 1 es:"+valor);

      System.out.println("La cantidad de palabras de los mails
son:"+gc.TotalPalabras());
  }
 }
```

9.2 El departamento de recursos humanos de una empresa de nuestra ciudad, nos ha solicitado la implementación de un sistema que facilite la gestión de recursos humanos de la misma.
Para ello se han detectado las siguientes entidades:
Persona que tiene los siguientes datos: dni, nombre, edad.
Empleado (es una Persona) y tiene: horas de trabajo.
Jefe (es una Persona) y tiene antigüedad.
Se necesita calcular y mostrar el sueldo de los empleados, teniendo en cuenta que para los empleados el sueldo se calcula en base a las horas trabajadas y se lo multiplica por un factor 10. Para los jefes en cambio el cálculo se realiza como el producto entre la antigüedad y un factor 600.
Se pide almacenar la información de las personas en una colección Vector, y mostrar el sueldo de cada uno de ellos.

```
//Resolución.
package modelo;

public abstract class Persona
{
    protected int dni;
    protected String nombre;
    protected int edad;

    public Persona()
    {
        dni = 0;
        nombre = "";
        edad = 0;
    }

    public Persona(int d, String n, int e)
    {
        dni = d;
        nombre = n;
        edad = e;
    }

    public int getDni()
    {
        return dni;
    }

    public void setDni(int d)
    {
        dni = d;
    }

    public String getNombre()
    {
        return nombre;
    }
```

```
    public void setNombre(String n)
    {
        nombre = n;
    }

    public int getEdad()
    {
        return edad;
    }

    public void setEdad(int e)
    {
        edad = e;
    }

    public abstract float getSueldo();

    public String toString()
    {
        return "\n- DNI: " + dni + "\n- Nombre: " + nombre + "\n- Edad: " + edad;
    }
}
```

```
package modelo;

public class Empleado extends Persona
{
    private int hsTrabajo;

    public Empleado()
    {
        super();
        hsTrabajo = 0;
    }
```

```
    public Empleado(int d, String n, int e, int hs)
    {
        super(d, n, e);
        hsTrabajo = hs;
    }

    public int getHsTrabajo()
    {
        return hsTrabajo;
    }

    public void setHsTrabajo(int hs)
    {
        hsTrabajo = hs;
    }

    public float getSueldo()
    {
        return hsTrabajo * 10;
    }

    public String toString()
    {
        return super.toString() + "\n- Hs. Trabajo: " + hsTrabajo;
    }
}

package modelo;

public class Jefe extends Persona
{
    private int antiguedad;

    public Jefe()
    {
        super();
        antiguedad = 0;
    }
```

```
    public Jefe(int d, String n, int e, int a)
    {
        super(d, n, e);
        antiguedad = a;
    }

    public int getAntiguedad()
    {
        return antiguedad;
    }

    public void setAntiguedad(int a)
    {
        antiguedad = a;
    }

    public float getSueldo()
    {
        return antiguedad * 600;
    }

    public String toString()
    {
        return super.toString() + "\n- Antiguedad: " + antiguedad;
    }
}
package vista;

import java.util.ArrayList;
import java.util.Iterator;

import modelo.Empleado;
import modelo.Persona;


public class Empresa
{
```

```
    private ArrayList lista;

    public Empresa()
    {
        lista = new ArrayList();
    }

    public boolean agregarPersona(Persona p)
    {
        return lista.add(p);
    }

    public String getSueldos()
    {
        Iterator i = lista.iterator();
        Persona p;
        String sueldos = "";
        while(i.hasNext())
        {
            p = (Persona) i.next();
            sueldos += "\n- Nombre: " + p.getNombre() + " - Sueldo: " +
p.getSueldo();
        }
        return sueldos;
    }


    public float getTotalSueldos()
    {
        Iterator i = lista.iterator();
        Persona p;
        float sueldos = 0;
        while(i.hasNext())
        {
            p = (Persona) i.next();
            sueldos += p.getSueldo();
        }
        return sueldos;
```

```
    }

    public int getCantHs()
    {
        Iterator i = lista.iterator();
        Persona p;
        int hs = 0;
        while(i.hasNext())
        {
            p = (Persona) i.next();
            if (p instanceof Empleado)
            {
                Empleado e = (Empleado) p;
                hs += e.getHsTrabajo();
            }
        }
        return hs;
    }
}


/*Ejemplo usando vector*/

package vista;


import java.util.Vector;

import modelo.Empleado;
import modelo.Persona;


public class VEmpresa
{
    private Vector vec;

    public VEmpresa()
    {
```

```
        vec = new Vector();
    }

    public void agregarPersona(Persona p)
    {
       if (p!= null)
             vec.add(p);

    }

    public String getSueldos()
    {
       Persona p;
        String sueldos = "";

        for( int i=0; i < vec.size(); i++ ) {
                    p = (Persona) vec.get(i);
              sueldos += "\n- Nombre: " + p.getNombre() + " - Sueldo: " +
p.getSueldo();
        }
        return sueldos;
    }

    public float getTotalSueldos()
    {
        Persona p;
        float sueldos = 0;
        for( int i=0; i < vec.size(); i++ ) {
                    p = (Persona) vec.get(i);
            sueldos += p.getSueldo();
        }
        return sueldos;
    }

    public int getCantHs()
    {
        Persona p;
        int hs = 0;
```

```
        for( int i=0; i < vec.size(); i++ ) {
                    p = (Persona) vec.get(i);
            if (p instanceof Empleado)
            {
                Empleado e = (Empleado) p;
                hs += e.getHsTrabajo();
            }
        }
        return hs;
    }
}

mport modelo.Empleado;
import modelo.Jefe;
import modelo.Persona;

public class Main
{
    public Main()
    {
    }

    public static void main(String[] args)
    {
        int cantPersonas;
        int tipo;

        //Empresa empresa = new Empresa();

        VEmpresa empresa = new VEmpresa();
        Persona p;

        System.out.println("\n- Ingrese la Cantidad de Personas a cargar: ");
        cantPersonas = In.readInt();

        int dni, edad, ant, horas;
        String nombre;
```

```
        for(int i = 0; i < cantPersonas; i ++)
        {
            do
            {
                System.out.println("\n- Ingrese el Tipo :\n1) Empleado\n2)
Jefe\n- Tipo: ");
                tipo = In.readInt();
            }
            while(tipo != 1 && tipo != 2);

            System.out.println("\n- Ingrese el dni: ");
            dni = In.readInt();
            System.out.println("\n- Ingrese el Nombre: ");
            nombre = In.readLine();
            System.out.println("\n- Ingrese la Edad: ");
            edad = In.readInt();

            if(tipo == 1)
            {
                System.out.println("\n- Ingrese las horas trabajadas: ");
                horas = In.readInt();
                p = new Empleado(dni, nombre, edad, horas);
            }
            else
            {
                System.out.println("\n- Ingrese la antiguedad: ");
                ant = In.readInt();
                p = new Jefe(dni, nombre, edad, ant);
            }
            empresa.agregarPersona(p);
        }

        //muestra los resultados

        System.out.println("Sueldo Total: " + empresa.getSueldos());
        System.out.println("---------------Total: " + empresa.getTotalSueldos());
        System.out.println("Cant Hs.: " + empresa.getCantHs());
    } }
```

9.3 Una empresa de desarrollo de software está llevando a cabo un sistema, para una importante editorial de nuestra ciudad. En la etapa de análisis se han detectado las siguientes entidades:

Libro: código, descripción, stock, tipo de libro (1-Libro de Texto 2-Manual 3-Novela 4-Libro Infantil.) y precio.

LibroNacional: que además tiene una provincia, que está representado por un valor entero(1-Cba 2- Santa Fe 3-Bs As).Su precio de venta depende del tipo de libro, es decir su precio se incrementa si es un libro de texto en $40, Manual 50$, Novela $60 e infantil 70$.

LibroInternacional: además de los datos de libro agrega un país de origen, representado por un valor entero.(1-Italia 2-Francia 3-Alemania). Su precio de venta depende del tipo de libro, es decir su precio se incrementa si es un libro de texto en $40, Manual 50$, Novela $60 e infantil 70$, más un 20% de recargo sobre el valor del libro.

Almacenar la información de los libros en un ArrayList, que permita resolver los siguientes puntos:

- Mostrar el precio de Venta para cada libro.
- Totalizar el stock de los libros internacionales.
- Informar la cantidad de libros internacionales por tipo.

9.4 Los organizadores de una carrera de autos han pedido el desarrollo de un sistema que permita almacenar las posiciones de salida y los datos de los participantes. El sistema debe tener una colección de Participantes indicando para cada uno: Nro. De participante, Nombre del conductor, marca de automóvil, número en orden de llegada.

El sistema debe almacenar una colección de participantes en ArrayList.

Una vez finalizada la carrera, el sistema deberá consultar al operador el número de llegada de los participantes y permitir el listado de los datos de los participantes que hayan llegado en las tres primeras posiciones.

10. Referencias

- Problemas resueltos de Programación en lenguaje Java. Autores: José María Pérez Menor. Félix García Carballeira. José Manuel Pérez Lobato.
- Estructura de Colecciones en Java. http://exa.unne.edu.ar/depar/areas/informatica/programacion4/public_html/documentos/tema5.pdf
- Java como programar. Séptima edición. Deitel. Deitel.