COLECCIONES EN JAVA (*java.util*)

http://toro.itapizaco.edu.mx/paginas/JavaTut/froufe/introduccion/indice.html

Cuando se necesitan características más sofisticadas para almacenar objetos, que las que proporciona un simple array, Java pone a disposición del programador las clases colección (**collections**) : **Vector**, **Stack**, **Hashtable y BitSet**,.

Estas forman la "**The Historical Collection Classes",** muy utilizadas pero ya sustituidas por especificaciones e implementaciones mas eficientesen versiones recientes del JDK. A las nuevas se las conoce como "**The Collections Framework"**

Entre otras características, las clases colección **se redimensionan automáticamente**, por lo que se puede colocar en ellas cualquier número de objetos, sin necesidad de tener que ir controlando continuamente en el programa la longitud de la colección.

La "*gran" desventaja* del uso de las colecciones en Java es que se pierde la información de tipo cuando se coloca un objeto en una colección. Esto ocurre porque cuando se escribió la colección, el programador de esa colección no tenía ni idea del tipo de datos específicos que se iban a colocar en ella, y teniendo en mente el hacer una herramienta lo más general posible, se hizo que manejase directamente objetos de tipo **Object**, que es el objeto raíz de todas las clases en Java. **La solución es perfecta, excepto por dos razones**:

1. Como la información de tipo se pierde al colocar un objeto en la colección, cualquier tipo de objeto se va a poder colar en ella, es decir, si la colección está destinada a contener animales mamíferos, nada impide que se pueda colar un coche en ella.

2. Por la misma razón de la pérdida de tipo, la única cosa que sabe la colección es que maneja un **Object**. Por ello, hay que colocar siempre un moldeo (**casting**) al tipo adecuado antes de utilizar cualquier objeto contenido en una colección.

Enumeraciones

En cualquier clase de colección, debe haber una forma de meter cosas y otra de sacarlas; después de todo, la principal finalidad de una colección es almacenar cosas. En un ***Vector***, el método ***addElement( )*** es la manera en que se colocan objetos dentro de la colección y llamando al método ***elementAt( )*** es cómo se sacan. Vector es muy flexible, se puede seleccionar cualquier cosa en cualquier momento y seleccionar múltiples elementos utilizando diferentes índices.

El concepto de enumerador, o ***iterador***, que es su nombre más común en C++ y OOP, es un objeto cuya misión consiste en moverse a través de una secuencia de objetos y seleccionar aquellos objetos adecuados sin que el programador cliente tenga que conocer la estructura de la secuencia .

La ***Enumeration*** en Java es un ejemplo de un iterador con esas características, y las cosas que se pueden hacer son:

- Crear una colección para manejar una **Enumeration** utilizando el método ***elements().*** Esta **Enumeration** estará lista para devolver el primer elemento en la secuencia cuando se llame por primera vez al método ***nextElement().***
- Obtener el siguiente elemento en la secuencia a través del método ***nextElement().***
- Ver si hay más elementos en la secuencia con el método ***hasMoreElements().***

Con la **Enumeration** no hay que preocuparse del número de elementos que contenga la colección, ya que del control sobre ellos se encargan los métodos ***hasMoreElements()*** y ***nextElement().***

```java
import java.util.Enumeration;
import java.util.Vector;

class Coche {
    private int numCoche;
    Coche( int i ) {
        numCoche = i;
        }
    void print() {
        System.out.println( "Coche #"+numCoche );
        }
    }

class Barco {
    private int numBarco;

    Barco( int i ) {
        numBarco = i;
        }
    void print() {
        System.out.println( "Barco #"+numBarco );
        }
    }

public class EjemploEnumeration {
    public static void main( String args[] ) {
        Vector coches = new Vector();

        for( int i=0; i < 7; i++ )
            coches.addElement( new Coche( i ) );
        // No hay ningun problema en añadir un barco a los coches
        coches.addElement( new Barco( 7 ) );
        Enumeration e = coches.elements();
        while( e.hasMoreElements() )
            (( Coche )e.nextElement()).print();
        // El barco solamente es detectado en tiempo de ejecucion
        }
    }
```

```
Coche #0
Coche #1
Coche #2
Coche #3
Coche #4
Coche #5
Exception in thread "main" Coche #6
```

Vector

El ***Vector*** es muy simple y fácil de utilizar. Aunque los métodos más habituales en su manipulación son ***addElement( )*** para insertar elementos en el Vector, ***elementAt( )*** para recuperarlos y ***elements( )*** para obtener una Enumeration con el número de elementos del Vector. Para el resto de los métodos ver la API.

Las colecciones estándar de Java contienen el método ***toString( ),*** que permite obtener una representación en forma de String de sí mismas, incluyendo los objetos que contienen. Dentro de Vector, por ejemplo, toString() va saltando a través de los elementos del Vector y llama al método toString() para cada uno de esos elementos.

Stack

Un ***Stack*** es una Pila, o una colección de tipo LIFO (last-in, first-out). Es decir, lo último que se coloque en la pila será lo primero que se saque. Como en todas las colecciones de Java, los elementos que se introducen y sacan de la pila son Object, así que hay que tener cuidado con el moldeo a la hora de sacar alguno de ellos.

Los diseñadores de Java, en vez de utilizar un Vector como bloque para crear un Stack, han hecho que Stack derive directamente de Vector, así que tiene todas las características de un Vector más alguna otra propia ya del Stack.

```java
import java.util.Stack;

public class EjemploStack {
    static String diasSemana[] = {
        "Lunes", "Martes", "Miercoles", "Jueves",
        "Viernes", "Sabado", "Domingo" };

    public static void main( String args[] ) {
        Stack pila = new Stack();

        for( int i=0; i < diasSemana.length; i++ )
            pila.push( diasSemana[i]+" " );
        System.out.println( "pila = "+pila );

        // Tratando la Pila como un Vector:
        pila.addElement( "Esta es la ultima linea" );
        // Se imprime el elemento 5 (sabiendo que la cuenta empieza en 0)
        System.out.println( "Elemento 5 -> "+pila.elementAt( 5 ) );
        System.out.println( "Elementos introducidos:" );
        while( !pila.empty() )
            System.out.println( pila.pop() );
        }
    }
```

==========

```
pila = [Lunes , Martes , Miercoles , Jueves , Viernes , Sabado , Domingo ]
Elemento 5 -> Sabado
Elementos introducidos:
Esta es la ultima linea
Domingo
Sabado
Viernes
Jueves
Miercoles
Martes
Lunes
```

Hashtable

Un **Vector** permite selecciones desde una colección de objetos utilizando un número, luego parece lógico pensar que hay números asociados a los objetos.

Bien, entonces ¿qué es lo que sucede cuando se realizan selecciones utilizando otros criterios?

Un **Stack** podría servir de ejemplo: su criterio de selección es "lo último que se haya colocado en el **Stack**".

Si rizamos la idea de "selección desde una secuencia", nos encontramos con un ***mapa***, un ***diccionario*** o un ***array asociativo***. Conceptualmente, todo parece ser un vector, pero en lugar de acceder a los objetos a través de un número, en realidad se utiliza ***otro objeto***.

Esto nos lleva a utilizar ***claves*** y al procesado de claves en el programa. Este concepto se expresa en Java a través de la clase abstracta **Dictionary**. El interfaz para esta clase es muy simple:

- ***size(),*** indica cuántos elementos contiene,
- ***isEmpty(),*** es **true** si no hay ningún elemento,
- ***put(Object clave,Object valor),*** añade un **valor** y lo asocia con una **clave**
- ***get(Object clave),*** obtiene el **valor** que corresponde a la **clave** que se indica
- ***remove(Object clave),*** elimina el par **clave-valor** de la lista
- ***keys(),*** genera una **Enumeration** de todas las claves de la lista
- ***elements(),*** genera una **Enumeration** de todos los valores de la lista

La librería estándar de Java solamente incorpora una implementación de un **Dictionary**, la ***Hashtable***.

Para poder utilizar una tabla hash es necesario proporcionar una **función hash** que convierta un objeto en un entero.

Las tablas hash soportan inserciones, eliminaciones y búsquedas en **tiempo constante** en promedio.

```java
package com.practicas.septima.estructurasdatos.colecciones;

import java.util.Enumeration;
import java.util.Hashtable;

public class EjemploTablaHash {
    public static void main(String args[]) {

        Hashtable tablaHash = new Hashtable();
        System.out.println(tablaHash.isEmpty());
        tablaHash.put("España", "Madrid");
        tablaHash.put("Francia", "Paris");
        tablaHash.put("Portugal", "Lisboa");
        tablaHash.put("Italia", "Roma");
        System.out.println(tablaHash);
        System.out.println(tablaHash.size());
        System.out.println(tablaHash.isEmpty());
        System.out.println(tablaHash.containsKey("Italia"));
        tablaHash.remove("Italia");
        System.out.println(tablaHash.containsKey("Italia"));

        Enumeration enum = tablaHash.keys();
        while (enum.hasMoreElements()) {
            String key = (String) enum.nextElement();
            System.out.println(key + " : " + tablaHash.get(key));
        }
        tablaHash.clear();
        System.out.println(tablaHash.isEmpty());

    }
}
```

==========

```
true
{Italia=Roma, Portugal=Lisboa, España=Madrid, Francia=Paris}
4
false
true
false
Portugal : Lisboa
España : Madrid
Francia : Paris
true
```

La nueva librería de colecciones, "**The Collections Framework**",parte de la premisa de almacenar objetos, y diferencia dos conceptos en base a ello:

- **Colección (Collection):** un grupo de elementos individuales, siempre con alguna regla que se les puede aplicar. Una **List** almacenará objetos en una secuencia determinada y, un **Set** no permitirá elementos duplicados.

- **Mapa (Map):** un grupo de parejas de objetos clave-valor, como la **Hastable**. En principio podría parecer que esto es una Collection de parejas, pero cuando se intenta implementar, este diseño se vuelve confuso, por lo que resulta mucho más claro tomarlo como un concepto separado.

Resumiendo:

1. solamente hay tres colecciones: **Map**, **List** y **Set**; y solamente dos o tres implementaciones de cada una de ellas.

2. cualquier **Collection** puede producir un **Iterator**, mientras que una **List** puede producir un **ListIterator** (al igual que un **Iterator** normal, ya que **List** hereda de **Collection**).

3. Los interfaces que tienen que ver con el almacenamiento de datos son: **Collection, Set, List** y **Map**.

Normalmente, un programador creará casi todo su código para entenderse con estos interfaces y solamente necesitará indicar específicamente el tipo de datos que se están usando en el momento de la creación. Por ejemplo, una Lista se puede crear de la siguiente forma:

List lista = new LinkedList();

Desde luego, también se puede decidir que lista sea una lista enlazada, en vez de una lista genérica, y **precisar más el tipo** de información de la lista. Lo bueno, y la intención, del uso de interfaces es que si ahora se decide cambiar la implementación de la lista, **solamente es necesario cambiar el punto de creación**, por ejemplo:

List lista = new ArrayList();

el resto del código permanece invariable.

Por lo tanto, a la hora de sacar provecho del diagrama es suficiente con lo que respecta a los interfaces y a las clases concretas. Lo normal será construir un objeto correspondiente a una clase concreta, moldearlo al correspondiente interfaz y ya usas ese interfaz en el resto del código.

El ejemplo consiste en una colección de objetos String que se imprimen.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class EjemploSencillo {
    public static void main( String args[] ) {
        Collection c = new ArrayList();

        for( int i=0; i < 10; i++ )
            c.add( Integer.toString( i ) );

        Iterator it = c.iterator();
        while( it.hasNext() )
            System.out.print( " "+it.next() );
        }
    }
```

```
0 1 2 3 4 5 6 7 8 9
```

- La primera línea del método main() **crea un objeto ArrayList y lo moldea a una Collection**. Como este ejemplo solamente utiliza métodos de Collection, cualquier objeto de una clase derivada de Collection debería funcionar, pero se ha cogido un ArrayList porque es el caballo de batalla de las colecciones **y viene a tomar el relevo al Vector.**

- El método ***add(),*** como su nombre sugiere, coloca un nuevo elemento en la colección. Sin embargo, la documentación indica claramente que ***add( )*** "***asegura que la colección contiene el elemento indicado***". Esto es para que un Set tenga significado, ya que solamente añadirá el elemento si no se encuentra en la colección. Con un ArrayList, o cualquier otra lista ordenada, add() significa siempre "**colocarlo dentro**".

Todas las colecciones pueden producir un **Iterator** invocando al método ***iterator( ).***

Un **Iterator** viene a ser equivalente a una Enumeration, **a la cual reemplaza**, excepto en los siguientes puntos:

1. Utiliza un nombre que está históricamente aceptado y es conocido en toda la literatura de programación orientada a objetos
2. Utiliza nombres de métodos más cortos que la **Enumeration**: ***hasNext( )*** en vez de ***hasMoreElements( ),*** o ***next ()*** en lugar de ***nextElement( )***
3. Añade un nuevo método, ***remove(),*** que permite eliminar el último elemento producido por el Iterator. Solamente se puede llamar a remove() una vez por cada llamada a next()

En el ejemplo anterior se utiliza un Iterator para desplazarse por la colección e ir imprimiendo cada uno de sus elementos.

Colecciones

A continuación se indican los métodos que están disponibles para las colecciones, es decir, lo que se puede hacer con un **Set** o una **List**, aunque las listas tengan funcionalidad añadida que ya se verá.

boolean add(Object)

Asegura que la colección contiene el argumento. Devuelve ***false*** si no se puede añadir el argumento a la colección

boolean addAll(Collection)

Añade todos los elementos que se pasan en el argumento. Devuelve ***true*** si es capaz de incorporar a la colección cualquiera de los elementos del argumento

void clear()

Elimina todos los elementos que componen la colección

boolean contains(Object)

Verdadero si la colección contiene el argumento que se pasa como parámetro

boolean isEmpty()

Verdadero si la colección está vacía, no contiene elemento alguno

Iterator iterator()

Devuelve un Iterator que se puede utilizar para desplazamientos a través de los elementos que componen la colección

boolean remove(Object)

Si el argumento está en la colección, se elimina una instancia de ese elemento y se devuelve ***true*** si se ha conseguido

boolean removeAll(Collection)

Elimina todos los elementos que están contenidos en el argumento. Devuelve true si consigue eliminar cualquiera de ellos

boolean retainAll(Collection)

Mantiene solamente los elementos que están contenidos en el argumento, es lo que sería una intersección en la teoría de conjuntos. Devuelve verdadero en caso de que se produzca algún cambio

int size()

Devuelve el número de elementos que componen la colección

Object [ ] toArray()

Devuelve un array conteniendo todos los elementos que forman parte de la colección. Este es un método opcional, lo cual significa que no está implementado para una Collection determinada. Si no puede devolver el array, lanzará una excepción de tipo UnsupportedOperationException

Ejemplo Collections

```
import java.util.*;

public class EjemploCollection {
    // Rellena la Colección con el número de elementos que se especifica
    // en 'tamano', comenzando a partir del elemento que indica el
    // parámetro 'primero'
    public static Collection fill( Collection c,int primero,int tamano ) {
        for( int i=primero; i < primero+tamano; i++ )
            c.add(Integer.toString( i ) );
        return( c );
        }

    // También rellena la Colección con el número de elementos que se
    // indique en el parámetro 'tamano', pero se inicia desde el
    // elemento 0
    public static Collection fill( Collection c,int tamano ) {
        return( fill( c,0,tamano ) );
        }

    // En este caso se rellena la Colección comenzando en el elemento
    // 0 y rellenando 10 elementos
    public static Collection fill( Collection c ) {
        return( fill( c,0,10 ) );
        }

    // Crea una estructura de datos y la moldea a Colección
    public static Collection nuevaColeccion() {
        // Se utiliza un ArrayList por simplicidad, pero se puede ver
        // como una Collection genérica en cualquier lugar del
        // programa
        return( fill( new ArrayList() ) );
        }

    // Rellena una Colección con un rango de valores
    public static Collection nuevaColeccion( int primero,int tamano ) {
        return( fill( new ArrayList(),primero,tamano ) );
        }

    // Se desplaza a través de una Lista utilizando un Iterador
    public static void print( Collection c ) {
        for( Iterator x=c.iterator(); x.hasNext(); )
            System.out.print( x.next()+" " );
        System.out.println();
        }
```

```
public static void main( String args[] ) {
    Collection c = nuevaColeccion();
    c.add( "diez" );
    c.add( "once" );
    print( c );

    // Busca los elementos máximo y mínimo; esto puede hacerse
    // de diversas formas, dependiendo de como esté implementado
    // el interfaz Comparable
    System.out.println( "Collections.max(c) = "+
        Collections.max( c ) );
    System.out.println( "Collections.min(c) = "+
        Collections.min( c ) );

    // Añade una Colección a otra Colección
    c.addAll( nuevaColeccion() );
    print( c );
    c.remove( "3" ); // Elimina el primero
    print( c );
    c.remove( "3" ); // Elimina el segundo
    print( c );

    // Elimina todos los componentes que esten en la Colección
    // que se pasa como argumento
    c.removeAll( nuevaColeccion() );
    print( c );
    c.addAll( nuevaColeccion() );
    print( c );

    // Mira si un elemento determinado está en la Colección
    System.out.println( "c.contains(\"4\") = "+c.contains("4") );
    // Mira si la SubColección está en la Colección base
    System.out.println( "c.containsAll( nuevaColeccion() ) = "+
    c.containsAll( nuevaColeccion() ) );

    Collection c2 = nuevaColeccion( 5,3 );
    // Mantiene todos los elementos que están en las colecciones
    // 'c' y 'c2'. Hace una intersección de las dos colecciones
    c.retainAll( c2 );
    print( c );

    // Elimina de la colección 'c' todos los elementos que se
    // encuentran también en la colección 'c2'
    c.removeAll( c2 );
    System.out.println( "c.isEmpty() = "+c.isEmpty() );
    c = nuevaColeccion();
    print( c );

    // Se eliminan todos los elementos
    c.clear();
    System.out.println( "Despues de c.clear():" );
    print( c );
    }
}
```

```
0 1 2 3 4 5 6 7 8 9 diez once
Collections.max(c) = once
Collections.min(c) = 0
0 1 2 3 4 5 6 7 8 9 diez once 0 1 2 3 4 5 6 7 8 9
0 1 2 4 5 6 7 8 9 diez once 0 1 2 3 4 5 6 7 8 9
0 1 2 4 5 6 7 8 9 diez once 0 1 2 4 5 6 7 8 9
diez once
diez once 0 1 2 3 4 5 6 7 8 9
c.contains("4") = true
c.containsAll( nuevaColeccion() ) = true
5 6 7
c.isEmpty() = true
0 1 2 3 4 5 6 7 8 9
Despues de c.clear():
```

Listas

Hay varias implementaciones de **List**, siendo **ArrayList** la que debería ser la elección por defecto, en caso de no tener que utilizar las características que proporcionan las demás implementaciones.

List (interfaz)

La ordenación es la característica más importante de una Lista, asegurando que los elementos **siempre se mantendrán en una secuencia concreta**. La Lista incorpora una serie de métodos a la Colección que permiten la inserción y borrar de elementos en medio de la Lista.

Además, en la Lista Enlazada se puede generar un ***ListIterator*** **para moverse a través de las lista en ambas direcciones.**

ArrayList

Es una Lista volcada en un array. **Se debe utilizar en lugar de Vector** como almacenamiento de objetos de propósito general.

Permite un acceso aleatorio muy rápido a los elementos, pero realiza con bastante lentitud las operaciones de insertado y borrado de elementos en medio de la Lista.

1. Añadir y eliminar elementos por el final es muy simple, con un coste O(1). Tomar un elemento de una posición especifica tiene también un coste O(1).
2. Añadir y quitar elementos del interior es más costoso: O(n-i), siendo n el tamaño de la lista e i la posición del elemento que se va a eliminar. Añadir y eliminar requiere desplazar el resto del array una posición hacia delante o hacia atrás

Se puede utilizar un ***ListIterator*** para moverse hacia atrás y hacia delante en la Lista.

LinkedList

Es una lista doblemente enlazada. Sus prestaciones son prácticamente las inversas de **ArrayList:**

1. Permiten **inserciones y borrado** de elementos de en medio de la Lista **muy rápidas**. De coste O(1).
2. Sin embargo es bastante lento el acceso aleatorio, en comparación con la **ArrayList**. Tomar un elemento de la posición i tiene una coste O(i).

Dispone además de los métodos ***addLast(), getFirst(), getLast(), removeFirst()*** **y** ***removeLast()***, que no están definidos en ningún interfaz o clase base y que permiten utilizar la Lista Enlazada como una Pila, una Cola o una Cola Doble.

LinkedList es también una buena opción para listas donde la mayoría de las operaciones no se realizan al final.

```
    public static void main(String args[]) {
        // Crea y rellena una nueva lista cada vez
        System.out.println("Test Básico");
        testBasico(fill(new LinkedList()));
        testBasico(fill(new ArrayList()));

        System.out.println("Mover Iterador");
        moverIterador(fill(new LinkedList()));
        moverIterador(fill(new ArrayList()));

        System.out.println("Manipular Iterador");
        manipularIterador(fill(new LinkedList()));
        manipularIterador(fill(new ArrayList()));

        System.out.println("Test Visual");
        testVisual(fill(new LinkedList()));
        System.out.println("Test Visual Lista enlazada");
        testListaEnlazada();
    }
}
```

=======

```
Test Básico
Mover Iterador
Manipular Iterador
Test Visual
0 1 2 3 4 5 6 7 8 9
b = 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 uno 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
5
6
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 uno 47 7 8 9 0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0 9 8 7 47 uno 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
testVisual concluido
Test Visual Lista enlazada
0 1 2 3 4
dos uno 0 1 2 3 4
dos
dos
uno
4
0 1 2 3
```

```java
/**
 * Este ejemplo es una muestra de las cosas que se permiten hacer
 * utilizando las Listas que se incorporan como parte de las nuevas
 * colecciones implementadas en el JDK 1.2
 */
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class EjemploListas {
    static boolean b;
    static Object o;
    static int i;
    static Iterator it;
    static ListIterator lit;
    // Utilizamos el método de rellenado
    // correspondiente al ejemplo anterior por conveniencia
    public static List fill(List a) {
        return ((List) EjemploCollection.fill(a));
    }
    // Se puede utilizar un Iterator, al igual que se hace con una
    // Collection, pero también se puede utilizar acceso aleatorio
    // con get()
    public static void print(List a) {
        for (int i = 0; i < a.size(); i++)
            System.out.print(a.get(i) + " ");
        System.out.println();
    }
```

```
public static void testBasico(List a) {
    a.add(1, "x"); // Añade en la posición 1
    a.add("x"); // Añade en la ultima posición
    a.addAll(fill(new ArrayList())); // Añade una colección
    //      Añade una colección a partir de la posición 3
    a.addAll(3, fill(new ArrayList()));
    b = a.contains("1"); // Comprueba la presencia de un elemento
    // Comprueba la presencia de una colección completa
    b = a.containsAll(fill(new ArrayList()));
    // Las Listas permiten acceso aleatorio, que es fácil en el
    // caso de ArrayList y complicado en caso de LinkedList
    o = a.get(1); // Recoge el elemento de la posición 1
    i = a.indexOf("1"); // Indica el índice del elemento
    b = a.isEmpty(); // Comprueba si hay elementos
    it = a.iterator(); // Iterator normal
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Comienza en la posición 3
    i = a.lastIndexOf("1"); // Ultima coincidencia
    // Ultima coincidencia a partir de la posición 2
    a.remove(1); // Elimina el elemento de la posición 1
    a.remove("3"); // Elimina ese objeto
    a.set(1, "y"); // Fija el elemento de la posición 1 a "y"
    // Genera un array a partir de la Lista
    Object array[] = a.toArray();
    // Mantiene todos los elementos que se pasan como argumento,
    // equivale a la intersección de dos conjuntos
    a.retainAll(fill(new ArrayList()));
    // Elimina todos los elementos de la lista
    a.removeAll(fill(new ArrayList()));
    i = a.size(); // Indica el tamaño
    a.clear(); // Elimina todos los elementos
}

public static void moverIterador(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
    i = it.previousIndex();
}
```

```
public static void manipularIterador(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
    // Se debe mover a un elemento después de add()
    it.next();
    // Elimina el elemento
    it.remove();
    // Se debe mover a un elemento después de remove()
    it.next();
    // Cambia el elemento que se ha introducido
    it.set("47");
}

public static void testVisual(List a) {
    print(a);
    List b = new ArrayList();
    fill(b);
    System.out.print("b = ");
    print(b);
    a.addAll(b);
    a.addAll(fill(new ArrayList()));
    print(a);
    // Inserta, elimina y reemplaza elementos utilizando un
    // ListIterator
    ListIterator x = a.listIterator(a.size() / 2);
    x.add("uno");
    print(a);
    System.out.println(x.next());
    x.remove();
    System.out.println(x.next());
    x.set("47");
    print(a);
    // Se mueve hacia atrás en la lista
    x = a.listIterator(a.size());
    while (x.hasPrevious())
        System.out.print(x.previous() + " ");
    System.out.println();
    System.out.println("testVisual concluido");
}
```

```java
// Hay algunas cosas que solamente se pueden hacer con las
// listas enlazadas: LinkedLists
public static void testListaEnlazada() {
    LinkedList ll = new LinkedList();
    EjemploCollection.fill(ll, 5);
    print(ll);

    // Trata la lista como una pila, (push)
    ll.addFirst("uno");
    ll.addFirst("dos");
    print(ll);
    // Como si sacase la cabecera de la pila
    System.out.println(ll.getFirst());
    // Como si sacase elementos de la pila (pop)
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());

    // Trata la lista como una cola, sacando los elementos del
    // final de la cola
    System.out.println(ll.removeLast());
    // Con las anteriores operaciones, tenemos una cola doble (dequeue)
    print(ll);
}
```