Estructuras de Datos
Grado en Ingeniería Informática en Sistemas de Información - Curso 2012/2013

Enseñanzas de Prácticas y Desarrollo
EPD-3: Listas

Objetivos

- Manipular colecciones de tipo lista haciendo uso de sus particularidades.
- Crear programas que manipulen listas beneficiándose de los métodos estáticos disponibles en la clase *Collections*.

Conceptos

1. Listas

Las listas son un tipo de colección en la que se mantiene el orden secuencial de sus elementos. Lo anterior implica que si se recorre la lista con un iterador, obtendremos los elementos siguiendo el orden en que éstos han sido insertados en la lista.

Todas las clases que permitan manejar listas han de implementar la interfaz *List*. Recuerde que esta interfaz está incluida en el paquete *java.util*, por lo que será necesario importar dicho paquete. La interfaz *List* hereda de la interfaz *Collection*, y por tanto incorpora todos los métodos de esta última. La interfaz *List* incorpora los siguientes métodos adicionales a los incorporados por *Collection*:

- *Object get(int index)*: Este método devuelve el elemento cuyo índice se pasa como argumento. Lanzará una excepción de la clase *IndexOutOfBoundsException* si el índice está fuera de rango. Los índices irán desde 0 a n-1, siendo *n* el número de elementos de la colección.

- *Object set(int index,Object o)*: El método reemplazará el objeto situado en la posición indicada por el índice, por el objeto *o* que se pasa como argumento. Se devolverá como resultado el objeto que ha sido reemplazado. Si se indica un índice fuera de rango se provocará una excepción. Este método es opcional, por lo que puede lanzar una excepción de la clase *UnsupportedOperationException* en caso de que la clase que implementa la interfaz no permita la operación.

- *boolean add(Object element)*: Este método, presente en la interfaz *Collection*, se comporta en las listas insertando el objeto que se le pasa como argumento al final de la lista. Devolverá verdadero en caso de que se haya realizado la inserción y falso en caso contrario. Esta operación es opcional.

- *void add(int index, Object element)*: El método inserta el elemento en la posición indicada por el índice, ambos pasados como argumentos. Al realizar la inserción se desplazará una posición hacia el final al elemento que originalmente estaba situado en el índice indicado, y los elementos situados a continuación del mismo. Esta operación es opcional.

- *Object remove(int index)*: Este método se comporta eliminando de la lista el elemento indicado por el índice, con la peculiaridad de que los elementos situados a continuación del elemento a eliminar se desplazarán una posición hacia el inicio de la lista. Devuelve el elemento eliminado de la lista. Esta operación es opcional.

- *boolean addAll(int index, Collection c)*: El método insertará todos los elementos de la colección que se le pasa como argumento en la posición especificada por el índice, desplazando hacia el final al elemento situado originalmente en dicha posición y los elementos posteriores a éste. Devuelve verdadero si la lista ha cambiado como resultado de la operación. Esta operación es opcional.

- *int indexOf(Object o)*: Este método devuelve el índice de la primera ocurrencia del objeto *o* que se pasa como argumento. Devuelve -1 en caso de que el objeto no sea un elemento de la lista.

- *int lastIndexOf(Object o)*: El método devuelve el índice de la última ocurrencia del objeto *o* que se pasa como argumento. El método devolverá -1 si la lista no contiene como elemento a dicho objeto.

- *List subList(int from, int to)*: Este método devuelve una vista de la lista conteniendo un subconjunto de sus elementos. Se incluirá en la vista devuelta a los elementos cuyo índice esté comprendido entre *from* (inclusive) hasta *to* (exclusive). Una vista de una lista no es una nueva lista, si no una porción de ella, por lo que las operaciones aplicadas a sus elementos se

verán reflejadas en la lista original. Este método se emplea para aplicar operaciones sólo sobre un subconjunto de elementos de la lista, evitando tener que iterar sobre ellos. Por ejemplo, para eliminar los elementos cuyo índice va desde 3 hasta 7 de una lista *l* se emplearía la sentencia *l.subList(3,7).clear( );*

En esta práctica haremos uso de la clase *ArrayList( )*. Por tanto, si deseamos crear una lista apuntada por una referencia l, emplearemos un código similar al siguiente:

```
List l = new ArrayList();
```

Bibliografía Básica

Documentación de la API de Java:

- Listas: http://java.sun.com/docs/books/tutorial/collections/interfaces/list.html
- Especificación de la interfaz *List*: http://java.sun.com/j2se/1.5.0/docs/api/java/util/List.html
- Especificación de la clase *Collections*: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html
- Algoritmos para la manipulación de colecciones: http://java.sun.com/docs/books/tutorial/collections/algorithms/index.html

Experimentos

E1. a) Analice el siguiente código y prediga antes de ejecutarlo la salida por pantalla de cada una de sus secciones.

```
import java.util.*;

public class Experimento1 {

    public static void main(String[] args){
        List l = new ArrayList();

        for(int i=1;i<=5;i++)
            l.add(i*Math.PI);
        System.out.println(l);

        for(int i=0;i<5;i++)
            l.add(i*2, i*Math.E);
        System.out.println(l);

        for(int i=0;i<5;i++)
            System.out.println(l.get(i*2));

        for(int i=0;i<5;i++)
            l.remove(i+1);
        System.out.println(l);
    }
}
```

b) Ejecute el código y corrobore su respuesta al apartado anterior.

E2. a) Estudie el código del siguiente programa. Su objetivo es copiar los elementos de una lista en otra. Ejecútelo para ver el resultado.

```
import java.util.*;

public class Experimento2 {

    public static void main(String[] args){
        List l,m;
        l = new ArrayList();
        m= new ArrayList();

        for(int i=0;i<5;i++)
            l.add(i);
        System.out.println("l: " + l);
```

```
            Collections.copy(m,l);

            System.out.println("m: " + m);
        }
    }
```

b) ¿Por qué no funciona el programa? Proporcione una solución para que el programa realice su cometido.

E3. a) El siguiente programa busca la posición del número 20 en la lista creada en el mismo programa. Analice el código y ejecútelo para obtener su resultado.

```
public class Experimento3 {

    public static void main(String args[]){
        List l = new ArrayList();

        for(int i=5;i>=0;i--)
            l.add(i*10);
        System.out.println("Lista: " + l);

        int posicion = Collections.binarySearch(l,20);

        if(posicion >= 0)
            System.out.println("El 20 está en la posición " + posicion + " de la lista");
        else
            System.out.println("No está el número 20 en la lista");
    }
}
```

b) ¿Hay algún problema? ¿Por qué? Modifique el código para evitarlo.

c) Añada el código necesario para que el programa anterior imprima el mínimo y máximo número de la lista.

Ejercicios

EJ1. (30 mins.) a) Añada a la clase *MiCollections* desarrollada en los ejercicios anteriores un método similar al método *rotate* de la clase *Collections*. Este método recibirá una lista y un entero y rotará hacia la derecha la lista tantos lugares como se especifica en el número entero. Tenga en cuenta que el entero que indica los lugares a rotar puede ser negativo, lo que implicaría rotar en sentido contrario.

Por ejemplo, si disponemos de la lista [‘a’, ’b’, ’c’, ’d’, ’e’, ’f’] y la rotamos 3 lugares, el resultado sería {‘d’, ’e’, ’f’, ’a’, ’b’, ’c’}, y si indicamos que rote -2 lugares el resultado sería [‘c’, ’d’, ‘e’, ’f’, ’a’, ’b’].

b) Cree un programa que compruebe si los resultados obtenidos por su método son similares a los obtenidos por el método de la clase *Collections*.

EJ2. (30 mins.) a) Cree una nueva clase llamada *MiCollections*. Esta clase incorporará nuevos métodos estáticos, adicionales a los ofrecidos por la clase *Collections*, que aporten nuevas utilidades para el manejo de listas y colecciones.

b) Añada un método estático llamado *reverse* que reciba una lista e invierta el orden de sus elementos.

c) Añada un método estático a la clase anterior llamado *reverseSort* que reciba una lista como argumento y ordene los elementos de ésta en orden descendente. Para ello, combine los métodos ofrecidos en la clase *Collections* y el método desarrollado en el punto anterior.

d) Cree un programa para demostrar el funcionamiento de estos métodos.

Problemas

P1. (60 mins.) a) Cree una interfaz llamada *IEnteros*. Esta interfaz contendrá un método llamado *desordenar*, que no recibirá ningún argumento ni devolverá ningún valor, y un método llamado *intercambio* que recibirá dos enteros como argumento sin devolver valor alguno. La interfaz también contendrá el método *toString*.

b) Cree una clase llamada *Enteros* que implementará la interfaz anterior. Esta clase poseerá como atributo una referencia a una lista que será creada cuando se llame al método constructor de la clase. El método constructor recibirá como argumento un vector de números enteros e incorporará éstos a la lista, siguiendo el orden en que aparecen en el vector. Recuerde implementar el método *toString* delegando en el método *toString* de la interfaz *List*.

c) La clase anterior deberá implementar el método *intercambio*. Este método recibe dos números enteros como argumento, indicando dos posiciones de elementos de la lista, y deberá de intercambiar los mismos.

d) Además, la clase anterior deberá implementar el método *desordenar*. Este método desordenará la lista intercambiando cada uno de los elementos de la misma, desde 0 hasta n-1 (siendo n el tamaño de la lista), con un elemento situado en una posición elegida aleatoriamente. Para generar número aleatorios recuerde que la clase *Ramdom*, incluida en *java.util*, dispone del método nextInt(int n). Recuerde también que el rango en el que han de estar incluidos los números aleatorios es [0,n-1].

e) Cree un programa que demuestre el funcionamiento de la interfaz *IEnteros* y la clase *Enteros*.

P2. (60 mins.) a) Cree una interfaz llamada *ICarta*. La interfaz contendrá los métodos consultores y modificadores para los atributos de la clase que se describe en el punto siguiente, así como el método *toString*.

b) Cree una clase llamada *Carta*. Esta clase representará una carta de una baraja francesa (de poker). Para ello, la clase dispondrá de dos atributos: *numero* y *palo*. El número será un entero que tomará valores del 1 al 13, teniendo en cuenta que el 1 es el As, el 11 es la J, el 12 es la Q y el 13 la K. El palo será codificado usando igualmente un entero (0 picas ♠, 1 corazones ♥, 2 tréboles ♣, 3 diamantes ♦). La clase implementará la interfaz *ICarta*, y su método *toString* devolverá una cadena indicando la carta que representa el objeto, por ejemplo “4 de Corazones”. La clase dispondrá de un método constructor que recibirá dos números enteros para inicializar correctamente sus dos atributos.

c) Cree una interfaz llamada *IBaraja*. Esta interfaz contendrá el método *reparteCarta*, que no recibirá ningún argumento y devolverá una referencia a *ICarta*, así como el método *barajar*, que no recibirá ningún argumento ni devolverá valores.
d) Cree una clase llamada *Baraja* que implemente la interfaz *IBaraja*. Esta clase representará una baraja de cartas y para ello contendrá como atributo una lista de cartas. Su método constructor por defecto creará la citada lista e incorporará dentro de la misma un objeto *Carta* inicializado correctamente por cada una de las cartas de la baraja de poker. La clase implementará el método *reparteCarta*, que devolverá una referencia a la primera carta de la lista y eliminará la carta de la misma. También se implementará el método *barajar*. Este método funcionará de manera similar al método *desordenar* del problema anterior.

e) Cree un programa principal que demuestre el uso de los métodos de la clase anterior.

P4. (35 mins.) Añada a la clase *MiCollections* el método *disconexo* de funcionamiento similar al método *disjoint* de la clase *Collections*. Este método recibe como argumento dos colecciones de tipo Lista y devuelve verdadero si las colecciones no tienen elementos en común, es decir ninguno de los elementos de una colección es igual a algún elemento de la otra. El método devuelve falso en caso contrario.

P5. (35 mins.) Incluya en la clase *MiCollections* el método *frecuencia* que emule el funcionamiento del método *frecuency* de la clase *Collections*. Este método recibe una colección de tipo Lista y un objeto y devuelve un entero indicando el número de elementos de la colección iguales al objeto que se ha pasado como argumento.

P6. (50 mins.) Cree un nuevo método para la clase *MiCollections* llamado *sortAs*. Este método recibirá como argumento dos listas y devolverá como resultado una lista procedente de ordenar la primera lista con un orden similar a la segunda. Esto es, se ordenará la primera lista siguiendo el orden de aparición de los elementos en la segunda lista. Si hay elementos repetidos en la primera lista éstos se colocarán uno al lado del otro. Si hay elementos repetidos en la segunda lista se ignorarán. Los elementos de la primera lista que no aparezcan en la segunda quedarán al final de la lista, siguiendo su orden original, después del proceso de ordenación.

Por ejemplo, si disponemos de las siguientes listas:

Primera lista: {1, 6, 3, 6, 4, 7, 2, 3, 8, 3}
Segunda lista: {4, 3, 8, 5, 1, 8, 2}

La lista resultante es: {4, 3, 3, 3, 8, 1, 2, 6, 6, 7}

Ampliación de Bibliografía

- Thinking in Java, 3rd Edition. Bruce Eckel. Prentice Hall, 2002 (http://www.mindview.net/Books/TIJ/). Capítulo 11.
- Aprenda Java como si Estuviera en Primero (http://mat21.etsii.upm.es/ayudainf/aprendainf/Java/Java2.pdf. Capítulo 4, sección 5. Páginas 71 a 76