Estructuras de Datos
Grado en Ingeniería Informática en Sistemas de Información

Enseñanzas Prácticas y de Desarrollo
EPD-2: Listas, Conjuntos e Iteradores

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*.
- Definir un orden natural y órdenes no naturales para los objetos de una clase.
- Ordenar listas y vectores usando diferentes tipos de criterios.
- Conocer las distintas implementaciones de la interfaz *List* incluidas en JCF.
- Comprender los diferentes esquemas de recorrido de un objeto *List* mediante el uso de la interfaz *ListIterator*.

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*:

- *E 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.

- *E set(int index,E 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(E 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, E 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.

- *E 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<? Extends E> 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<E> 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<TIPO> l = new ArrayList<>();
```

1. Ordenación de listas de objetos en Java

Como se ha explicado en anteriores guiones, las listas son colecciones en las que se mantiene el orden en el que han sido introducidos los objetos. Por tanto, cuando un iterador las recorre obtendrá los objetos de la lista en el orden descrito anteriormente. Cuando trabajamos con listas suele ser necesario ordenarlas para algún propósito. Por ejemplo, si disponemos de una lista de empleados, es posible que tengamos que ordenar ésta por los apellidos de los empleados para generar algún tipo de informe.

Para ordenar una serie de objetos hace falta un criterio de ordenación y un algoritmo de ordenación. En el anterior ejemplo de una lista de empleados, el criterio de ordenación es el orden alfabético de los apellidos de los empleados. El algoritmo de ordenación nos lo proporciona Java en las clases de utilidad incluidas en JCF. El programador sólo deberá indicar a Java qué criterio de ordenación se debe seguir.

El criterio de ordenación que se desea aplicar se puede especificar implementando las interfaces *Comparable* y/o *Comparator*. Estas interfaces, incluidas en JCF, definen una serie de métodos que emplean automáticamente los sistemas de ordenación incluidos en Java. Estos métodos, implementados por el programador, indicarán a Java cómo ordenar los objetos, esto es, el criterio de ordenación elegido.

2. Orden natural: La interfaz Comparable

Para dotar de un criterio de orden por defecto a los objetos de una clase se emplea la interfaz *Comparable*, que es parte del paquete por defecto *java.lang*. Este paquete, a diferencia de los demás, no es necesario importarlo ya que Java lo importa automáticamente al compilar. Esta interfaz se define de la siguiente forma:

```
public interface Comparable{
       public int compareTo(Object o);
}
```

El único método que incluye permitirá comparar dos objetos de la clase que implemente la interfaz. La implementación del método deberá comparar el objeto que invoca dicho método y el objeto *o* que se pasa como argumento. El método devolverá como resultado de la comparación un número entero, cuyo valor será:

- Cero en el caso de que los objetos sean iguales.
- Un valor negativo, típicamente -1, si el objeto que invoca el método es anterior, en el orden establecido, al objeto que se pasa como argumento.
- Un valor positivo típicamente +1, si el objeto que invoca el método es posterior, en el orden establecido, al objeto que se ha pasado como argumento.

Las clases que implementan la interfaz *Comparable* se dice que poseen un *orden natural*. Entenderemos por orden natural aquel orden que se aplica típicamente a los objetos de una clase. Por ejemplo, el orden natural de los números enteros es el orden ascendente, es decir, de menor a mayor. Muchas de las clases de la API de Java implementan esta interfaz, entre ellas tenemos las clases de envoltura y la clase *String*.

Se ha de resaltar que es muy importante que la implementación del método *compareTo* sea coherente con el resultado que devuelve el método *equals*. Si el método *equals* indica que dos objetos son iguales, el método *compareTo* también debería indicarlo, y viceversa para el caso en que sean diferentes. Esta regla no es obligatoria, pero si no se cumple los distintos tipos de

colecciones pueden comenzar a funcionar de manera errónea cuando se intente ordenar sus elementos o buscar elementos dentro de ellas.

Otra cuestión que ha de observar toda implementación del método *compareTo* es la transitividad. Se dice que un orden cumple la propiedad transitiva si dados tres objetos *x*, *y*, *z*, cumpliendo que *x* < *y* e *y* < *z* (donde *a* <*b* significa que *a* es anterior a *b*), se cumple que *x* < *z*. En caso de que la implementación no cumpla la propiedad transitiva los distintos tipos de colecciones pueden comenzar a funcionar de manera errónea.

El orden natural de los objetos de una clase puede ser empleado para ordenar los elementos de vectores o listas. Para ordenar un vector de objetos siguiendo el orden natural definido para los mismos se emplea el método *sort* de la clase *Arrays* que estudiamos en anteriores sesiones de prácticas. En el caso de las colecciones existe un método equivalente en la clase *Collections*. Por ejemplo, para ordenar una lista *l* este método se invoca empleando la siguiente sentencia:

```
List l = new ArrayList();
...
Collections.sort(l);
```

3. Órdenes no naturales: La interfaz Comparator

Puede que los objetos de una clase tengan un orden natural pero para una tarea específica deseemos ordenarlos con respecto a otro criterio. A este tipo de criterios los denominaremos *órdenes no naturales*. Por ejemplo, un orden no natural para los números enteros consistiría en ordenar los números de forma descendente, es decir, de mayor a menor.

Para una determinada clase sólo puede definirse un orden natural, pero pueden existir tantos órdenes no naturales para sus objetos como sea necesario. En Java, por cada orden no natural que deseemos definir deberemos crear una clase que represente dicho orden. A esta clase la llamaremos genéricamente *comparador*.

Toda clase que represente un comparador deberá implementar la interfaz *Comparator*. Esta interfaz está definida de la siguiente forma:

```
public interface Comparator {
        int compare(Object o1, Object o2);
}
```

Su único método permite comparar los objetos *o1* y *o2* que se le pasan como argumentos. Este método devolverá un entero cuyo valor será:

- Cero cuando los dos objetos sean iguales.
- Negativo, típicamente -1, cuando el objeto *o1* preceda en el orden definido al objeto *o2*.
- Positivo, típicamente +1, cuando el objeto *o1* suceda en el orden definido al objeto *o2*.

Al igual que sucede con el método *compareTo* de la interfaz *Comparable*, es fundamental que la implementación del método *compare* sea coherente con el valor devuelto por *equals* y que cumpla la propiedad transitiva, ya que en caso contrario los métodos de ordenación incluidos en Java pueden comenzar a funcionar de manera errónea e imprevisible.

Tanto la clase *Arrays* como la clase *Collections* disponen de versiones sobrecargadas del método *sort* que aceptan, como argumento adicional al vector o lista a ordenar, un objeto de una clase comparadora que defina el criterio que se desea aplicar en el proceso de ordenación. Estos métodos se emplean de la siguiente manera:

```
Comparator comparador = new ClaseComparadora();
        ...
Arrays.sort(v,comparador);
        ...
Collections.sort(c,comparador);
```

donde *v* y *c* son el vector y la colección a ordenar respectivamente, y *comparador* es una referencia a un objeto de la clase comparadora en la que se implementa el método *compare* que define el criterio de ordenación que se desea aplicar.

Además de lo anterior, la clase *Collections* ofrece los siguientes métodos de utilidad para manipular colecciones empleando comparadores:

2. public static Comparator reverseOrder( ): Este método devuelve un comparador que al ser aplicado impone el orden inverso al orden natural de los objetos de una colección. Es útil para emplearlo en conjunción con el método *sort* para ordenar una colección de forma descendente.

3. public static Comparator reverseOrder(Comparator c): Similar al anterior método sólo que invierte el orden no natural definido por el comparador que se le pasa como argumento.

4. *public static Object max(Collection c, Comparator c)*: *)*: Este método devolverá el máximo de los elementos de la colección según el orden no natural de los elementos definido por el comparador que se pasa como argumento.

5. *public static Object min(Collection c, Comparator c)*: Similar al método anterior, pero en este caso devolviendo el mínimo de los elementos de la colección en función de orden no natural definido por el comparador que se pasa como argumento.

4. Implementaciones de Listas

JCF incluye un par de clases que implementan la interfaz *List*, las clases *ArrayList* y *LinkedList*. Estas clases se diferencian por la forma que tienen de organizar en memoria una lista.

La clase *ArrayList* organiza la lista en memoria como un vector. Tiene la ventaja de que el acceso aleatorio a los elementos de la lista, mediante su índice, es muy rápido. Como desventaja se ha de destacar que el tiempo necesario para insertar y borrar elementos de una lista es muy alto ya que debe desplazar los elementos en el vector.

Por el contrario, la clase *LinkedList* organiza la lista como una lista doblemente enlazada. En una lista doblemente enlazada cada elemento posee una referencia de los elementos anterior y posterior al mismo. Este tipo de implementación tiene como ventaja que el tiempo necesario para insertar y eliminar elementos de la lista es muy pequeño, ya que sólo es necesario variar un par de referencias. Como parte negativa, se ha de tener en cuenta que el acceso aleatorio a los elementos de la lista usando su índice es muy lento, ya que para encontrar un elemento en función de su índice hay que ir recorriendo una por una las referencias al siguiente elemento desde el primer elemento de la lista.

Las clases *ArrayList* y *LinkedList* tienen en común un par de constructores:

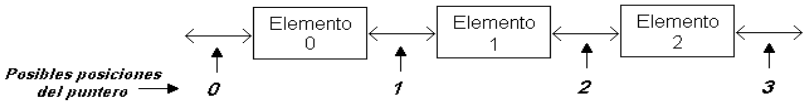
- *ArrayList( )* y *LinkedList( )*: Es el constructor por defecto que crea una lista vacía.
- *ArrayList(Collection c)* y *LinkedList(Collection c)*: Estos constructores crean una lista que inicialmente contiene a los elementos de la colección que se le pasa como argumento.

Adicionalmente, la clase *ArrayList* tiene el constructor *ArrayList(int initialCapacity)*. Este constructor permite crear una lista vacía con una capacidad inicial para el número de elementos que se ha indicado como argumento. Aunque una lista puede tener cualquier número de elementos, al implementar esta clase la lista como un vector, éste debe tener un tamaño fijo. Cuando no hay suficiente capacidad en el vector, la clase reserva más memoria agrandando la capacidad del mismo. Esta operación requiere un cierto tiempo para llevarse a cabo, por lo que si se conoce que una lista de esta clase tendrá inicialmente un número concreto de elementos, es conveniente emplear este constructor para evitar operaciones de ampliación de memoria innecesarias.

5. La interfaz ListIterator

La interfaz *ListIterator*, perteneciente al paquete *java.util*, hereda de la interfaz *Iterator* extendiendo ésta para incorporar nuevos métodos para recorrer una lista hacia atrás, insertar nuevos elementos y reemplazar elementos.

Este tipo de iterador dispone de un "puntero" para controlar el recorrido sobre la colección. Este puntero no apunta directamente a un elemento, sino que entre cada par de elementos, y también antes del primero y después del último existe una posible posición del puntero. Por ejemplo, para una colección de tres elementos tiene los siguientes huecos:



Como puede observar, los elementos se numeran desde 0 hasta *n-1*, pero las posibles localizaciones del puntero van desde 0 hasta *n*, siendo *n* el número de elementos de la lista. Este tipo de iterador adopta este sistema de punteros para poder devolver en cualquier momento el elemento anterior o el siguiente.

Los **nuevos** métodos que incorpora la interfaz, ya que también incluye los métodos de la interfaz *Iterator* debido a la herencia, son los siguientes:

- *Object previous( )*: Este método devuelve una referencia al objeto situado en la posición anterior al hueco al que apunta actualmente el puntero, permitiendo al usuario recorrer la colección hacia atrás. Si no hay un elemento anterior a la posición del puntero, el método lanzará una excepción de la clase *NoSuchElementException*, hija de *RuntimeException*.

- *boolean hasPrevious( )*: El método devuelve verdadero si existe un elemento de la colección previo a la posición en la que apunta actualmente el puntero que controla el recorrido. En otro caso devolverá el valor falso.

- *int nextIndex( )*: Devuelve el índice del elemento que sería devuelto en la próxima llamada al método *next*. En caso del que el puntero esté situado al final de la lista devolverá el tamaño de la misma.

- *int previousIndex( )*: Este método devuelve el índice del elemento que sería devuelto en la próxima llamada al método *previous*. En el caso de que el puntero de la lista esté situado al principio de la misma devolverá -1.

- *void add(Object)*: Inserta el objeto en la lista en la posición actual del puntero y aumenta en uno su valor: la siguiente llamada a *next* quedaría sin afectar, pero *previous* devolvería el elemento recién insertado. Los valores de los índices de elementos posteriores son incrementados en uno. El método puede lanzar varios tipos de excepciones, siendo el tipo más importante las excepciones de la clase *UnsupportedOperationException*, que indican que la lista no permite insertar elementos ya que ésta es una operación opcional que no todas las listas soportan.

- *void set(Object):* Este método reemplaza en la lista el objeto cuya referencia devolvió la última llamada a *next* o *previous* por el objeto que se pasa como argumento. Debido a que este método es opcional es posible que el método lance una excepción de la clase *UnsupportedOperationException*. En caso de que no se haya llamado previamente al método *next* o *previous*, o que los últimos métodos que hayan sido llamados sean *add* o *remove*, el método lanzará una excepción de la clase *IllegalStateException*.

Bibliografía Básica

Documentación de la API de Java:

- Listas: http://docs.oracle.com/javase/tutorial/collections/interfaces/list.html
- Especificación de la interfaz *List*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/List.html
- Especificación de la clase *Collections*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Collections.html
- Algoritmos para la manipulación de colecciones: http://docs.oracle.com/javase/tutorial/collections/algorithms/index.html
- Ordenación de objetos: http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html
- Especificación de la interfaz *Comparable*: http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Comparable.html
- Especificación de la interfaz *Comparator*:http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Comparator.html
- Especificación de la clase *Collections*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Collections.html
- Especificación de la interfaz *ListIterator*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/ListIterator.html
- Implementaciones de la interfaz *List:* *http://docs.oracle.com/javase/tutorial/collections/implementations/list.html*
- Especificación de la clase *ArrayList*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/ArrayList.html
- Especificación de la clase *LinkedList*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/LinkedList.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<Double> 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<Integer> 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<Integer> 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.

E4. a) El siguiente código corresponde a una clase diseñada para que sus objetos alberguen un número entero del 1 al 100 de origen aleatorio. Observe como su constructor sin argumentos inicializa automáticamente el atributo a un número aleatorio. La clase implementa la interfaz *Comparable*, por lo que tiene un orden natural definido. Examine el código y determine cuál es el orden natural que se ha definido. ¿Por qué se hace un *casting* del objeto que el método *compareTo* recibe como argumento?

```
public class Numero implements Comparable {
    private int numero;

    public Numero() {
        this.numero = (int)Math.round(Math.random()*100);
    }

    public int getNumero(){return numero;}

    public int compareTo(Object o){
        Numero n = (Numero) o;
        if(this.numero==n.numero) return 0;
        else if (this.numero<n.numero) return -1;
        else return 1;
    }

    public String toString(){
        return String.valueOf(numero);
    }
}
```

b) Ejecute el siguiente programa y compruebe el funcionamiento del método *sort* y de la aplicación del orden natural definido para la clase.

```
import java.util.*;

public class Experimento4 {

    public static void main(String[] args){
        List l= new ArrayList();
        Iterator it;
        for(int i=0;i<10;i++)
            l.add(new Numero());

        System.out.println("Antes de ordenar:");

        it = l.iterator();
        while(it.hasNext()){
            System.out.print((Numero)it.next());
            if(it.hasNext())
                System.out.print(", ");
        }

        System.out.println("\nDespués de ordenar:");

        Collections.sort(l);

        it = l.iterator();
        while(it.hasNext()){
            System.out.print((Numero)it.next());
            if(it.hasNext())
                System.out.print(", ");
        }
    }
}
```

E5. a) Estudie el código de la siguiente clase y su interfaz asociada. Esta interfaz está diseñada para representar personas de las cuales se almacena su edad.

```
public interface IPersona {
    public int getEdad();
    public void setEdad(int edad);
```

```
}

public class Persona implements IPersona {
    private int edad;
    public Persona(int edad){ setEdad(edad); }
    public int getEdad(){ return edad; }
    public void setEdad(int edad){ this.edad = edad; }
    public String toString(){ return String.valueOf(edad); }
}
```

b) Analice el código del siguiente programa. Su objetivo es obtener la edad máxima y mínima de una colección de personas que se crea en el mismo programa. Ejecútelo para ver el resultado.

```
import java.util.*;

public class Experimento5 {
    public static void main(String[] args){
        Collection c = new ArrayList();

        for(int i=0;i<5;i++)
            c.add(new Persona(i*10));

        System.out.println("Edades: " + c);
        System.out.println("Maxima edad: " + Collections.max(c));
        System.out.println("Minima edad: " + Collections.min(c));
    }
}
```

c) ¿Cuál es la solución para conseguir que el programa funcione correctamente?

E6. a) Si deseamos ordenar los objetos de la clase *Numero* con respecto a otro orden distinto al natural deberemos definir un comparador. Examine el código del siguiente comparador para la clase *Numero* y determine que orden impone a los objetos.

```
import java.util.Comparator;

public class ComparadorNumero implements Comparator {

    public int compare(Object o1, Object o2){
        Numero n = (Numero) o1;
        Numero m = (Numero) o2;

        if(n.getNumero()==m.getNumero()) return 0;
        else if (n.getNumero()>m.getNumero()) return -1;
        else return 1;
    }
}
```

b) Modifique el código del programa principal del experimento anterior para que se aplique en la ordenación el criterio de orden establecido por el anterior comparador.

E7. El siguiente programa proviene de sustituir la clase *Numero* por la clase *Integer* en el experimento 4 y la aplicación del orden definido anteriormente. Ejecute dicho programa ¿Cuál es la razón de la excepción?

```
import java.util.*;

public class Experimento7 {

    public static void main(String[] args){
        List l= new ArrayList();
        Iterator it;
        for(int i=0;i<10;i++)
            l.add(new Integer((int)(Math.random()*100)));

        System.out.println("Antes de ordenar:");

        it = l.iterator();
        while(it.hasNext()){
            System.out.print((Integer)it.next());
```

```
            if(it.hasNext())
                System.out.print(", ");
        }

        System.out.println("\nDespués de ordenar:");

        Collections.sort(l,new ComparadorNumero());

        it = l.iterator();
        while(it.hasNext()){
            System.out.print((Integer)it.next());
            if(it.hasNext())
                System.out.print(", ");
        }
    }
}
```

E8. a) El siguiente código tiene como objetivo recorrer una lista hacia atrás. Corrija el código, ejecútelo y observe la salida.

```
import java.util.*;

public class Experimento8 {

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

        for(int i=0;i<5;i++){ // Añadimos letras
            l.add(new Character( (char)('a'+i) ));
        }
        System.out.println(l);

        it = l.listIterator(l.size()-1);
        while(it.hasPrevious())
            System.out.println(it.previous().charValue());
    }
}
```

b) ¿Por qué no funciona como se esperaba? Solucione el problema

Ejercicios

EJ1. (20 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.

EJ2. (20 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. (20 mins.) Nos disponemos a crear una clase que implemente una interfaz para representar empleados. Cada empleado tendrá su nombre, apellidos, edad, sueldo y fecha de incorporación.

a) Cree una interfaz *IFecha* que contenga los métodos consultores y modificadores para los atributos *dia, mes* y *anyo* de tipo entero. Cree una clase *Fecha* que implemente la interfaz *IFecha*.

b) Cree una interfaz *IEmpleado* que contenga los métodos consultores y modificadores para los atributos *nombre* y *apellidos* de la clase *String*, *edad* de tipo entero, *sueldo* de tipo flotante, y *fechaDeIncorporacion* de tipo *IFecha*.

c) Cree una clase *Empleado* que contenga como atributos privados los citados en el punto anterior. La clase deberá implementar la interfaz *IEmpleado* y dispondrá de un constructor que reciba el mismo número de argumentos, y del mismo tipo, que los atributos de la clase y los inicializará al valor especificado por los argumentos. Esta clase deberá redefinir el método *toString* para que éste devuelva una descripción del empleado en función del valor de los atributos.

d) Defina como orden natural de la clase anterior un orden tal que se ordenen los empleados por orden alfabético en función de sus apellidos y posteriormente, si los apellidos son iguales, en función del nombre.

e) Cree un programa principal que demuestre el correcto funcionamiento del método anterior.

EJ3. (10 mins.) a) Cree un comparador que permita ordenar a los empleados de una lista en función de su edad, de tal forma que se muestren los de mayor edad primero.

b) Cree un programa principal para demostrar el funcionamiento del comparador.

EJ4. (20 mins.) a) Cree una subclase de *ArrayList*, llamada *OrderedArrayList*. Esta clase implementará una lista en la que cuando se inserta un elemento éste lo hace de tal forma que los elementos de la lista estén ordenados. Para ello, la nueva clase deberá de disponer de tantos constructores como la original, y los implementará haciendo una llamada a los constructores de la clase padre. Además, deberá redefinir el método add(Object o). La nueva implementación de este método deberá buscar la posición correcta del elemento de la lista de forma tal que los elementos queden ordenados de menor a mayor. Para ello, se convertirá toda referencia de objeto a una instancia de la interfaz *Comparable* mediante un *casting*, lo que nos permitirá comparar los elementos de la lista siempre que éstos implementen dicha interfaz. Para insertar un elemento en la lista se recorrerá la misma hasta encontrar el primer elemento que sea mayor que el elemento a insertar. El nuevo elemento se incorporará en la lista en la posición anterior al primer elemento de la lista mayor que él. Para realizar esta operación de búsqueda e inserción utilice un *ListIterator*.

Nota: No todos los objetos implementan la interfaz *Comparable*. Si los objetos sobre los que se hace el casting no implementan dicha interfaz Java provocará una excepción. Para realizar pruebas use objetos de la clase String, o de las clases de envoltura, ya que éstos implementan dicha interfaz.

b) Cree un programa principal que demuestre el funcionamiento de la nueva clase.

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}

P7. (20 mins.) a) Cree una clase, y su interfaz asociada, destinada a representar rectángulos para lo que se dispondrá de su largo y alto. La clase deberá definir un orden natural para sus objetos de tal forma que se ordenen de forma creciente en función de su área. El área de un rectángulo es el producto de su largo por su ancho.

b) Cree un par de comparadores que permitan ordenar de forma ascendente una lista de rectángulos. El primero en función de su largo, y en caso de que lo largos sean iguales en función de su ancho. El segundo ordenará en función del ancho y en caso de igualdad, en función del largo.

c) Cree un programa principal que demuestre el funcionamiento de los órdenes definidos en los apartados anteriores.

P8. (10 mins.) a) Cree un comparador para números enteros, de tal forma que ordene una lista de números enteros colocando primero a los pares y posteriormente a los impares. Los pares estarán ordenados de forma ascendente y los impares se ordenarán de forma descendente.

b) Cree un programa que demuestre el correcto funcionamiento del comparador.

P9. (40 mins.) a) Implemente un método estático que permita ordenar una lista en función del orden natural de sus elementos empleando el algoritmo de ordenación burbuja. Este algoritmo avanza por la lista, y cuando encuentra dos elementos adyancentes que no siguen el orden los intercambia. El procedimiento anterior se repite tantas veces como sea necesario hasta que en una pasada por la lista no se encuentren elementos desordenados. Para recorrer la lista y cambiar sus elementos emplee exclusivamente un *ListIterator*.

b) Cree una sobrecarga del método anterior que ordene aplicando el orden no natural definido por un comparador que usará como argumento.

c) Cree un programa principal que demuestre el funcionamiento de ambos métodos.

P10. (35 mins.) Implemente los mismos métodos de ordenación pero que aplicando el algoritmo de inserción usando para ello dos *ListIterator*.

P11. (20 mins.) a) Cree varias clases, y sus interfaces asociadas, que permitan contener las estadísticas de puntos de los jugadores de baloncesto de una liga. Para ello será necesario representar mediante clases a los equipos, y a los jugadores. Cada equipo estará compuesto por una lista de jugadores, y cada jugador dispondrá de un contador para las canastas de un punto, un contador para las canastas de dos puntos, y un contador para las canastas de tres puntos.

b) Defina para la clase que representa a los jugadores un orden natural, que será descendente en función de los puntos anotados por el jugador.

c) Defina un orden natural para la clase que representa a los equipos de tal forma que éste sea descendente en función del número de puntos anotados por los jugadores del equipo. Es decir, un equipo A será menor que un equipo B cuando la suma total de puntos de todos los jugadores del equipo A sea menor que la suma total de puntos de todos los jugadores del equipo B.

e) Defina un comparador para la clase que representa a los jugadores de forma que ordene a éstos de forma decreciente en función del número de triples anotados, posteriormente en función del número de canastas de dos puntos y finalmente en función del número de canastas de un punto.

f) Cree un programa que muestre el funcionamiento de los comparadores definidos anteriormente.

P12. (20 mins.) a) Cree una interfaz llamada *IModeloCoche* para representar coches. La interfaz contendrá los métodos consultores y modificadores para los siguientes datos:

- Año de fabricación (anyo) de tipo entero.
- Velocidad máxima (velocidad) de tipo entero.
- Precio (precio) de tipo flotante.
- Nombre del modelo (nombre) de tipo *String*.
- Marca del modelo (marca) de tipo *String*.

b) Cree una clase llamada *ModeloCoche* que implemente la anterior interfaz. Esta clase contendrá un método constructor con tantos argumentos como atributos que permitirá inicializar el valor de los mismos en el momento de creación.

c) Cree una interfaz llamada *IModelosCoches* para manipular una lista de coches. La interfaz dispondrá de un método *add* para añadir nuevos coches a la lista. Cree la clase *ModelosCoches* que implemente la interfaz anterior. Su constructor por defecto creará una lista vacía.

d) Añada a la anterior interfaz un método que permita imprimir por pantalla un informe de los modelos de coches que figuran en la lista en el que conste toda la información que se dispone de los mismos. El informe debe estar ordenado por el nombre del modelo del coche y si el nombre de dos modelos de coches coincide se ordenará por el año de fabricación. La clase *ModelosCoches* debe seguir implementando la interfaz. Este orden será el orden natural de los objetos de la clase *ModeloCoche*, por tanto modifique ésta convenientemente.

d) Cree un método similar al anterior en el que se imprima por pantalla un informe ordenado por el año de fabricación, de tal forma que se muestren primero los modelos de fabricación más reciente.

e) Añada otro método similar al anterior que muestre un informe de los distintos modelos de coches según su velocidad máxima, de forma tal que se muestren los de mayor velocidad máxima primero.

f) Cree nuevos métodos para permitir la impresión por pantalla de los informes anteriores en orden inverso.

g) Cree varios programas para probar el funcionamiento de los métodos desarrollados en apartados anteriores.

P13. (20 mins.) a) Cree una interfaz *IAlumno* para manipular alumnos. La interfaz contendrá los métodos consultores y modificadores para los siguientes datos: nombre (tipo *String*), apellidos (tipo *String*), edad (tipo entero), y nota del expediente (tipo flotante).

b) Cree una clase llamada *Alumno* que implemente la interfaz anterior.
c) Cree una interfaz *IAlumnos* para manipular una lista de alumnos. La interfaz dispondrá de un método para añadir alumnos. Cree la clase *Alumnos* que implemente la interfaz anterior. Su constructor por defecto creará una lista vacía.

d) Añada a la clase anterior varios métodos, tantos como atributos tiene la clase *Alumno*, que nos permitan buscar a alumnos en función del valor de alguno de sus atributos. Estos métodos devolverán el primer alumno encontrado, o una referencia nula (*null*) si no se ha encontrado un alumno de esas características en la lista. La búsqueda se hará empleando comparadores, consulte la documentación de la clase *Collections* para ver la especificación de la sobrecarga del método *binarySearch* que acepta como argumento un comparador. Estos métodos tendrán los siguientes prototipos:

- *IAlumno buscaPorNombre(String nombre).*
- *IAlumno buscaPorApellidos(String apellidos).*
- *IAlumno buscaPorEdad(int edad).*
- *IAlumno buscaPorNota(float nota).*

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
- Thinking in Java, 3rd Edition. Bruce Eckel. Prentice Hall, 2002. Capítulo 11. http://www.mindview.net/Books/TIJ/
- Aprenda Java como si Estuviera en Primero. Capítulo 4, 71 a 74 http://mat21.etsii.upm.es/ayudainf/aprendainf/Java/Java2.pdf