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

Enseñanzas de Prácticas y Desarrollo
EPD-6: Set y SortedSet

Objetivos

- Conocer los conceptos de conjunto y conjunto ordenado.
- Manipular conjuntos y conjuntos ordenados de objetos en Java.
- Crear conjuntos basados en las clases *HashSet y TreeSet*.

Conceptos

1. Concepto de Conjunto

Los conjuntos se definen como agrupaciones de elementos en las que no se pueden incluir elementos duplicados y en las que los elementos no están dispuestos en una determinada secuencia.

Que un conjunto no pueda incluir elementos duplicados significa que no se podrá añadir a un conjunto más de una referencia a un mismo objeto, o referencias a objetos que se consideren iguales. Para el caso de los conjuntos, dos referencias a objetos se considerarán duplicadas si:

- Son idénticas: Las dos referencias hacen referencia al mismo objeto.

- Son iguales: Aunque las referencias apunten a objetos distintos, éstos son considerados iguales por la implementación del método *equals* y por extensión por el método *hashCode*.

Recuerde que si deseamos que dos objetos de una misma clase se consideren iguales sin ser idénticos deberemos sobrescribir los métodos *equals* y *hashCode* heredados de la clase *Object*. Estos métodos tienen la siguiente definición:

- *public boolean equals(Object obj)*: El método recibirá como argumento el objeto con el que se desea comparar el objeto que invoca el método (objeto *this*). Este método devolverá el valor verdadero si los objetos comparados se consideran iguales o falso en caso contrario.

- *public int hashCode( )*: Este método devuelve un número entero que identificará al objeto. Si dos objetos se consideran iguales por el método *equals*, éste deberá devolver el mismo número entero. Este método es fundamental para las implementaciones de colecciones que emplean internamente vectores asociativos (todas las implementaciones cuya clase contenga la palabra *hash*). Una forma sencilla y genérica de implementar este método, y que se debe seguir como regla general, es que si un atributo es el que determina que dos objetos son iguales se devolverá el valor que devuelve el método *hashCode* de su clase. Para el caso en que el tipo sea primitivo, se recurrirá al valor devuelto por el método *hashCode* de la clase de envoltura que le corresponda. Cuando sean varios los atributos que determinen si dos objetos son iguales sume los valores enteros generados para cada uno de los atributos siguiendo la regla anterior.

Si en una clase no se sobrescriben los métodos anteriores se empleará la implementación heredada de la clase *Object*. Para el caso del método *equals*, esta implementación considera que dos objetos son iguales sólo si éstos son idénticos. En el caso del método *hashCode*, la implementación devuelve un entero basado en la posición en memoria que ocupa el objeto.

Otra particularidad de los conjuntos es que sus elementos no están organizados en una secuencia. Este hecho implica que un iterador sobre una colección de este tipo irá ofreciendo cada uno de los objetos de la colección en un orden que muy probablemente no coincidirá con el orden en el que fueron insertados los elementos de dicha colección, al contrario de lo que ocurre con las listas en las que el orden de inserción, o el que se haya establecido posteriormente, sí se mantiene.

2. La interfaz Set.

Toda clase de JCF que represente un conjunto ha de implementar la interfaz *Set*, que al igual que las interfaces y clases de JCF se encuentra en el paquete *java.util* y por tanto deberá importar dicho paquete cuando use esta interfaz. Esta interfaz hereda de la

interfaz *Collection* por lo que contiene todos los métodos de la citada interfaz que se vieron en guiones anteriores. Los métodos más importantes para manejar conjuntos son los siguientes:

- *boolean add(Object obj)*: Permite insertar el objeto que se pasa como argumento al conjunto. Este método ya está incluido en la interfaz *Collection*, pero para el caso de conjuntos el valor devuelto por éste es relevante, al contrario que para otro tipo de colecciones en las que se permiten duplicados. El método devolverá verdadero si se pudo realizar la inserción del citado objeto o falso en caso contrario. Cuando se indica que un objeto no se ha podido incluir en el conjunto, el motivo de esta negativa es que el objeto, o un objeto igual, ya pertenece a la colección.

- *boolean addAll (Collection col)*: Incluye en el conjunto todos los elementos de la colección que se pasa como argumento que no pertenecieran previamente al citado conjunto, es decir, realiza la operación unión de conjuntos (*this* = *this* $\cup$ *col*). Al igual que el anterior, este método está incluido en la interfaz *Collection* y el valor que devuelve es relevante en contraste con las colecciones que admiten duplicados. El método devolverá verdadero en caso de que se haya podido insertar al menos un elemento de la colección, o falso en caso contrario.

- *void clear( )*: Elimina todos los elementos del conjunto dejando éste vacío.

- *boolean contains(Object obj)*: Indica si el objeto que se pasa como argumento pertenece al conjunto, devolviendo verdadero en caso afirmativo o falso en caso contrario.

- *boolean containsAll(Collection col)*: Este método devolverá verdadero si todos los elementos de la colección que se pasa como argumento son elementos del conjunto, o lo que es lo mismo devolverá verdadero si *col* es un subconjunto del conjunto que invoca el método (objeto *this*). El método devolverá falso en caso contrario.

- *boolean equals(Collection col)*: Este método indica si la colección que se pasa como argumento es igual al conjunto, devolviendo verdadero en caso de que éstas se consideren iguales o falso en caso contrario. Para el caso de la interfaz *Set*, la colección que se pasa como argumento se considerará igual al conjunto sobre el que se invoca el método (*this*) si, y solo sí, dicha colección es un conjunto (implementa la interfaz *Set*), ambos conjuntos tienen el mismo número de elementos, y todo elemento perteneciente a uno de los conjuntos está contenido en el otro.

- *boolean isEmpty( )*: El método indica si el conjunto está vacío, devolviendo verdadero en caso de que lo esté o falso en caso contrario.

- *boolean remove(Object obj):* Elimina del conjunto al objeto que se pasa como argumento. El método devolverá verdadero si el conjunto contenía al objeto o falso en caso contrario.

- *boolean removeAll(Collection col)*: Elimina del conjunto todos los objetos pertenecientes a la colección que se pasa como argumento, es decir, la operación diferencia de conjuntos (*this* = *this* – *col*). El método devolverá verdadero cuando al menos se ha eliminado un elemento del conjunto o falso en caso contrario.

- *boolean retainAll(Collection col)*: Elimina del conjunto todos aquellos elementos que no estén contenidos en la colección que se pasa como argumento, es decir, realiza una intersección del conjunto con la colección realizando la intersección de conjuntos (*this* = *this* $\cap$ *col*). Devolverá verdadero si se ha eliminado al menos un elemento o falso en caso contrario.

- *int size( )*: Devuelve el número de objetos que pertenecen al conjunto. Matemáticamente hablando, se devuelve el cardinal del conjunto.

- *Object[] toArray( )*: Devuelve un vector que contiene todos los objetos que forman parte del conjunto.

3. La clase HashSet.

La clase *HashSet*, incluida en el paquete *java.util* como parte de JCF, es una implementación de la interfaz *Set*, y por tanto esta clase permite representar conjuntos. Esta clase realiza la implementación de un conjunto empleando un vector asociativo, de ahí la palabra "*hash*" en su nombre. Que la clase esté implementada internamente usando un vector asociativo significa en términos prácticos que la inserción, borrado y búsqueda de elementos en el conjunto llevará un tiempo reducido y prácticamente constante, en contraste con lo que ocurre en el caso de las implementaciones de listas.

La clase posee los siguientes métodos constructores:

- *public HashSet(int tamaño, float factorCarga)*: Crea un conjunto empleando un vector asociativo con un espacio inicial para el número de elementos indicado por el argumento *tamaño*. El factor de carga, indicado por el argumento *factorCarga*, sirve para indicar el número de elementos que ha de contener el conjunto para que se amplíe automáticamente la capacidad del vector asociativo, significando cada ampliación automática la duplicación de la capacidad de dicho vector asociativo. Este número límite de elementos se calculará multiplicando el factor de carga por el la capacidad del vector asociativo. Por ejemplo, para un factor de carga de 0.8 y una capacidad máxima de 10 elementos la colección ampliará su tamaño cuando contenga 8 (0.8 * 10) elementos.

- *public HashSet(int tamaño)*: Crea un conjunto que empleará un vector asociativo del tamaño indicado por el argumento y un factor de carga de 0.75.

- *public HashSet( )*: Crea un conjunto que empleará un vector asociativo con un tamaño inicial de 16 elementos y un factor de carga de 0.75.

- *public HashSet(Collection col)*: Crea un conjunto que contendrá los elementos de la colección que se pasa como argumento. El vector asociativo que empleará tendrá un tamaño suficiente para contener a los elementos de la colección y un factor de carga de 0.75.

Como norma general crearemos un conjunto vacío empleando los valores por defecto para la capacidad y factor de carga. Para ello emplearemos una sentencia similar a la siguiente:

```
Set s = new HashSet();
```

4. Concepto de conjunto ordenado.

Un conjunto ordenado es un conjunto en el que sus elementos se organizan en memoria por un orden determinado, de tal forma que queden almacenados en orden ascendente.

Al ser conjuntos, los conjuntos ordenados no permiten contener elementos duplicados. Recuerde que para determinar si dos objetos son iguales la clase deberá redefinir el método *equals* heredado de la clase *Object*, y por extensión el método *hashCode*.

El que los conjuntos sean ordenados no implica que sus elementos guarden una determinada secuencia, simplemente se almacenan en memoria de tal forma que el acceso a ellos en función de su orden sea rápido. El único criterio determinante en este tipo de colecciones es el orden establecido para los elementos de la colección, y por tanto los elementos de un conjunto ordenado deben tener asignado algún tipo de orden.

Como se estudió en prácticas anteriores, los objetos de una clase pueden tener asignado un orden natural y uno o varios órdenes no naturales. Los elementos de un conjunto ordenado deberán disponer de al menos uno de esos tipos de órdenes, ya que como se ha indicado anteriormente el orden elegido determinará la forma en la que los elementos del conjunto ordenado son almacenados.

Recuerde que para que los objetos de una clase dispongan de un orden natural esta clase deberá implementar la interfaz *Comparable*. Asimismo, recuerde que si deseamos definir un orden no natural para los elementos de una clase deberemos crear un comparador para la clase, esto es, crear una clase que implemente la interfaz *Comparator*.

5. La interfaz SortedSet.

Toda clase que permita almacenar objetos en forma de conjunto ordenado deberá implementar la interfaz *SortedSet*, incluida en el paqueta *java.util*. Esta interfaz hereda de la interfaz *Set*, y por tanto incluye todos los métodos definidos en dicha interfaz. La interfaz incluye los siguientes métodos propios para la manipulación de conjuntos ordenados:

1. *Object first( )*: Este método devuelve el primer objeto, según el orden establecido, de la colección. Como los conjuntos ordenados están ordenados de forma ascendente, el primer objeto de la colección también es el menor. El método lanzará una excepción de la clase *NoSuchElementoException* en caso de que el conjunto esté vacío.

2. *Object last( )*: El método devolverá el último objeto de la colección, también en este caso según el orden establecido para los elementos de la colección. Teniendo en cuenta el orden ascendente en que se organizan los conjuntos ordenados, el último elemento del conjunto será también el mayor elemento del mismo. Este método lanza una excepción de la clase *NoSuchElementExcepcion* si el conjunto está vacío.

3. *SortedSet subSet(Object min, Object max)*: Este método devuelve una vista del conjunto ordenado que invoca que representa un subconjunto ordenado del mismo. Los objetos incluidos en dicho subconjunto deben cumplir las condiciones *min<=objeto* y *objeto<max* según el orden en que se organizan los elementos de la colección. Tenga en cuenta que los objetos *min* y *max* no tienen que ser elementos de dicha colección, ya que éstos simplemente representan unos límites al subconjunto seleccionado.

4. *SortedSet headSet(Object max)*: El método devuelve una vista del conjunto ordenado en la que están incluidos todos los elementos que cumplen la condición *objeto<max*. Recuerde que *max* no tiene que ser un elemento del conjunto.

5. *SortedSet tailSet(Object min)*: Este método devuelve una vista del conjunto ordenado en la que se incluyen todos los elementos que cumplen la condición *objeto>=min*. De igual forma, *min* no tiene obligatoriamente que ser un elemento del conjunto.

6. *Comparator comparator( )*: Devuelve una referencia al comparador que se está usando para ordenar los elementos en el caso de que dicho orden sea un orden no natural. Si el orden por el que se organizan los elementos del conjunto es el orden natural asociado a su clase, entonces este método devolverá *null*.

6. La clase TreeSet.

La clase *TreeSet* es una implementación de la interfaz *SortedSet* y está incluida en el paquete *java.util*. Esta clase permite representar conjuntos ordenados, usando para ello un tipo de árbol binario llamado *red-black tree*. La clase ofrece los siguientes métodos constructores:

- *public TreeSet( )*: Este constructor crea un conjunto ordenado en el que se empleará el orden natural de los elementos como orden para organizar los elementos del conjunto.

- *public TreeSet(Collection col)*: El constructor crea un conjunto ordenado que contendrá los elementos de la colección que se pasa como argumento. Para organizar los elementos del conjunto ordenado se empleará el orden natural de dichos elementos.

- *public TreeSet(Comparator comp)*: Este método constructor creará un conjunto ordenado en el que se organizan los elementos de dicho conjunto en función de un orden no natural definido por el comparador que se pasa como argumento.

- public TreeSet(SortedSet ss): Crea un conjunto ordenado de características idénticas al que se le pasa como argumento, es decir, un conjunto ordenado que contiene los elementos del conjunto ordenado que se le pasa como argumento y cuyo criterio de orden será el mismo que el aplicado en dicho conjunto ordenado.

Bibliografía Básica

Documentación de la API de Java:

- Documentación de la API de Java:
- Tutorial sobre la interfaz *Set*: http://docs.oracle.com/javase/tutorial/collections/interfaces/set.html
- Especificación de la interfaz *Set*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/Set.html
- Especificación de la clase *HashSet*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/HashSet.html
- Tutorial sobre la interfaz *SortedSet*: http://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html
- Especificación de la interfaz *SortedSet*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/SortedSet.html
- Especificación de la clase *TreeSet*: http://docs.oracle.com/javase/1.5.0/docs/api/java/util/TreeSet.html

Experimentos

E1. a) Analice el código del siguiente programa. Ejecútelo para ver la salida por pantalla que genera.

```
public class Experimento1 {
```

```
    public static void main(String[] args){
        List l = new ArrayList();
        Set s = new HashSet();
        Iterator it;

        for(int i=5;i>=1;i--){
            l.add(i);
            s.add(i);
        }

        it = l.iterator();
        while(it.hasNext())
            System.out.println((Integer)it.next());

        System.out.println("------------------");

        it = s.iterator();
        while(it.hasNext())
            System.out.println((Integer)it.next());
    }
}
```

b) ¿Por qué el orden de los elementos no coincide en los dos tipos de colecciones?

c) Varíe la iteración del bucle *for* para que sea creciente ¿Por qué ahora el orden coincide?

E2. a) Estudie el código del siguiente programa y determine el objetivo del mismo. Ejecute el programa para obtener su salida por pantalla.

```
import java.util.*;

public class Experimento2 {
    public static void main(String[] args){
        List l = new ArrayList();
        Set s;

        l.add("yo");l.add("soy");l.add("yo");
        l.add("tu");l.add("eres");l.add("tu");
        System.out.println("l: " + l);

        s = new HashSet(l);
        System.out.println("s: " + s);
    }
}
```

b) Explique razonadamente los motivos que provocan la salida por pantalla obtenida.

E3. a) El siguiente programa rellena una colección ordenada con palabras y posteriormente consulta el contenido de dicha colección. Analice el código y complete los huecos que faltan.

```
import java.util.*;

public class Experimento1 {

    public static void main(String[] args){
        String[] palabras = {"en","un","lugar","de","la","mancha"};
        SortedSet ss= new _____________;

        for(int i=0;i<palabras.length;i++)
            ss.add(palabras[i]);
        System.out.println("ss: " + ss);

        System.out.println("Primera palabra: " + _________);
        System.out.println("Última palabra: " + _________);
        System.out.println("Palabras entre la letra g y la r: " + _________);
        System.out.println("Palabras desde la palabra \"lugar\" : " + _________);
        System.out.println("Palabras hasta la palabra \"lugar\" : " + _________);
    }
}
```

b) ¿Por qué la palabra “lugar” aparece en el subconjunto “Palabras desde la palabra lugar” y no en el subconjunto “Palabras hasta la palabra lugar”?

E4. a) Nos disponemos a crear un par de colecciones ordenadas de personas. Analice la definición de la clase Persona y su interfaz asociada completando el método *compareTo* para que las personas se comparen según su DNI.

```
public interface IPersona {
    public String getDni();
    public String getNombre();
    public int getEdad();
    public int compareTo(Object o);
    public String toString();
}

public class Persona implements IPersona, Comparable {
    private String dni;
    private String nombre;
    private int edad;

    public Persona(String dni,String nombre,int edad){
        this.dni = dni;
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getDni(){ return dni; }
    public String getNombre(){ return nombre; }
    public int getEdad(){ return edad; }

    public int compareTo(Object o){
        IPersona p = (IPersona) o;
        return _____________________
    }

    public String toString(){
        return dni + " " + nombre + " (" + edad + ")";
    }
}
```

b) Complete los huecos para disponer de un orden no natural que compare las personas según su edad. Estudie el funcionamiento del comparador.

```
import java.util.Comparator;

public class OrdenarPersonasPorEdad implements Comparator{

    public int compare(Object o1,Object o2){
        IPersona p1,p2;
        _____________________________
        _____________________________

        if(_____________________________)
            return -1;
        else if (_____________________)
            return +1;
        else
            return 0;
    }
}
```

Ejercicios

EJ1. (10 mins.) a) Cree una clase llamada *Numeros*, y su interfaz asociada *INumeros*, que contendrá como único atributo un conjunto de números (objetos de la clase *Integer*). La clase dispondrá de un método llamado *addNumero* que permitirá añadir un número al conjunto de números, y que devolverá un valor booleano indicando si se ha podido añadir con éxito el número que se le pase como argumento. Así mismo, la clase dispondrá de un método constructor que inicializará el atributo a un conjunto vacío.

b) Añada a la clase anterior un método llamado *getPrimos*, y modifique adecuadamente su interfaz. Este método no recibirá argumentos y devolverá un conjunto incluyendo todos los números primos presentes en el conjunto de números que almacena el objeto de la clase que invoca el método (*this*). Recuerde que un número primo es aquel que sólo es divisible por sí mismo y por 1.

c) Cree un programa que demuestre el funcionamiento de la clase *Numeros* y de sus métodos.

EJ2. (20 mins.) a) Cree una clase llamada *Palabras*, y su interfaz asociada *IPalabras*, que nos permitirá procesar palabras de la siguiente forma. Se dispondrá de un par de conjuntos (que serán dos atributos en la clase), el primero para palabras nuevas, y el segundo para palabras repetidas. Cada vez que procesemos una palabra, ésta se incorporará al conjunto de palabras nuevas, a no ser que ya se hubiese procesado previamente dicha palabra, en cuyo caso ésta deberá de incluir en el conjunto de palabras repetidas y eliminar del conjunto de palabras nuevas.

Para procesar palabras la clase dispondrá de los siguientes métodos:

- *incluirPalabra*: Este método recibirá una palabra (*String*) y la procesará como se ha descrito anteriormente. El método devolverá verdadero si la palabra era nueva, o falso si la palabra ya estaba repetida.
- *incluirFrase*: Este método recibirá una frase, determinará las palabras que la forman, e irá procesando una a una las palabras de la misma haciendo uso del método anterior. El método devolverá verdadero en el caso de que todas las palabras de la frase fuesen nuevas o falso en el caso de que alguna de las palabras de la frase estuviesen repetidas.

Además de los métodos anteriores la clase dispondrán de los siguientes métodos:

- *getPalabrasNuevas*: Este método devolverá el conjunto de palabras que no han sido repetidas.
- *getPalabrasRepetidas*: Este método devolverá el conjunto de palabras que han sido repetidas.
- *getTodasLasPalabras*: Este método devolverá el conjunto de todas las palabras que han sido procesadas.

La clase dispondrá de un constructor sin argumentos que tendrá como única tarea inicializar los atributos para representar los conjuntos de palabras nuevas y repetidas, de forma que éstos se inicialicen cada uno a un conjunto vacío.

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

EJ3. (30 mins.) Necesitamos crear un programa que acepte como entrada las diferentes ofertas que se han hecho para una venta por Internet al mejor postor. El programa deberá tomar el nombre del oferente y la oferta que hace. Una vez entradas todas las ofertas se imprimirá por pantalla la oferta ganadora y un informe del resto de ofertas ordenado de forma decreciente por la cuantía de ofertas como lista de suplentes. Para ello, deberá crear los siguientes elementos:

a) Cree una clase, y su interfaz asociada, para representar las ofertas. La clase deberá poseer los atributos necesarios para contener la información de cada oferta, un constructor que permita inicializar cómodamente dichos atributos, métodos consultores y modificadores para acceder y variar el valor de los atributos, así como un método *toString* adecuado para crear las diferentes líneas del informe.

b) Dote a los objetos de la anterior clase de un orden natural de tal forma que las ofertas se ordenen de forma decreciente según la cuantía de la oferta.

c) Cree una clase llamada Venta, y su interfaz asociada, que contenga en un conjunto ordenado las diferentes ofertas. La clase deberá disponer de un método para incorporar las ofertas según se van obteniendo de funcionamiento similar al método *add* de la interfaz *Collection*, y un constructor sin argumentos que inicializará el conjunto ordenado a un conjunto vacío. La clase dispondrá de un método *ganador* que devolverá la oferta ganadora, y de un método *suplentes* que devolverá un conjunto ordenado de los suplentes.
d) Cree un programa que utilizando los elementos creados en los apartados anteriores funcione tal y como se ha descrito al inicio del ejercicio. Para indicar al programa que no se introducirán más ofertas se dejará el nombre del oferente en blanco.

Problemas

P1. (75 mins.) a) Cree una clase *Empleado*, y su interfaz asociada *IEmpleado*, con los siguientes atributos:

- *nombre*: Atributo de tipo *String*.
- *dni*: Atributo de tipo *String*.
- *edad*: Atributo entero.

- *casado*: Atributo de tipo booleano.
- *nHijos*: Atributo de tipo entero que representará el número de hijos del empleado.
- *sueldo*: Atributo de tipo flotante.

Por cada atributo la clase dispondrá de un método consultor (*get*) y otro modificador (*set*). La clase dispondrá de un constructor que recibirá como argumentos los valores iniciales para los atributos. Asimismo, la clase deberá disponer de los métodos *toString*, *equals* y *hashCode* implementados de manera adecuada. Dos objetos de esta clase se considerarán el mismo si cuentan con el mismo DNI.

b) Cree una clase llamada *Empleados*, y su interfaz asociada *IEmpleados*, que contendrá el conjunto de empleados de una empresa. Esta clase contendrá el conjunto de empleados en un atributo privado. La clase dispondrá de un método *addEmpleado* que permitirá añadir un empleado al conjunto y que devolverá un valor booleano de la misma manera que lo hace el método *add* de la interfaz *Set*. La clase dispondrá de un método constructor sin argumentos que inicializará el atributo que referencia el conjunto de empleados a un conjunto vacío.

c) Añada a la clase anterior los siguientes métodos que servirán para confeccionar listados filtrados del personal de la empresa (no olvide modificar su interfaz de forma adecuada):

- *getPrejubilables*: Este método devolverá un conjunto con los empleados cuya edad sea igual o superior a los 60 años.
- *getCasados*: Este método devolverá el subconjunto de empleados casados.
- *getFamiliasNumerosas*: Este método devolverá el subconjunto de empleados que tengan una familia numerosa (con 3 o más hijos).

d) Cree un programa que, usando las clases e interfaces anteriores, muestre un listado con los datos de cada empleado y el complemento de sueldo que les corresponderá (expresado en euros) según las siguientes reglas (inicialice el conjunto de empleados como crea conveniente con objeto de realizar la prueba):

- Prejubilables: (6-n) * 3%, siendo n el número de años que le quedan para la jubilación.
- Familias numerosas: 2% por hijo que exceda la cantidad de dos hijos.
- Casados: 1%.

Estos complementos son excluyentes, y están ordenados en la lista anterior por orden de aplicación. Por ejemplo, si un empleado es prejubilable, tiene una familia numerosa y está casado sólo cobrará el complemento que está antes en la lista, es decir, el complemento de prejubilación.

En el listado se mostrarán en primer lugar todos los empleados a los que se les aplica el complemento de prejubilación, en segundo lugar a todos los que se les aplica el complemento de familia numerosa y en último lugar a todos los que se les aplica el complemento de casados.

P2. (90 mins.) a) Cree una clase llamada *Asignatura*, y su interfaz asociada, que permita representar los datos de una asignatura de una carrera universitaria. Esta clase contendrá atributos para representar el nombre de la asignatura, y el curso al que pertenece, y dispondrá de métodos consultores y modificadores para cada uno de sus atributos así como de un método constructor que recibirá como argumentos los valores necesarios para inicializar el valor de cada uno de los atributos. Dos objetos de esta clase se considerarán iguales si el nombre de la asignatura y el curso son iguales. Implemente adecuadamente los métodos *toString*, *equals* y *hashCode* para esta clase de objetos.

b) Cree una clase *Alumno*, y su interfaz asociada, que permita representar los datos de los alumnos de una facultad. Se almacenará por cada alumno su nombre, su DNI, el conjunto de asignaturas en las que está matriculado, y el conjunto de asignaturas que ya ha aprobado. Por cada atributo se dispondrá de un método consultor y otro modificador, a excepción de los atributos que almacenan conjuntos que dispondrán de un método para añadir elementos al conjunto, uno para eliminar elementos del conjunto y otro para borrar todos los elementos del conjunto así como de un método consultor que nos permita acceder directamente a dicho conjunto. Estos métodos se comportarán de forma similar a como lo hacen los métodos *add*, *remove* y *clear* de la interfaz *Set*. La clase dispondrá de un método constructor que permitirá inicializar los atributos que no almacenan conjuntos, y que inicializará los atributos que almacenan conjuntos a conjuntos vacíos. Implemente adecuadamente los métodos *toString*, *equals* y *hashCode* para esta clase de objetos considerando que dos alumnos son el mismo si tienen un mismo DNI.

c) Cree una clase *Alumnos*, y su interfaz asociada, que contendrá un conjunto de objetos de la clase Alumno. Esta clase dispondrá de métodos para añadir y eliminar alumnos del conjunto, así como un método para borrar todos los alumnos almacenados en el conjunto. El comportamiento de estos métodos será similar al de los métodos *add*, *remove* y *clear* de la interfaz *Set*. El único método constructor de esta clase no dispondrá de argumentos e inicializará el conjunto de alumnos a un conjunto vacío.

d) Añada los siguientes métodos a la clase anterior y modifique su interfaz de forma adecuada:

- *getNingunaAprobada*: Devuelve el subconjunto de alumnos que no han aprobado ninguna asignatura.
- *getMatriculados*: Este método recibe como argumento una asignatura y devuelve el subconjunto de alumnos matriculados en la misma. Se creará también una sobrecarga de este método que recibirá un conjunto de asignaturas y devolverá el subconjunto de alumnos que están matriculados en todas ellas.
- *getNoMatriculados*: Se crearán un par de métodos que funcionen de manera inversa a los métodos *getMatriculados*.
- *getAprobados*: Este método recibe como argumento una asignatura y devuelve el subconjunto de alumnos que han aprobado la misma. Además se creará una sobrecarga de este método que recibirá como argumento un conjunto de asignaturas y devuelve el subconjunto de alumnos que han aprobado todas ellas.
- *getNoAprobados*: Crear un par de métodos inversos a los métodos *getAprobados*.
- *getMatriculadosYAprobados*: Este método recibe dos conjuntos de asignaturas y devuelve el subconjunto de alumnos que están matriculados en todas las asignaturas del primer subconjunto y que han aprobado todas las asignaturas del segundo subconjunto.

e) Cree un programa que demuestre el funcionamiento de los métodos anteriores, y que demuestre la imposibilidad de incluir elementos duplicados en las colecciones. Para ello el programa deberá simular un sistema de información para la secretaría de un centro universitario.

P3. (75 mins.) a) Cree una clase llamada *Atleta*, y su interfaz asociada, que contenga los siguientes miembros:

- Atributos privados para almacenar el nombre del atleta, su dorsal y el tiempo en que ha realizado la prueba en la que compite, estando expresado dicho tiempo en horas, minutos y segundos.
- Métodos consultores y modificadores para cada uno de los atributos anteriores. Los métodos modificadores no deberán permitir un tiempo inválido, como por ejemplo -3 horas 80 minutos 70 segundos, y lanzarán una excepción de la clase *Exception* si se intenta establecer un tiempo inválido.
- Un constructor que permita establecer el valor inicial de cada uno de lo atributos de la clase según los valores que se le pasen como argumentos.
- Una implementación adecuada del método *toString* redefiniendo el que se hereda de la clase *Object*.

b) Dote a la clase anterior de un orden natural en el que los atletas queden ordenados de forma ascendente en función de su dorsal, y de un método *equals* coherente con dicho orden.

c) Cree una clase llamada *Atletas*, y su interfaz asociada, que contendrá un conjunto ordenado de objetos de la clase Atleta. La clase deberá poseer al menos los siguientes miembros:

- Un atributo privado que será una referencia al conjunto ordenado.
- Un método constructor que inicializará la referencia anterior a un nuevo conjunto ordenado vacío en el que el orden de los elementos estará definido por el orden natural que tengan asociado.
- Un par de métodos para añadir y eliminar atletas del conjunto ordenado con un funcionamiento similar a los métodos *add* y *remove* de la interfaz *Collection*.

d) Cree un par de métodos, uno para obtener el atleta con el número de dorsal menor, y otro para obtener el atleta con el número de dorsal mayor.

e) Cree un par de métodos, uno para obtener el subconjunto de atletas con dorsal anterior a uno dado, y otro para obtener los atletas con el dorsal posterior a uno dado. En ningún caso podrá estar contenido en el subconjunto un atleta con el dorsal especificado.

f) Cree un par de métodos, uno para obtener el subconjunto de atletas con un tiempo inferior, y otro para obtener el subconjunto de atletas con un tiempo superior al de un atleta que se pasa como argumento. Recuerde que para realizar esta operación deberá crear un comparador adecuado y crear un nuevo conjunto ordenado temporal regido por dicho orden.

g) Cree un programa principal que demuestre el funcionamiento de todos los métodos creados en los apartados anteriores, simulando para ello un programa de gestión de competiciones de atletismo.

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

Datos de la Práctica

Estimación temporal:

- Parte presencial: 120 minutos.
  - Explicación inicial: 35 minutos.
  - Experimentación: 25 minutos.
    - Experimento 1: 5 minutos.
    - Experimento 2: 10 minutos.
    - Experimento 3: 10 minutos.
  - Ejercicios: 60 minutos.
- Parte no presencial: 360 minutos.
  - Lectura y estudio del guión y bibliografía básica: 120 minutos
  - Problemas: 240 minutos.